RETASK: COMPILING EXECUTION RECORDS INTO TASKS FOR EVALUATION AND SUPERVISION

Anonymous authors
Paper under double-blind review

ABSTRACT

Working agents routinely produce execution records, but reusable executable tasks for evaluation and supervision remain costly to construct and retain. An execution record provides only partial evidence about task inputs and completion conditions, mixed with results produced by one solver. We ask whether a single execution record can support a reusable task for *evaluation* and *supervision* without requiring exact recovery of the original task. **ReTask** first infers a task specification from the actionable request and evidence about the initial state. It then uses the specification to construct both an unsolved initial state for a new solver and an evaluator for the solver's outcome. Artifacts and outcomes produced during the recorded execution may identify content to withhold or provide validation cases for the evaluator, but they cannot define task requirements and are never shown to the new solver. For **comparative evaluation**, the observed Native and Compiled aggregate rankings agree on 20 of 21 pairwise orders among seven agent configurations on 100 paired Workspace-Bench tasks ($\rho = 0.964$). We further analyze two record conditions: the evaluated majority-synthesis and failed-source subsets contain solvable tasks with different model outcomes. Separately, for **supervision**, 2,040 new patches pass the original repository tests: 52.5% of all 3,889 Compiled attempts and 72.7% of the 2,807 evaluable patches. On the 1,123 tasks solved in both the Native and Compiled conditions, training on Compiled trajectories improves two base models on all six evaluated SWE-bench settings by 7.00–23.84 percentage points. These results show that tasks compiled from single execution records can support model comparison and new supervision without exactly reconstructing the original task.

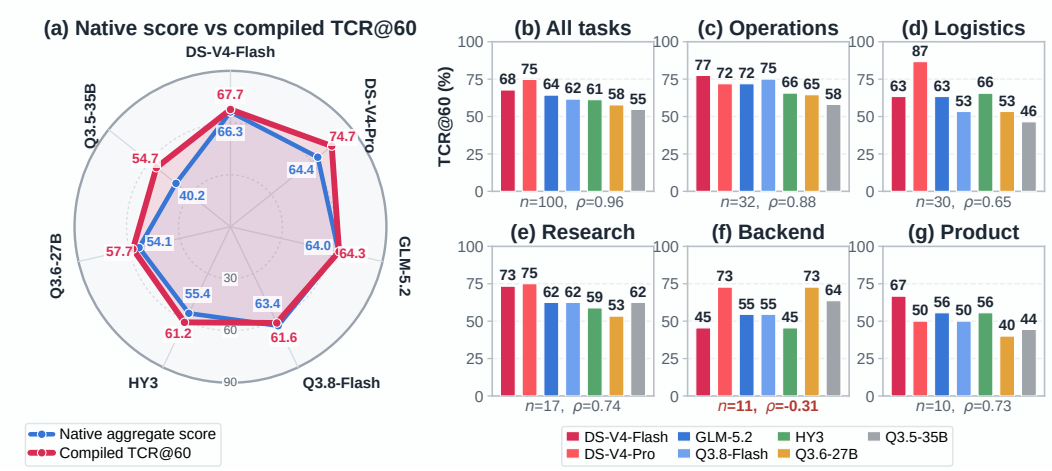


Figure 1: **Native and Compiled aggregate orderings agree on model pairs.** Panel (a) compares aggregate profiles on their respective score scales; panels (b)–(g) report compiled TCR@60, the percentage of valid runs whose normalized evaluator score reaches the completion threshold of 60, overall and by task category. Panel (b) gives original–compiled rank agreement. Correlations in panels (c)–(g) compare each category's compiled ordering with the overall original ordering.

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.

1 INTRODUCTION

Working agents perform extended tasks in software repositories, terminals, and digital workspaces. They use tools, modify persistent state, and combine information across files to produce verifiable outcomes (Jimenez et al., 2024; Merrill et al., 2026; Tang et al., 2026). Comparing new agents and collecting supervision require reusable tasks with an instruction, initial state, interaction protocol, and evaluator. Retaining these components together is costly.

Existing methods usually construct executable tasks from information available before an agent starts. Methods that use repositories recover historical project states or modify working repositories to create tasks and checks (Pan et al., 2024; Jain et al., 2025; Yang et al., 2025; Lin et al., 2026; Wu et al., 2026b), while other systems start from specifications, capability taxonomies, reusable skills, or supplied assets (Gandhi et al., 2026; Zhu et al., 2026; Shen et al., 2026; Shi et al., 2026). Agent executions leave behind records of requests, tool calls, observations, edits, and outcomes. Such records can be reused directly as training data or used to recover environments for renewed interaction (Wu et al., 2026b; Yang et al., 2026). Terminal-Universe reconstructs a reusable environment from a trajectory and then recovers or generates queries in that environment for further interaction and supervision (Wu et al., 2026a). ReTask studies a different construction target: one executable task per record, including an unsolved state and an evaluator. We ask whether the same partial record can support both components so that the resulting task can be used for evaluation and supervision.

One execution record need not determine the original task uniquely. It records one agent's partial execution, may omit required resources, and mixes initial task evidence with results produced while solving it. Several initial states and evaluators can fit the same record. Replaying the recorded execution can reveal the solution, while retaining only observed inputs can leave the task incomplete. Figure 2 contrasts task compilation with construction from source assets and environment reconstruction from trajectories by the reusable object produced and how it is evaluated.

ReTask addresses this question by deriving the unsolved initial state and evaluator from a shared task specification. The actionable request and initial-state evidence support one specification of the goal, required inputs and outputs, resource dependencies, and completion conditions. Execution-produced artifacts may identify content to withhold or validation cases, but cannot introduce requirements and are never shown to the new solver. Compilation thus turns a past execution into new attempts whose outcomes can support comparison and whose verified trajectories can provide supervision.

We first test model comparison and its behavior under extensive resource synthesis and failed source executions on Workspace-Bench (Tang et al., 2026). We then test supervision on SWE-ReBench, whose original repository tests provide an external correctness check on new solutions beyond ReTask's evaluator (Badertdinov et al., 2025). We compile SWE-ReBench tasks from public OpenHands trajectories (Trofimova et al., 2025; Wang et al., 2024) standardized under the Agent Data Protocol (Song et al., 2025). On 100 paired Workspace-Bench tasks, the observed Native and Compiled aggregate rankings agree on 20 of 21 pairwise orders. Task-removal and nearby-threshold checks support this aggregate agreement, and paired comparisons identify differences among some non-source configurations. Compiled tasks remain solvable and distinguish models under extensive resource synthesis and failed recorded executions. On SWE-ReBench, 2,040 patches pass the original tests, representing 52.5% of all 3,889 attempts and 72.7% of 2,807 evaluable patches. On the 1,123 tasks solved in both conditions, training on Compiled trajectories improves two base models in all six evaluated settings.

The main contributions are:

- We formulate *task compilation from an execution record*: the problem of constructing a reusable executable task from one partial recorded execution for comparative evaluation and supervision, without requiring exact recovery of the original task.
- We introduce **ReTask**, which infers one task specification from the available evidence and uses it to construct an unsolved initial state and an evaluator.
- We show aggregate ranking agreement for the evaluated configurations and identify differences between some model pairs. We also show solvability and model discrimination in the evaluated high-synthesis and failed-source subsets, and verify new SWE solutions and their training value using original repository tests and paired SFT comparisons.

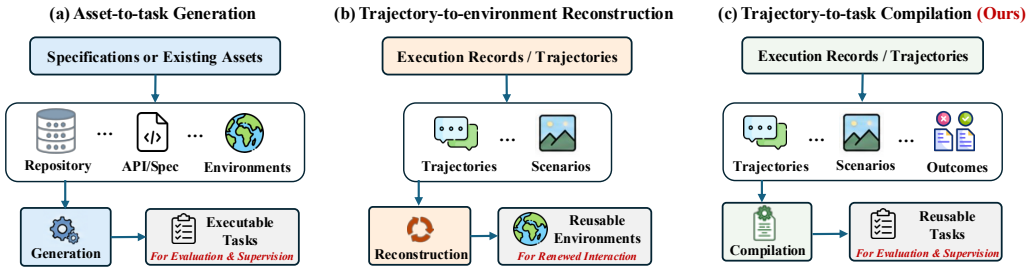


Figure 2: **Ways to construct reusable agent resources.** **(a)** Task construction from source assets produces executable tasks from specifications, repositories, skills, or existing environments. **(b)** Environment reconstruction from execution records recovers reusable environments for renewed interaction. **(c)** Task compilation from an execution record infers one task specification and uses it to construct an unsolved initial state and evaluator for comparative evaluation and supervision.

2 Problem: Constructing a Reusable Task from One Execution

We represent an executable task as $\mathcal{T} = (I, D_0, R, V)$, with instruction I, initial state D_0, interaction protocol R, and evaluator V. A record may omit resources or completion conditions and need not identify the original task uniquely. We instead construct a new task whose initial state and evaluator follow the same inferred specification.

ReTask receives one execution record $x = (\tau, E, O)$. The trajectory $\tau = \{(h^t, a^t, e^t)\}_{t=1}^{L}$ contains the context, action, and observation at each step, E contains retained evidence such as file snapshots and metadata, and O contains the recorded outcome. Without the original initial state or evaluator, ReTask constructs

$$x = (\tau, E, O) \xrightarrow{\text{ReTask}} \widetilde{\mathcal{T}} = (\widetilde{I}, \widetilde{D}_0, \widetilde{R}, \widetilde{V}). \tag{1}$$

Here, *execution record* denotes x and *trajectory* denotes τ. In the experiments, *Native* refers to the original benchmark tasks and *Compiled* to tasks produced by ReTask. The original task package is withheld from the compiler and used only for subsequent comparison, verification, and Native supervision.

The Compiled task must be solvable but initially unsolved. Let $\mathcal{R}_{\text{in}}$ denote resources required at task start and $\mathcal{R}_{\text{out}}$ the outputs that the solver must produce. The initial state must satisfy the requirements of every $r \in \mathcal{R}_{\text{in}}$ and $\text{Paths}(\mathcal{R}_{\text{out}}) \cap \text{Paths}(\widetilde{D}_0) = \varnothing$, while excluding recorded edits, intermediate results, and answers. The evaluator $\widetilde{V}$ judges completion rather than replay of the recorded actions. An unavailable input may be replaced only when its task-relevant properties can be preserved without its exact identity. These requirements apply to successful and failed records.

Comparative evaluation. We compare a fixed panel of agent configurations, each defined by a model, reasoning mode, and harness (Appendix C.2). Let S_k^{N} and S_k^{C} denote the aggregate Native and Compiled scores for configuration k, with corresponding rank vectors $\mathbf{r}^{\text{N}}$ and $\mathbf{r}^{\text{C}}$. Because the two collections use different evaluators and score scales, we compare model rankings through rank correlation and pairwise order agreement, and report whether individual Compiled tasks separate configurations.

Supervision. For each Compiled task, a teacher produces a trajectory τ_j^{C} and outcome y_j^{C}. An external checker $W_j(y_j^{\text{C}}) \in \{0, 1\}$ determines whether the outcome also solves the original task when it is evaluable. We report verified success over all attempts and conditional pass rate over evaluable outcomes. For training, corpora $\mathcal{D}_{\text{C}}$ and $\mathcal{D}_{\text{N}}$ contain trajectories from tasks solved in both conditions, with matched teacher, protocol, correctness test, and exposure.

3 Method

An execution record mixes evidence about the task with results produced by one solver. The central question is which observations may support the task specification. ReTask resolves this question

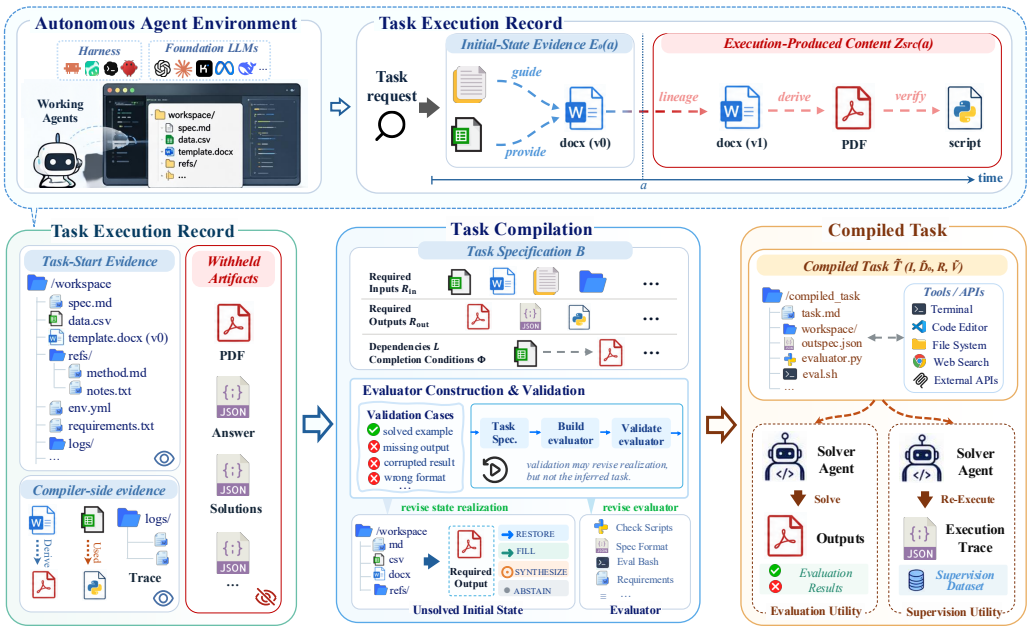


Figure 3: **Overview of ReTask. Top:** an execution record is partitioned at the actionable-request boundary a into initial-state evidence $\mathcal{E}_0(a)$ and execution-produced content $\mathcal{Z}_{\mathrm{src}}(a)$. **Bottom:** the compiler infers a task specification $\mathcal{B}$, uses it to realize an unsolved initial state and evaluator, and validates both before packaging the Compiled task. Recorded artifacts and solutions are withheld from the new solver; they may support validation but cannot revise the inferred task. The resulting task supports independent solving for evaluation and re-execution for supervision.

before constructing the task: the actionable request and evidence about the initial state support one task specification, which determines both what a new solver receives and how its outcome is judged. Artifacts and outcomes produced during the recorded execution may identify content to withhold or provide validation cases for the evaluator, but they cannot introduce a task requirement. Figure 3 summarizes this compilation process and its two downstream uses.

Given an execution record x, ReTask selects an actionable request q_a and partitions the evidence at boundary a:

$$\big(\mathcal{E}_0(a), \mathcal{Z}_{\mathrm{src}}(a)\big) = \mathrm{Partition}(x, a). \tag{2}$$

Eq. (2) separates evidence about the initial state from results produced during the recorded execution. Specifically, $\mathcal{E}_0(a)$ describes the state from which the new task should start, and $\mathcal{Z}_{\mathrm{src}}(a)$ contains state changes, artifacts, and outcomes produced during the recorded execution. From the latter, ReTask identifies recorded artifacts $\mathcal{X}_{\mathrm{src}}(a)$ that must be withheld. The overall construction is $x \to (q_a, \mathcal{E}_0, \mathcal{X}_{\mathrm{src}}) \to \mathcal{B} \to (\widetilde{I}, \widetilde{D}_0, \widetilde{R}, \widetilde{V})$. The initial state provided to a new solver and the completion conditions implemented by the evaluator are therefore derived from the same task specification, rather than from replaying the trajectory.

3.1 Inferring the Task Specification from Available Evidence

The first stage determines what task is supported by the record. Temporal position alone is insufficient: an observation made after the request but before the first modification can still describe the initial version of a resource. ReTask therefore groups observations by resource and version. Evidence describing a resource before modification may enter $\mathcal{E}_0(a)$; created files, replacement content, observations of modified versions, verification outputs, and final responses enter $\mathcal{Z}_{\mathrm{src}}(a)$. Appendix B.3 gives the version rules.

Artifacts in $\mathcal{X}_{\mathrm{src}}(a)$ may identify content that must be absent from the new initial state or may be matched to an output requirement already supported by the request, prior context, task rules, or input

evidence. They cannot introduce a new goal, output requirement, or completion condition. Other recorded results are reserved for evaluator validation after the task specification has been fixed.

ReTask infers

$$\mathcal{B} = \mathrm{InferTask}(q_a, \mathcal{E}_0(a); \mathcal{X}_{\mathrm{src}}(a)) = \big(g, \mathcal{R}_{\mathrm{in}}, \mathcal{R}_{\mathrm{out}}, \mathcal{L}, \Phi\big). \tag{3}$$

The semicolon in Eq. (3) indicates that the inputs are used differently. The request q_a and evidence about the initial state $\mathcal{E}_0(a)$ may support fields of $\mathcal{B}$, whereas $\mathcal{X}_{\mathrm{src}}(a)$ may only match an output requirement that is already supported or identify content to withhold. The resulting specification determines the requirements for both the constructed initial state and evaluator. Here, g defines the requested outcome and visible constraints, $\mathcal{R}_{\mathrm{in}}$ and $\mathcal{R}_{\mathrm{out}}$ define required inputs and outputs, $\mathcal{L}$ records dependencies among required inputs, and $\Phi = \{\phi_j\}_{j=1}^{J}$ contains observable completion conditions. Each ϕ_j specifies what a valid outcome must satisfy, not which recorded action produced it. Because $\mathcal{B}$ depends on task evidence rather than the recorded solver's success, a failed execution can still provide enough evidence to infer a task specification.

3.2 Constructing an unsolved initial state

The second stage determines how to construct each required input while preserving its required role, representation, and content. For $r \in \mathcal{R}_{\mathrm{in}}$, let $\psi_r = (u_r, f_r, q_r)$ describe its task role, required representation, and required content. Given the strongest available evidence $e_r^\star$, ReTask chooses $\alpha_r \in \{\textsc{Restore}, \textsc{Fill}, \textsc{Synthesize}, \textsc{Abstain}\}$ according to the support for ψ_r.

It selects Restore when $e_r^\star \models \psi_r$. When the evidence is partial but the missing content can be inferred, it uses Fill, preserving the observed portion and constructing only what is missing. When no usable original content is available, it uses Synthesize only if another resource can satisfy ψ_r without preserving the identity of r. Otherwise, it selects Abstain. In particular, ReTask abstains when correctness depends on unavailable original content, live state, credentials, or another unsupported dependency. Appendix B.4 gives the full conditions.

When every required input can be realized, the compiler assembles $\widetilde{D}_0 = \bigoplus_{r \in \mathcal{R}_{\mathrm{in}}} \alpha_r(r \mid \mathcal{B})$, renders $\widetilde{I} = \mathrm{Render}(g)$, and binds $\widetilde{R} = \mathrm{Bind}(g, \{\psi_r\})$. The resulting state must satisfy $\widetilde{D}_0[r] \models \psi_r$ for every $r \in \mathcal{R}_{\mathrm{in}}$, contain the resources referenced by $\mathcal{L}$, and satisfy $\mathrm{Paths}(\mathcal{R}_{\mathrm{out}}) \cap \mathrm{Paths}(\widetilde{D}_0) = \varnothing$. Recorded outputs, edits, and identified answer fragments are also excluded. The state therefore provides the information needed to attempt the task while keeping the task unsolved.

3.3 Implementing and validating fixed completion conditions

The completion conditions Φ are fixed before evaluator construction. The final stage implements these conditions for outcomes produced from the constructed state. For a new outcome $y = (m, D_f)$, ReTask constructs $\widetilde{V} = \mathrm{BuildEvaluator}(\Phi, \widetilde{D}_0, \widetilde{R})$. For condition ϕ_j, let $v_j(y) \in [0, 1]$ be its check and w_j its weight; the evaluator returns $s(y) = \sum_j w_j v_j(y)$ with $\sum_j w_j = 100$, and a valid outcome is complete when $s(y) \geq 60$. Evaluation depends on the submitted response and final state, not on reproducing the recorded action sequence.

ReTask validates $\widetilde{V}$ using $\mathcal{Q} = \{(y_\ell, \omega_\ell)\}_{\ell=1}^{K}$. A scoring example y_{smk} must reach the substantive checks; the unchanged state $y_{\mathrm{noop}} = (\varnothing, \widetilde{D}_0)$ must remain below the completion threshold; and task-specific negative cases y_{neg} must lose credit for the violated completion condition. When the recorded outcome maps directly to the Compiled task, it may also be evaluated as y_{src}. Because Φ is already fixed, this case can expose an inconsistency but cannot change the evaluator criteria.

A Compiled task is valid only when its required inputs and dependency endpoints are present, target outputs and recorded results are absent, and every validation case produces the expected evaluator behavior. Failed validation may revise $\widetilde{D}_0$ or $\widetilde{V}$, but the inferred task remains unchanged:

$$(\mathcal{B}, \widetilde{I}, \widetilde{R})^{(i+1)} = (\mathcal{B}, \widetilde{I}, \widetilde{R})^{(i)}. \tag{4}$$

Eq. (4) allows validation to revise the constructed state or evaluator, but not to redefine the task from the recorded outcome. If the available evidence does not support a valid construction, ReTask abstains. Algorithm 1 and Appendix B provide the full procedure.

4 EXPERIMENTS

Construction defines what a new solver receives and how its outcome is scored; the experiments test whether new outcomes support model comparison on Workspace-Bench Lite and externally verified trajectories support supervision on SWE-ReBench (Tang et al., 2026; Badertdinov et al., 2025; Trofimova et al., 2025; Wang et al., 2024; Song et al., 2025).

4.1 EXPERIMENTAL SETUP

Comparative evaluation. We compile one task from each of the 100 public Workspace-Bench Lite execution records. Seven fixed agent configurations attempt every Compiled task. We compare their TCR@60 ordering with the official Native ordering using all 21 model pairs, Spearman ρ, and Kendall τ; rankings avoid comparing raw scores from different evaluators and scales.

Synthesis and failed executions. We group tasks by the fraction of used resources synthesized and separately analyze the 23 tasks compiled from failed records. For each group, we report solvability and mixed outcomes over the six non-source models; 49 of 97 eligible tasks synthesize more than half of their used resources.

Supervision. For SWE-ReBench, we use public OpenHands trajectories standardized under the Agent Data Protocol as compilation evidence (Trofimova et al., 2025; Wang et al., 2024; Song et al., 2025). GLM-5.2 independently solves the Native and Compiled versions of the same original software issues under the same agent protocol; original repository tests check both sets of patches. Training uses the 1,123 tasks solved in both conditions, with one trajectory per condition and matched teacher, protocol, correctness criterion, and exposure. The conditions differ in instruction and initial state, with a median Compiled-to-Native instruction-length ratio of 3.26. We fine-tune Qwen3.5-35B-A3B and Qwen3-30B-A3B and evaluate on SWE-bench Verified, Pro, and Multilingual (Jimenez et al., 2024). We report $\Delta_{\mathrm{C}} = A_{\mathrm{C}} - A_{\mathrm{B}}$ and $\gamma = \Delta_{\mathrm{C}}/(A_{\mathrm{N}} - A_{\mathrm{B}})$, where A_{B}, A_{N}, and A_{C} denote the Base, Native SFT, and ReTask SFT accuracies. Appendices C.1–C.3 provide the complete protocol.

4.2 COMPARATIVE EVALUATION WITH COMPILED TASKS

The Compiled tasks retain 20 of 21 pairwise model orderings. The two leading models exchange positions while the other five retain their ranks, giving 20 of 21 matching pairwise orders, Spearman $\rho = 0.964$, and Kendall $\tau = 0.905$ (exact two-sided permutation $p = 0.003$; Table 1).

Agreement under model and task subsets. Removing the model that generated the records gives the same ordering for the remaining six models ($\rho = 1.000$). On the 90 tasks with scores for all models, $\rho = 0.927$ with a task-bootstrap 95% interval of $[0.393, 0.955]$. Across 10,000 bootstrap samples, 99.81% yield $\rho > 0$ and 94.95% yield $\rho \geq 0.5$; 53 of 100 tasks produce both passing and failing outcomes. Leaving out any single task gives $\rho \in [0.929, 0.964]$. On the 90 complete tasks, nearby thresholds of 55 and 65 give $\rho = 0.955$ and 0.929, respectively (Appendix D.1).

Paired outcomes identify differences among non-source models. On the same 90 tasks, DeepSeek-V4-Pro exceeds Hy3, Qwen3.6-27B, and Qwen3.5-35B-A3B by 14.44, 18.89, and 22.22 percentage points. All three differences pass two-sided exact McNemar tests with Holm correction over all 21 pairs ($p_{\mathrm{Holm}} < 0.05$), and remain significant when each pair uses all its jointly scored tasks. The Pro–Qwen3.5 comparison also passes the correction at thresholds 55 and 65 under both task-set definitions. Close comparisons remain unresolved: Qwen3.8-Flash-Next and Hy3 each pass 57 of the 90 tasks, with a paired 95% difference interval of $[-8.89, 8.89]$ points. Tables D.1–D.2 report all pairs and sensitivity results.

Together, the aggregate ordering closely agrees with Native, while paired outcomes resolve some, but not all, model differences; Figure 1 and Appendices D.2–D.3 report the breakdowns by task, category, difficulty, and workspace size.

Table 1: **Native and Compiled aggregate orderings agree on 20 of 21 model pairs.** The Native columns report the official aggregate score and rank; Compiled entries report TCR@60 (%) and rank overall and by difficulty. TCR@60 is the percentage of valid runs with normalized evaluator score ≥ 60; the pooled row aggregates valid runs. Overall pairwise agreement uses the Native order; difficulty columns use the overall Compiled order. Leading values are bold, ties share a rank, and tied pairs are excluded from agreement.

Model	Native benchmark		Compiled TCR@60 / rank			
	Score	r	Overall ($N = 100$)	Easy ($N = 14$)	Medium ($N = 54$)	Hard ($N = 32$)
DeepSeek-V4-Flash	**66.27**	1	67.7 / 2	64.3 / 4	58.8 / 2	83.9 / 2
DeepSeek-V4-Pro	64.43	2	**74.7 / 1**	**85.7 / 1**	**64.2 / 1**	**87.5 / 1**
GLM-5.2	63.95	3	64.3 / 3	69.2 / 3	57.4 / 3	74.2 / 4
Qwen3.8-Flash-Next	63.41	4	61.6 / 4	64.3 / 4	56.6 / 4	68.8 / 5
Hy3	55.41	5	61.2 / 5	64.3 / 4	51.9 / 5	76.7 / 3
Qwen3.6-27B	54.11	6	57.7 / 6	71.4 / 2	47.1 / 6	68.8 / 5
Qwen3.5-35B-A3B	40.22	7	54.7 / 7	64.3 / 4	45.1 / 7	66.7 / 7
Pooled TCR@60		–	63.2	69.1	54.5	75.2
Tasks with both outcomes		–	53/100	8/14	30/54	15/32
Pairwise order agreement		–	**20/21**	10/15	21/21	18/20

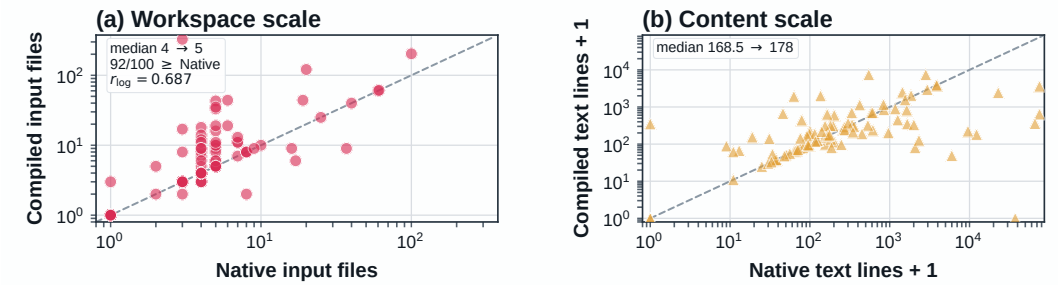


Figure 4: **Native and Compiled tasks have comparable workspace sizes. (a)** Input-file counts have medians of four and five; 92 of 100 Compiled workspaces contain at least as many inputs as their Native counterpart. **(b)** Extractable text has medians of 168.5 and 178 lines. Dashed lines mark equality.

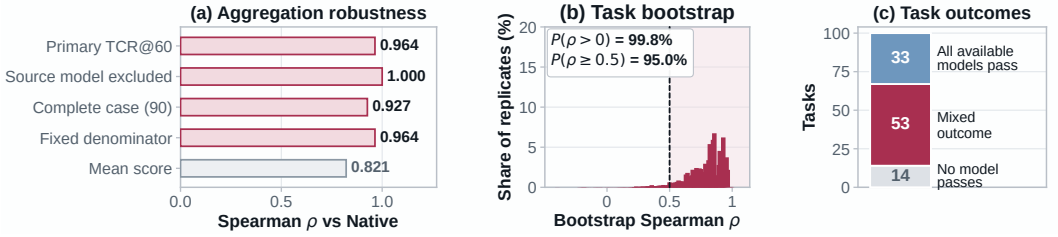


Figure 5: **Ranking agreement across model subsets, task subsets, and score aggregation. (a)** Agreement with the Native ordering after excluding the model that generated the records, restricting the comparison to tasks with scores for all models, counting missing scores as failures, or aggregating mean scores. **(b)** Rank correlations across 10,000 bootstrap samples of the 90 complete tasks. **(c)** Task outcomes across the panel of seven models: 53 tasks separate at least two models, 14 are passed by no model, and 33 are passed by every model with an available score. The appendix provides the complete score matrix.

4.3 Evaluation with synthesized resources and failed executions

We examine whether Compiled tasks remain solvable and differentiate models when most used resources are synthesized or the recorded execution was unsuccessful.

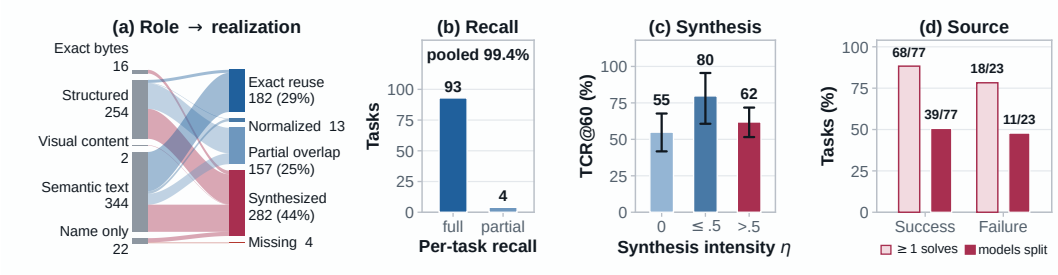


Figure 6: **Solvability and model differentiation in the majority-synthesis and failed-source subsets. (a)** Construction types for 638 resources used in the recorded executions. **(b)** Fraction of these resources available at their recorded paths in each task, with 99.4% available overall. **(c)** TCR@60 over the six models that did not generate the records, grouped by the fraction of synthesized resources. **(d)** Numbers of tasks solved by at least one model and tasks solved by some but not all models, grouped by the outcome of the recorded execution.

Most used resources require completion or synthesis. Among 638 resources used in the recorded executions, 195 can be restored from recorded content, 157 are filled from partial observations, and 282 are synthesized replacements (Figure 6a). This makes 634 resources available at their recorded paths; the remaining four locate directories. Completion or synthesis accounts for 439 of the 638 used resources.

The majority-synthesis subset contains solvable tasks that distinguish models. In the 49 tasks where more than half of the used resources are synthesized, 261 of 320 used resources are newly constructed. The six models that did not generate the records reach 61.9% pooled TCR@60; at least one model solves 44 of the 49 tasks, and 27 are solved by some but not all models (Figure 6c). This subset retains 12 of 15 pairwise orders relative to the full Compiled ordering of the same six models ($\rho = 0.771$).

The failed-source subset contains solvable tasks that distinguish models. Among the 23 tasks compiled from failed executions, at least one of the six other models solves 18, and 11 are solved by some but not all of these models (Figure 6d). Their pooled TCR@60 is 56.9%. The subset also retains 13 of 14 non-tied pairwise orders relative to the full Compiled six-model ordering ($\rho = 0.928$).

Both analyzed subsets yield successful solutions and cross-model outcome differences. Compiled input file counts are at least Native counts in 92 of 100 tasks (Figure 4), and the largest Compiled size quartile, with 10–327 input files, has $\rho = 0.883$ against the full Compiled ordering.

Appendices D–F provide the complete score matrix, workspace statistics, resource construction, and examples of individual tasks.

4.4 Collecting supervision from new attempts

On SWE-ReBench, we test whether re-solving Compiled tasks produces externally verified solutions and trajectories with downstream training value. Original repository tests identify correct patches and the trajectories admitted to the SFT corpus; Figure 7 reports test outcomes and modified files.

Solutions from Compiled tasks pass the original repository tests. Across 2,927 original tasks, 3,889 Compiled attempts yield 2,814 submitted nonempty patches, 2,807 evaluable patches, and 2,040 test successes: 52.5% of all attempts and 72.7% of evaluable patches. These successes cover 1,614 original tasks, of which 1,123 are also solved in the Native condition and form the paired SFT corpus.

Compiled workspaces cover the source files used by Native repairs. Within this verified paired corpus, we compare repair locations and workspace coverage. Compiled and Native solutions modify identical source-file sets in 93.9% of cases, with mean Jaccard similarity 0.967. Compiled workspaces contain a median of four files and cover every source file modified by the Native solution

Table 2: **SFT on Compiled trajectories improves both base models on all three benchmarks.** Scores are official resolved rates. The last two columns report improvement over Base and the ratio between the gains from ReTask SFT and Native SFT.

Base model	Benchmark	Base	Native	ReTask		
			SFT	SFT	Gain (pp)	Retention (%)
Qwen3.5-35B-A3B	SWE-Verified	67.00	77.80	74.00	+7.00	64.8
	SWE-Pro	36.94	50.48	45.96	+9.02	66.6
	SWE-Multilingual	48.56	69.22	65.56	+17.00	82.3
Qwen3-30B-A3B	SWE-Verified	20.00	41.23	43.84	+23.84	112.3
	SWE-Pro	5.52	16.83	16.96	+11.44	101.1
	SWE-Multilingual	16.00	24.33	26.89	+10.89	130.7

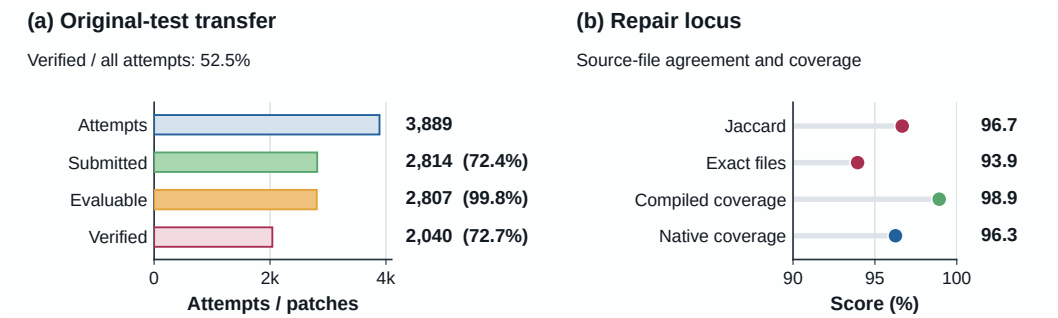


Figure 7: **Solutions produced on Compiled tasks pass the original tests and show high source-file agreement with Native solutions. (a)** Of 3,889 attempts, 2,814 submit nonempty patches, 2,807 are evaluable, and 2,040 pass: 52.5% of all attempts and 72.7% of evaluable patches. **(b)** Among 1,123 tasks solved in both conditions, source-file sets have mean Jaccard similarity 0.967 over 1,119 defined pairs and are identical in 93.9% of task pairs; Compiled workspaces cover files modified by Compiled and Native solutions in 98.9% and 96.3% of tasks.

in 96.3% of tasks. Across four groups defined by Compiled text size, Native repair-file coverage remains above 95%, and 92.5–96.4% of paired solutions modify identical source-file sets (Table E.1).

Trajectories from Compiled tasks improve both downstream models. Table 2 tests the downstream value of the verified paired corpus by comparing Native and ReTask trajectories from the same software issues. On Verified, Pro, and Multilingual, ReTask SFT improves Qwen3.5-35B-A3B by 7.00, 9.02, and 17.00 percentage points, corresponding to 64.8%, 66.6%, and 82.3% of the gains from Native SFT. It improves Qwen3-30B-A3B by 23.84, 11.44, and 10.89 points, corresponding to 112.3%, 101.1%, and 130.7% of the Native SFT gains. All six settings improve over Base. The comparison matches the software issue, teacher, correctness test, agent protocol, and training exposure.

Successful Compiled trajectories average 4.35 fewer assistant steps (paired 95% CI $[-5.08, -3.62]$), so the gains are observed on shorter trajectories under matched token exposure. Appendices E and E.2 provide full analyses.

5 Conclusion

ReTask turns one execution record into an unsolved task for new solvers. A shared task specification connects partial evidence about the initial state to both the resources a new solver receives and the conditions used to judge its outcome. On Workspace-Bench, the Native and Compiled aggregate orderings agree on 20 of 21 model pairs; the evaluated high-synthesis and failed-source subsets contain solvable tasks with cross-model outcome differences. On SWE-ReBench, 2,040 patches pass original tests, and verified Compiled trajectories from 1,123 paired tasks improve two models in all six settings. These results show that useful task reuse can be achieved without exact reconstruction of the original task in the studied settings. Beyond retaining a past execution, records can support new attempts whose outcomes enable comparison and whose verified trajectories provide supervision.

AI USE STATEMENT

Generative AI tools were used to assist with language editing and result analysis. The authors reviewed and verified all AI-assisted outputs and take full responsibility for the content and conclusions of the final manuscript.

REFERENCES

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. In *Advances in Neural Information Processing Systems*, pp. 26420–26466, 2025. doi: 10.52202/085713-0788.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026a. URL https://arxiv.org/abs/2606.19348.

DeepSeek-AI. DeepSeek-V4-Flash update, July 2026b. URL https://api-docs.deepseek.com/updates/#date-2026-07-31.

DeepSeek-AI. DeepSeek-V4-Pro update, August 2026c. URL https://api-docs.deepseek.com/updates/#date-2026-08-13.

Kanishk Gandhi, Shivam Garg, Noah D. Goodman, and Dimitris Papailiopoulos. Endless terminals: Scaling RL environments for terminal agents, 2026.

GLM-5 Team. GLM-5: From vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents, 2025. URL https://arxiv.org/abs/2504.07164.

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*, 2024.

Yusong Lin, Haiyang Wang, Shuzhe Wu, Lue Fan, Feiyang Pan, Sanyuan Zhao, and Dandan Tu. CLI-Gym: Scalable CLI task generation via agentic environment inversion, 2026.

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, Rithesh Murthy, Liangwei Yang, Silvio Savarese, Juan Carlos Niebles, Huan Wang, Shelby Heinecke, and Caiming Xiong. APIGen: Automated pipeline for generating verifiable and diverse function-calling datasets. In *Advances in Neural Information Processing Systems*, pp. 54463–54482, 2024. doi: 10.52202/079017-1725.

Fanzhe Meng, Guoxin Chen, Jiale Zhao, Shuang Sun, Zhiyu Lin, Wayne Xin Zhao, Ruihua Song, Ji-Rong Wen, and Kai Jia. CalibForge: Adversarial solver calibration for scaling learnable terminal tasks, 2026. URL https://arxiv.org/abs/2608.06352.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, 2026.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym, 2024. URL https://arxiv.org/abs/2412.21139.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026a. URL https://qwen.ai/blog?id=qwen3.5. Official model card: https://huggingface.co/Qwen/Qwen3.5-35B-A3B.

Qwen Team. Qwen3.6-27B: Flagship-level coding in a 27B dense model, April 2026b. URL https://qwen.ai/blog?id=qwen3.6-27b. Official model card: https://huggingface.co/Qwen/Qwen3.6-27B.

Qwen Team. Qwen3.8-Flash-Next: A new architecture, towards ultimate cost-efficiency, August 2026c. URL https://qwen.ai/blog?id=qwen3.8-flash-next. Official model card: https://huggingface.co/Qwen/Qwen3.8-Flash-Next.

Qijia Shen, Zhiqi Huang, Vamsidhar Kamanuru, Aznaur Aliev, Jay Rainton, Ahmed Awelkair, Zhichen Zeng, Jiajun Li, Shi Dong, Yueming Yuan, Boyuan Ma, Qizheng Zhang, Jiwei Fu, Yuzhen Mao, Wendong Fan, Ping Nie, Philip Torr, Bernard Ghanem, Changran Hu, Jonathan Lingjie Li, Urmish Thakker, and Guohao Li. SETA: Scaling environments for terminal agents, 2026. URL https://arxiv.org/abs/2607.10891.

Kou Shi, Zun Wang, Qisheng Su, Shiting Huang, Ziao Zhang, Zhen Fang, Qingnan Ren, Jin Liu, Yu Zeng, Yiming Zhao, Lin Chen, Zehui Chen, and Feng Zhao. FACET: Preserving source intent and executable state in terminal task synthesis, 2026.

Yueqi Song, Ketan Ramaneti, Zaid Sheikh, Ziru Chen, Boyu Gou, Tianbao Xie, Yiheng Xu, Danyang Zhang, Apurva Gandhi, Fan Yang, Joseph Liu, Tianyue Ou, Zhihao Yuan, Frank Xu, Shuyan Zhou, Xingyao Wang, Xiang Yue, Tao Yu, Huan Sun, Yu Su, and Graham Neubig. Agent data protocol: Unifying datasets for diverse, effective fine-tuning of LLM agents, 2025. URL https://arxiv.org/abs/2510.24702.

Hongjin Su, Ruoxi Sun, Jinsung Yoon, Pengcheng Yin, Tao Yu, and Sercan O. Arik. Learn-by-interact: A data-centric framework for self-adaptive agents in realistic environments. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2501.10893.

Qiushi Sun, Kanzhi Cheng, Zichen Ding, Chuanyang Jin, Yian Wang, Fangzhi Xu, Zhenyu Wu, Chengyou Jia, Liheng Chen, Zhoumianze Liu, Ben Kao, Guohao Li, Junxian He, Yu Qiao, and Zhiyong Wu. OS-Genesis: Automating GUI agent trajectory construction via reverse task synthesis. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, 2025. URL https://arxiv.org/abs/2412.19723.

Zirui Tang, Xuanhe Zhou, Yumou Liu, Linchun Li, Weizheng Wang, Hongzhang Huang, Jun Zhou, Jiachen Song, Shaoli Yu, Fan Wu, et al. Workspace-Bench 1.0: Benchmarking AI agents on workspace tasks with large-scale file dependencies, 2026.

Tencent Hy Team. Tencent Hy3: Model card, 2026. URL https://huggingface.co/tencent/Hy3.

Maria Trofimova, Anton Shevtsov, Ibragim Badertdinov, Konstantin Pyaev, Simon Karasik, and Alexander Golubev. OpenHands trajectories with Qwen3-Coder-480B-A35B-Instruct. Nebius blog, 2025. URL https://nebius.com/blog/posts/openhands-trajectories-with-qwen3-coder-480b.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents, 2024.

Ziyi Wang, Yuxuan Lu, Yimeng Zhang, Pei Chen, Ziwei Dong, Jing Huang, Jiri Gesi, Xianfeng Tang, Chen Luo, Qun Liu, Yisi Sang, Hanqing Lu, Manling Li, Jin Lai, and Dakuo Wang. Trajectory2Task: Training robust tool-calling agents with synthesized yet verifiable data for complex user intents. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics*, pp. 44021–44044, 2026. doi: 10.18653/v1/2026.acl-long.2037.

Jie Wu, Zhenru Zhang, Beichen Zhang, Xuwu Wang, Yuhui Su, Mouxiang Chen, Peng Wang, Zhihai Wang, Que Shen, Hao Zhou, An Yang, Fei Huang, Yujiu Yang, and Dayiheng Liu. Terminal-Universe: Turning agent trajectories into scalable terminal environments, 2026a. URL https://arxiv.org/abs/2609.04148.

Siwei Wu, Yizhi Li, Yuyang Song, Wei Zhang, Yang Wang, Riza Batista-Navarro, Xian Yang, Mingjie Tang, Bryan Dai, Jian Yang, and Chenghua Lin. Large-scale terminal agentic trajectory generation from dockerized environments, 2026b.

Minrui Xu, Zilin Wang, Mengyi Deng, Zhiwei Li, Zhicheng Yang, Xiao Zhu, Yinhong Liu, Boyu Zhu, Baiyu Huang, Chao Chen, Heyuan Deng, Fei Mi, Lifeng Shang, Xingshan Zeng, and Zhijiang Guo. EnvFactory: Scaling tool-use agents via executable environments synthesis and robust RL, 2026. URL https://arxiv.org/abs/2605.18703.

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

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, 2025.

Sidi Yang, Chaofan Tao, Jierun Chen, Tiezheng Yu, Ruoyu Wang, Yuxin Jiang, Yiming Du, Wendong Xu, Jing Xiong, Taiqiang Wu, Lifeng Shang, Xiaohui Li, Ngai Wong, and Haoli Bai. What makes interaction trajectories effective for training terminal agents?, 2026.

Z.ai. GLM-5.2: Built for long-horizon tasks, June 2026. URL https://z.ai/blog/glm-5.2.

Wenhao Zeng, Yuling Shi, Xiaodong Gu, Chao Hu, Chaofan Wang, Yuhao Cui, Hongting Zhou, Mengnan Qi, Jianqiao Wangni, Zhaojian Yu, Shuzheng Gao, Kai Cai, and Shilin He. Dockerless: Environment-free program verifier for coding agents, 2026a. URL https://arxiv.org/abs/2606.28436.

Xingshan Zeng, Zishan Xu, Boju Zhang, Yuzhou Wu, Lingzhi Wang, Jianghao Lin, Liangyou Li, Yasheng Wang, Lifeng Shang, Xin Jiang, Weinan Zhang, Yong Yu, Qun Liu, and Weiwen Liu. What makes good agentic data? an ACE lens on data generation for LLM agents, 2026b. URL https://arxiv.org/abs/2608.27260.

Kaijie Zhu, Yuzhou Nie, Yijiang Li, Yiming Huang, Jialian Wu, Jiang Liu, Ximeng Sun, Zhenfei Yin, Lun Wang, Zicheng Liu, Emad Barsoum, William Yang Wang, and Wenbo Guo. TermiGen: High-fidelity environment and robust trajectory synthesis for terminal agents, 2026. URL https://arxiv.org/abs/2602.07274.

Appendix

A Related Work 14

B Task Compilation Details 15

B.1 Notation 15

B.2 Information Available to the Compiler and Solver 15

B.3 Inferring the Task Specification from Available Evidence 16

B.4 Constructing an Unsolved Initial State 16

B.5 Implementing and Validating Fixed Completion Conditions 17

B.6 Compilation Procedure 18

C Experimental Protocols 19

C.1 Runtime and Optimization Configuration 19

C.2 Datasets and Their Roles 19

C.3 Metrics 21

D Comparative Evaluation: Extended Results 22

D.1 Agreement between Model Rankings 22

D.2 Tasks That Distinguish Models 23

D.3 Task-Group Summary 23

D.4 Evaluation with Synthesized Resources and Failed Executions 23

E Supervision: Verified Solutions and Paired Corpus 26

E.1 Repair-File Coverage 26

E.2 Paired Solution and Trajectory Summary 26

F Compilation Case Studies 27

G Reference Prompt Templates 31

A Related work

Figure 2 and Table A.1 distinguish three routes for constructing reusable agent resources by their starting evidence, direct output, and criterion for successful reuse. ReTask addresses the setting in which one execution record is available but the original task package need not be. It uses a shared task specification to construct an unsolved initial state and evaluator, then tests whether new solver outcomes support model comparison and externally verified trajectories provide useful supervision.

Table A.1: **Representative routes for constructing reusable agent resources.** The routes may overlap in capability; they differ in the evidence they start from, the reusable object they construct directly, and how that object is evaluated.

Route	Starting evidence	Reusable object	Evaluation
Task construction from source assets	Specifications, repositories, skills, or existing environments	Executable task	Task validity, executability, or training value
Environment reconstruction from execution records	Execution records	Environment that supports new interactions	Completeness and value for new interaction or data collection
Task compilation from execution records	**One execution record**	**Executable task with an unsolved initial state and evaluator**	**Model comparison, correctness under checks from the original tasks, and training value**

Recent work on data generation for agents emphasizes consistency among environments, tasks, interactions, and verification signals (Zeng et al., 2026b). The categories below distinguish the evidence from which these objects are constructed and the reusable object produced directly.

Constructing tasks from source assets. A large body of work constructs executable tasks from resources that are available before agent execution. SWE-Gym, R2E-Gym, and SWE-smith derive software engineering tasks and checks from repositories (Pan et al., 2024; Jain et al., 2025; Yang et al., 2025), while SWE-ReBench continuously extracts executable software engineering tasks from public repositories (Badertdinov et al., 2025). CLI-Gym constructs tasks from working environments, while TerminalTraj prepares Dockerized repositories before generating aligned tasks, validation code, and trajectories (Lin et al., 2026; Wu et al., 2026b). Endless Terminals, TermiGen, and SETA scale task or environment construction from specifications or generated seeds (Gandhi et al., 2026; Zhu et al., 2026; Shen et al., 2026), and FACET starts from reusable skills and grounds the resulting task artifacts in an executable state (Shi et al., 2026). Related data generation systems construct verifiable tasks or interactions from available interfaces and environments. APIGen generates verified function calling data from executable APIs, Learn-by-Interact constructs instructions from interactions generated using environment documentation, OS-Genesis derives GUI tasks after environment exploration, and EnvFactory synthesizes executable tool environments and trajectories from authentic resources (Liu et al., 2024; Su et al., 2025; Sun et al., 2025; Xu et al., 2026). These methods assume access to repositories, specifications, environments, skills, or other source assets from which a task can be deliberately constructed. ReTask instead starts from one execution record after the original task package may no longer be available.

Reusing execution records. Recorded trajectories can be used directly as supervision (Yang et al., 2026). Trajectory2Task first generates valid tool use trajectories in available environments and then converts them into user tasks with controlled intent adaptations (Wang et al., 2026). Other work goes further by recovering resources for later use. Terminal-Universe restores state before recorded edits, fills missing context, reconstructs intent, and supports new interaction and supervision from recovered environments (Wu et al., 2026a). Its Intent Recovery variant can also reconstruct requests and solve them again for supervision. ReTask instead studies whether one record can define an executable task that includes its own initial state and evaluator. It derives the unsolved initial state and completion conditions from the same task specification, then evaluates aggregate model comparisons and whether newly collected, externally verified trajectories provide downstream training value.

Constructing evaluators. Executable agent tasks require reliable checks of solver outcomes. SWE-Gym and R2E-Gym construct verifiers together with software environments (Pan et al., 2024;

Jain et al., 2025); Dockerless studies program verification without reproducing the full execution environment (Zeng et al., 2026a); and CalibForge uses solver outcomes for verifier calibration during task generation (Meng et al., 2026). ReTask instead fixes the completion conditions before constructing the evaluator and derives both the initial state provided to the new solver and the evaluator from the same task specification. The evaluator is validated with a scoring example, the unchanged initial state, negative cases designed for the task, and the recorded outcome when it maps to the Compiled task. These validation cases test completion conditions already established by the specification; recorded results do not define additional criteria.

B TASK COMPILATION DETAILS

Task compilation connects execution evidence to an unsolved initial state and evaluator through a shared task specification. The details below separate evidence about the initial state from content produced during the recorded execution, and distinguish the compiler interface from the interface presented to a new solver. Construction follows the same three stages as Section 3: infer a task specification from the available evidence, construct an unsolved initial state that satisfies its resource requirements, and implement an evaluator from its fixed completion conditions.

B.1 NOTATION

Table B.1 summarizes the notation used throughout the compiler. Tildes denote components of the Compiled task.

Table B.1: **Core notation for ReTask.**

Symbol	Definition
$\mathcal{T} = (I, D_0, R, V)$	Executable task consisting of an instruction, initial state, interaction protocol, and evaluator.
$x = (\tau, E, O)$	Execution record provided to ReTask: trajectory τ, additional retained evidence E, and recorded outcome O.
q_a	Selected actionable request together with the prior context needed to interpret it.
$\mathcal{E}_0(a)$	Evidence about the state from which the Compiled task should start.
$\mathcal{Z}_{\mathrm{src}}(a)$	Content, state changes, and outcomes produced during the recorded execution.
$\mathcal{X}_{\mathrm{src}}(a)$	Recorded artifacts that must be withheld from the new solver or matched to an output requirement already supported by task evidence.
$\mathcal{B} = (g, \mathcal{R}_{\mathrm{in}}, \mathcal{R}_{\mathrm{out}}, \mathcal{L}, \Phi)$	Task specification containing the goal, required inputs and outputs, dependencies among inputs, and completion conditions.
$\psi_r = (u_r, f_r, q_r)$	Requirement for input resource r: its role in the task, required representation, and required content.
$e_r^\star$	Strongest available evidence about input resource r; $e_r^\star = \varnothing$ means that no usable content is available.
α_r	Construction choice for input resource r: RESTORE, FILL, SYNTHESIZE, or ABSTAIN.
$\mathrm{Paths}(A)$	Resource paths associated with workspace state or resource set A.
$y = (m, D_f)$	Outcome of a new solver, consisting of submitted response m and final state D_f.
$\mathcal{Q} = \{(y_\ell, \omega_\ell)\}_{\ell=1}^K$	Validation cases for the evaluator and their expected responses.
$\widetilde{\mathcal{T}} = (\widetilde{I}, \widetilde{D}_0, \widetilde{R}, \widetilde{V})$	Compiled task presented to new solvers and scored independently.

B.2 INFORMATION AVAILABLE TO THE COMPILER AND SOLVER

The compiler and solver receive different information. The compiler receives one execution record $x = (\tau, E, O)$, where $\tau = \{(h^t, a^t, e^t)\}_{t=1}^L$ contains the recorded interaction. The selected request and its necessary prior context are obtained from τ, while resource events, retained snapshots, manifests, and metadata may come from either τ or E. Recorded verdicts and reference artifacts, when available, belong to O. The compiler does not receive the original task package as additional input.

The new solver receives only $(\widetilde{I}, \widetilde{D}_0, \widetilde{R})$. The evaluator $\widetilde{V}$ runs separately on the solver's final response and workspace. The recorded trajectory, recorded outcome, reference artifacts, and compiler metadata are not shown to the solver. This information can identify content that must be withheld and can provide validation cases for an evaluator whose criteria are already fixed. It appears in the task shown to the solver only when independent evidence establishes that it belongs to the initial state. As in the main method, the request and evidence about the initial state may support the task specification. Artifacts produced during the recorded execution may identify content to withhold, and recorded outcomes may provide validation cases for an evaluator whose completion conditions are already fixed.

B.3 Inferring the Task Specification from Available Evidence

To identify the task supported by the record, ReTask locates the actionable request boundary a and the preceding context needed to interpret the request. It then groups resource observations by resource identity r and version $\nu(r, t)$. The relevant distinction is the state described by the evidence, rather than whether an observation occurs before or after the request in the recorded trajectory.

Evidence describing the version of a resource before the recorded agent changes it may enter $\mathcal{E}_0(a)$. For example, a read performed after the request but before the first modification can still describe the initial version of the resource. Created files, replacement content, observations of modified versions, validation outputs, and the final response instead enter $\mathcal{Z}_{\mathrm{src}}(a)$. Thus, the same path may contribute evidence about the initial state and also identify later content that must be withheld.

From $\mathcal{Z}_{\mathrm{src}}(a)$, the compiler identifies $\mathcal{X}_{\mathrm{src}}(a)$, the artifacts clearly produced during the recorded execution. These artifacts may identify content that must be absent from the Compiled initial state and may be matched to an output requirement already supported by the request, prior context, task rules, or input evidence. They cannot introduce a new goal, output requirement, or completion condition. Other recorded results are used only as evaluator validation cases after the task specification is fixed.

The task specification is

$$\mathcal{B} = \mathrm{InferTask}(q_a, \mathcal{E}_0(a); \mathcal{X}_{\mathrm{src}}(a)) = \big(g, \mathcal{R}_{\mathrm{in}}, \mathcal{R}_{\mathrm{out}}, \mathcal{L}, \Phi\big).$$

The goal g states the requested outcome and visible constraints. $\mathcal{R}_{\mathrm{in}}$ contains resources required at task start, including files that must be modified in place. $\mathcal{R}_{\mathrm{out}}$ contains outputs that the solver must create. $\mathcal{L}$ records dependencies among required inputs that are needed for the task, and $\Phi = \{\phi_j\}_{j=1}^{J}$ contains observable completion conditions.

For each input resource r, the compiler also records $\psi_r = (u_r, f_r, q_r)$, describing its role in the task, required representation, and required content. These requirements determine how the resource may be constructed in the next stage. Task inference does not condition on whether the recorded execution succeeds, so records from failed executions remain eligible for compilation.

B.4 Constructing an Unsolved Initial State

To realize the inputs required by the task specification, ReTask selects one construction operation for each $r \in \mathcal{R}_{\mathrm{in}}$ according to the available evidence:

$$\alpha_r \in \{\textsc{Restore}, \textsc{Fill}, \textsc{Synthesize}, \textsc{Abstain}\}.$$

RESTORE is used when the available evidence already satisfies ψ_r. FILL is used when part of the required content is observed and the missing portion can be inferred; the observed portion is retained rather than replaced. SYNTHESIZE is used when no usable original content is available but correctness depends on ψ_r rather than the exact identity of the original resource. If task correctness instead depends on unavailable original content, live state, credentials, or another dependency that cannot be justified from the record, ReTask selects ABSTAIN. Any required input assigned ABSTAIN prevents the task from being released.

When all required inputs can be constructed, the initial state is assembled as

$$\widetilde{D}_0 = \bigoplus_{r \in \mathcal{R}_{\mathrm{in}}} \alpha_r(r \mid \mathcal{B}).$$

The resulting workspace must satisfy every ψ_r for $r \in \mathcal{R}_{\text{in}}$, while

$$\text{Paths}(\mathcal{R}_{\text{out}}) \cap \text{Paths}(\widetilde{D}_0) = \varnothing.$$

Resources referenced by each required dependency in $\mathcal{L}$ must also be present. Construction uses $\mathcal{L}$ to preserve the dependency structure supported by the available evidence.

Files that must be edited remain at their initial versions. No recorded output, edit, intermediate result, or identified answer fragment may appear in $\widetilde{D}_0$. ReTask renders $\widetilde{I} = \text{Render}(g)$ and binds $\widetilde{R} = \text{Bind}(g, \{\psi_r\})$ to the tools and interaction rules required by the task. The resulting instruction, state, and protocol therefore provide what the solver needs without exposing the recorded solution.

B.5 Implementing and Validating Fixed Completion Conditions

The fixed completion conditions Φ determine how a new solver outcome is judged. The required output selects one of three evaluator types:

$$c \in \{\texttt{execute_code}, \texttt{validate_artifact}, \texttt{response_only}\}.$$

Executable code is tested through focused execution, such as unit tests, startup checks, or API probes. Artifact tasks are evaluated by parsing and inspecting the produced files. Tasks that require only a response are evaluated from the submitted response.

For a solver outcome $y = (m, D_f)$, the evaluator returns $(s(y), z(y), d(y))$. Here, $s(y) \in [0, 100]$ is the normalized score, $z(y)$ indicates whether evaluation completed, and $d(y)$ reports the evidence for each criterion. The evaluator implements the completion conditions Φ:

$$\widetilde{V} = \text{BuildEvaluator}(\Phi, \widetilde{D}_0, \widetilde{R}).$$

Writing $v_j(y) \in [0, 1]$ for the check associated with ϕ_j and w_j for its weight,

$$s(y) = \sum_{j=1}^{J} w_j v_j(y), \qquad \sum_j w_j = 100.$$

A valid outcome is complete when $s(y) \geq 60$. Invalid evaluator execution produces NO-SCORE. At least 60 points are assigned to correctness for the task, while at most 20 points may depend on simple properties such as file existence, naming, or directory placement.

ReTask validates the evaluator on

$$\mathcal{Q} = \{(y_\ell, \omega_\ell)\}_{\ell=1}^{K}.$$

Evaluator validation covers four cases:

- A scoring example y_{smk} must reach the evaluator's substantive checks and return a valid score.
- The unchanged initial state $y_{\text{noop}} = (\varnothing, \widetilde{D}_0)$ must satisfy $s(y_{\text{noop}}) < 60$.
- Negative cases y_{neg}, such as a missing artifact, corrupted output, or excessive edit, must lose the credit associated with the violated completion condition.
- The recorded outcome y_{src} may be tested when its artifacts correspond directly to the Compiled task.

The recorded outcome is evaluated only after Φ has been fixed. Its evaluator response can reveal that the constructed state or evaluator is inconsistent with the specification, but it cannot introduce, remove, or modify a completion condition.

A Compiled task must therefore satisfy three conditions: every required input satisfies ψ_r and all required dependency endpoints are present; target outputs and recorded results are absent; and every case in $\mathcal{Q}$ receives its required evaluator response. When the constructed state or evaluator violates these conditions, ReTask revises it. The compiler abstains when the record cannot support a task satisfying the fixed specification.

B.6 Compilation Procedure

Algorithm 1 connects task inference, state construction, and evaluator validation through the shared specification. If the first constructed state or evaluator violates one of these conditions, ReTask may revise the initial state or evaluator within a fixed budget. The inferred task specification, instruction, and interaction protocol remain unchanged:

$$(\mathcal{B}, \widetilde{I}, \widetilde{R})^{(i+1)} = (\mathcal{B}, \widetilde{I}, \widetilde{R})^{(i)}.$$

This invariant permits changes to the constructed state or evaluator without changing the task goal, required resources, or completion conditions after inspecting the outcome.

Algorithm 1 ReTask

Require: execution record $x = (\tau, E, O)$; repair budget B_{repair}
Ensure: Compiled task $\widetilde{\mathcal{T}}$ or ABSTAIN

identify actionable request q_a and required context
$(\mathcal{E}_0(a), \mathcal{Z}_{\text{src}}(a)) \leftarrow \text{Partition}(x, a)$
identify recorded artifacts $\mathcal{X}_{\text{src}}(a)$ to withhold
$\mathcal{B} \leftarrow \text{InferTask}(q_a, \mathcal{E}_0(a); \mathcal{X}_{\text{src}}(a))$
render $\widetilde{I}$ and bind interaction protocol $\widetilde{R}$
for $i = 0, \ldots, B_{\text{repair}}$ **do**
    for each $r \in \mathcal{R}_{\text{in}}$ **do**
        choose $\alpha_r \in \{\text{RESTORE}, \text{FILL}, \text{SYNTHESIZE}, \text{ABSTAIN}\}$
    end for
    if any required input receives ABSTAIN **then**
        return ABSTAIN
    end if
    assemble initial state $\widetilde{D}_0 \leftarrow \bigoplus_{r \in \mathcal{R}_{\text{in}}} \alpha_r(r \mid \mathcal{B})$
    build evaluator $\widetilde{V} \leftarrow \text{BuildEvaluator}(\Phi, \widetilde{D}_0, \widetilde{R})$
    check inputs, output exclusion, exclusion of recorded results, and evaluator behavior
    if all validation checks pass **then**
        return $\widetilde{\mathcal{T}} = (\widetilde{I}, \widetilde{D}_0, \widetilde{R}, \widetilde{V})$
    end if
    revise the initial state or evaluator that failed a condition
end for
return ABSTAIN

C EXPERIMENTAL PROTOCOLS

The protocols connect task compilation to two uses: comparing agent configurations on Workspace-Bench and collecting externally verified supervision on SWE-ReBench.

C.1 RUNTIME AND OPTIMIZATION CONFIGURATION

Compiler and solver configuration. Table C.1 gives the compiler and solver configurations; Table C.2 gives the SFT settings.

C.2 DATASETS AND THEIR ROLES

Workspace-Bench supplies the model-comparison tasks (Tang et al., 2026); SWE-ReBench supplies software issues with repository tests for verifying new solutions and selecting supervision (Badertdinov et al., 2025). Table C.3 reports the data sources and sample counts.

Recorded executions for comparative evaluation. The comparative evaluation uses one DeepSeek-V4-Flash-0731 execution (DeepSeek-AI, 2026a;b) for each of the 100 public Workspace-Bench Lite tasks. Each record contains the user request, tool events, and workspace observations. The set contains 77 successful and 23 failed executions, with one Compiled task per record.

Models and execution protocol for comparative evaluation. The seven configurations in Table C.4 independently attempt every Compiled task under the budgets in Table C.1. A configuration consists of a model, reasoning mode, and harness; result tables abbreviate it by model name. Native scores come from the official Workspace-Bench harness. Compiled runs share the task input and submission format and are scored by their respective task evaluators. We compare these configurations using Native aggregate ranks and Compiled TCR@60 (Appendix C.3).

Recorded executions for supervision. The recorded executions correspond to SWE-ReBench tasks (Badertdinov et al., 2025) and come from a public trajectory release collected with Qwen3-Coder-480B-A35B-Instruct and OpenHands v0.54.0 (Trofimova et al., 2025; Wang et al., 2024).

Table C.1: **Compiler and solver configurations.**

Component	Configuration
Compiler model	DeepSeek-V4-Flash-0731 (DeepSeek-AI, 2026a;b), temperature 0.6, reasoning disabled; infers task specifications and constructs tasks in both domains.
Compilation budget	Claude Code harness, at most 250 turns and 5,400 s per task; at most one revision with the task specification fixed.
Completion rule	Normalized evaluator score in $[0, 100]$; completion threshold 60; validation follows Appendix B.5.
Workspace-Bench runs	Temperature 0.6, at most 250 turns and 7,200 s for each model on each task, with the same task input and submission format.
SWE teacher runs	GLM-5.2 (GLM-5 Team, 2026; Z.ai, 2026), temperature 0.6, SWE-agent (Yang et al., 2024) with `bash`, `str_replace_editor`, and `submit`; at most 50 iterations and 7,200 s per attempt.

Table C.2: **SFT optimization protocol.** Native and ReTask use the same settings within each base model; only the training corpus changes.

Parameter	Configuration
Optimizer	AdamW (`foreach`), $\epsilon = 10^{-8}$, maximum gradient norm 1.0
Learning rate schedule	Peak 5×10^{-6}, cosine decay to 10^{-7}, no warmup
Batch and context	Local batch size 1, global batch size 32, sequence length 262,144
Random seed	42
Update budget	200 steps for Qwen3-30B-A3B-Thinking-2507; 100 steps for Qwen3.5-35B-A3B
Data exposure	One successful trajectory per original task solved in both conditions; matched training exposure measured in tokenizer tokens

Table C.3: **Data used for comparative evaluation and supervision.** Recorded executions provide evidence for task construction. Teacher trajectories are newly collected on the resulting tasks and retained when they satisfy the stated correctness criterion.

Data	Origin and scale	Experimental subset	Role
Workspace-Bench Lite	100 public tasks with one recorded execution per task	One Compiled task per recorded execution	Comparison of Native and Compiled model rankings
Public SWE-ReBench executions	67,074 OpenHands trajectories over 6,306 instances; 3,792 instances have at least one resolved execution	3,000 tasks with resolved source trajectories and repository images: 1,966 fixed plus 1,034 sampled with seed 42	Evidence for constructing SWE tasks
ReTask teacher collection	3,889 GLM-5.2 attempts over 2,927 original tasks; multiple Compiled variants per task are allowed	2,814 submitted nonempty patches; 2,807 evaluable patches; 2,040 test successes; 1,614 successful original tasks	Compiled training trajectories
Native teacher collection	GLM-5.2 attempts on the same 3,000 original tasks in their repositories	1,694 successful original tasks	Native training trajectories
Paired SWE supervision	Native and Compiled solutions that pass the tests for the same original tasks	1,123 trajectories per condition; matched tokenizer-token exposure within each base model	Native SFT versus ReTask SFT
Downstream SWE evaluation	SWE-bench Verified, SWE-bench Pro, and SWE-bench Multilingual	Official splits; no overlap with the paired training tasks	Downstream resolved rate

We use the version standardized under the Agent Data Protocol (Song et al., 2025). The 3,000-task compilation pool is specified in Table C.3. GLM-5.2 then produces new trajectories by solving the Compiled tasks.

Teacher trajectories on matched tasks. GLM-5.2 collects new trajectories by independently solving the Compiled and Native tasks under the conditions in Table C.5.

Table C.4: **Public model identities for the Workspace-Bench comparison.**

Public model name	Provider	Reasoning mode
DeepSeek-V4-Flash-0731 (DeepSeek-AI, 2026a;b)	DeepSeek-AI	default
DeepSeek-V4-Pro-0813 (DeepSeek-AI, 2026a;c)	DeepSeek-AI	default
GLM-5.2 (GLM-5 Team, 2026; Z.ai, 2026)	Z.ai	default
Qwen3.8-Flash-Next (Qwen Team, 2026c)	Qwen Team	default
Tencent Hy3 (Tencent Hy Team, 2026)	Tencent Hy Team	default
Qwen3.6-27B (Qwen Team, 2026b)	Qwen Team	default
Qwen3.5-35B-A3B (Qwen Team, 2026a)	Qwen Team	default

Table C.5: **Native and ReTask conditions for collecting supervision.** The conditions share the original task, teacher, agent protocol, test criterion, and matched SFT exposure measured in tokenizer tokens; they differ in instruction and initial state.

Protocol element	ReTask condition	Native condition
Original task	Same SWE-ReBench instance	Same SWE-ReBench instance
Initial state	New Git workspace containing only compiled `task_data`	Original repository at the task's base state
Instruction	Compiled instruction, supplied verbatim	Native SWE instruction
Trajectory collection	GLM-5.2; identical SWE-agent prompt, tools, temperature, 50-iteration limit, and 7,200 s timeout	Same
Success criterion	Submitted nonempty patch satisfying the original repository's FAIL-TO-PASS and PASS-TO-PASS tests	Same
SFT exposure	One trajectory per original task solved in both conditions; exposure measured in tokenizer tokens is fixed within each base model	Same

The intersection of the 1,614 original tasks solved in Compiled and the 1,694 solved in Native contains 1,123 tasks. We select one successful trajectory per condition and original task for SFT, with matched training exposure measured in tokenizer tokens within each base model.

Downstream effect of supervision. For each base model, Base receives no additional trajectory training, while Native SFT and ReTask SFT use their corresponding paired corpus under Table C.2. We evaluate Qwen3.5-35B-A3B and Qwen3-30B-A3B-Thinking-2507 with the same agent scaffold on SWE-bench Verified, Pro, and Multilingual (Jimenez et al., 2024); the latter model name is shortened to Qwen3-30B-A3B in result tables. Matching by original task confirms zero overlap between the 1,123 training tasks and each downstream evaluation set. Table 2 reports resolved rates from one training run per configuration using seed 42.

C.3 METRICS

Task completion rate. For model k under its fixed agent configuration, let $\mathcal{J}_k$ be the set of tasks with a valid evaluator result and let $s_{jk} \in [0, 100]$ be its normalized evaluator score on task j. The fixed completion rule marks run (j, k) complete when $s_{jk} \geq 60$. The default aggregate Compiled score is

$$\mathrm{TCR@60}(k) = \frac{100}{|\mathcal{J}_k|} \sum_{j \in \mathcal{J}_k} \mathbf{1}[s_{jk} \geq 60]. \tag{5}$$

A run timeout, truncated trajectory, or evaluator failure produces no valid score and is excluded from $\mathcal{J}_k$. For a task subset and model panel, pooled TCR@60 is the percentage of all valid model and task runs whose score reaches 60. We also recompute TCR@60 with runs without valid scores counted as failures and on the 90 tasks scored for all seven models.

Ordering agreement. The primary comparative evaluation result uses Spearman ρ, Kendall τ_b, and pairwise order accuracy, the fraction of model pairs ordered in the same direction by the Native aggregate score and Compiled TCR@60. The main result uses all valid task measurements,

with mean evaluator score as an additional aggregation. Category orderings use the overall Native ordering as their reference, while difficulty and workspace-size orderings use the full Compiled ordering. Spearman correlations assign average ranks to tied scores, including ties created by task resampling; Kendall τ_b adjusts for ties. Pairwise agreement excludes tied pairs. The task bootstrap holds the Native aggregate scores fixed and resamples the 90 complete tasks 10,000 times with seed 20260914.

State construction. Let $\mathcal{R}_{\text{used}}$ contain resources that are read, transferred, or explicitly referenced in the recorded execution, and let $m(r) \in \{0, 1\}$ indicate whether resource r is materialized in the compiled workspace. Recall of used resources is

$$\text{Rec}_{\text{used}} = \frac{\sum_{r \in \mathcal{R}_{\text{used}}} m(r)}{|\mathcal{R}_{\text{used}}|}.$$

We report resource availability separately from the construction method. Exact copies, resources completed from partial observations, and synthesized replacements all count as available, but only exact copies preserve the original bytes.

Supervision metrics. For trajectory collection, a verified success is a submitted nonempty patch that satisfies the original repository's FAIL-TO-PASS and PASS-TO-PASS tests. The verified success rate divides this count by all Compiled attempts ($2{,}040/3{,}889 = 52.5\%$); the conditional pass rate divides it by the number of evaluable patches ($2{,}040/2{,}807 = 72.7\%$). Neither rate deduplicates attempts by original task. Successful task counts do deduplicate, and the paired training set contains the intersection of successful original tasks in the two conditions. Downstream resolved rates use each benchmark's official task denominator. For Compiled and Native patches from the same original task, let F_j^{C} and F_j^{N} denote the sets of modified source files. Their agreement is

$$J_j = \frac{|F_j^{\text{C}} \cap F_j^{\text{N}}|}{|F_j^{\text{C}} \cup F_j^{\text{N}}|}.$$

We also report the ratio between the gain from ReTask SFT over Base and the corresponding gain from Native SFT. Trajectory comparisons pair the same original tasks; reported confidence intervals resample tasks rather than individual messages or tool calls.

D Comparative evaluation: extended results

These analyses examine aggregate ranking agreement and the model differences resolved by Compiled tasks, extending Section 4.2 with sensitivity checks and task-level outcomes. Metrics follow Appendix C.3.

D.1 Agreement between model rankings

Excluding the source model gives $\rho = 1.000$ over the remaining six models. Counting missing scores as failures leaves the seven-model ordering unchanged; the common 90-task panel gives $\rho = 0.927$ with average ranks for ties (Figure 5a).

Resampling the 90 complete tasks gives a 95% rank correlation interval of $[0.393, 0.955]$. Among 10,000 replicates, 99.81% have $\rho > 0$ and 94.95% have $\rho \geq 0.5$ (Figure 5b). Aggregating mean scores gives $\rho = 0.821$ and Kendall $\tau = 0.619$.

Task-removal sensitivity. Removing each task in turn, recomputing TCR@60 over remaining valid runs, and holding the Native reference fixed gives $\rho \in [0.929, 0.964]$.

Paired comparison protocol. For models A and B, we compute the completion-rate difference on a shared task set $\mathcal{J}_{AB}$:

$$\delta_{AB} = \frac{100}{|\mathcal{J}_{AB}|} \sum_{j \in \mathcal{J}_{AB}} \big(\mathbf{1}[s_{jA} \geq 60] - \mathbf{1}[s_{jB} \geq 60]\big).$$

The primary paired analysis uses the same 90 complete tasks for all 21 model pairs. A sensitivity analysis instead uses all tasks jointly scored for each pair, giving 93–98 tasks. Each pair's point estimate and interval use a common task set. We draw 10,000 task samples with replacement, using seed 20260914 and resampling the paired outcomes together. Complete-panel comparisons share the same resamples, and each task set reuses its resamples across thresholds. Percentile 95% intervals use linear quantile interpolation.

Two-sided exact McNemar tests use the tasks where the models' binary outcomes disagree. We apply Holm correction across all 21 pairs, separately for each task-set definition and threshold, at level 0.05. Bootstrap intervals are pointwise intervals, whereas the adjusted tests account for the family of comparisons. The intervals quantify task-sampling uncertainty conditional on the fixed Compiled evaluator outputs and Native aggregate scores. Table D.1 gives all pairs under both task-set definitions at threshold 60.

Paired outcomes. Under both task-set definitions, DeepSeek-V4-Pro exceeds Hy3, Qwen3.6-27B, and Qwen3.5-35B-A3B after Holm correction. The remaining pairs do not pass this correction; their estimates and intervals appear in Table D.1.

Nearby completion thresholds. Thresholds 55 and 65 are post-hoc sensitivity checks; the primary threshold remains 60. Reusing the recorded runs with fixed task membership gives the results in Table D.2. The Pro–Qwen3.5 difference passes Holm correction under both task-set definitions at all three thresholds.

D.2 Tasks that distinguish models

Of the 100 tasks, 53 produce both passing and failing outcomes, 14 are passed by no model, and 33 are passed by every model with a valid score. Of the last group, 30 have scores for all seven models and three have missing scores. Figure D.1 reports the complete 7×100 score matrix.

D.3 Task-group summary

Table D.3 groups Compiled outcomes by task category, difficulty, and workspace-size quartile, complementing Figure 1 and Table 1. Native per-task model scores are unavailable: category correlations use the overall Native ordering, while difficulty and size correlations use the full Compiled ordering.

D.4 Evaluation with synthesized resources and failed executions

We examine solvability and model differentiation under two separately analyzed conditions: synthesized resources and failed source executions. The synthesis grouping includes 97 tasks with resources used in the recorded execution; three tasks have no such resource. The groups synthesize none, at most half, or more than half of these resources. The majority-synthesis group contains 49 tasks and synthesizes 261 of 320 used resources. Independently, all 100 tasks are grouped by the recorded outcome: 77 successful and 23 failed.

Table D.4 reports solvability, mixed outcomes, and ordering over the six non-source models. Its rank reference is the full Compiled collection for the same panel, including each subset. TCR intervals resample tasks within each group 10,000 times.

Table D.1: **All model pairs at TCR@60 under both task-set definitions.** A has the higher Native aggregate score; δ_{AB} is Compiled $A - B$ in percentage points. The interval is a pointwise 95% paired bootstrap interval; p_H is the Holm-adjusted two-sided exact McNemar p-value over all 21 pairs. The complete panel uses 90 tasks; jointly scored task sets have the indicated N and may differ between pairs. Shaded rows have $p_H < 0.05$ under both definitions. DS-Flash/Pro denote the two DeepSeek-V4 configurations; Q3.8-Flash, Q3.6-27B, and Q3.5-35B denote Qwen3.8-Flash-Next, Qwen3.6-27B, and Qwen3.5-35B-A3B.

		Complete panel ($N = 90$)		Jointly scored per pair		
A	B	δ_{AB} [95% CI]	p_H	N	δ_{AB} [95% CI]	p_H
DS-Flash	DS-Pro	−10.00 [−17.78, −2.22]	0.3594	96	−9.38 [−17.71, −2.08]	0.5550
DS-Flash	GLM-5.2	+4.44 [−3.33, +12.22]	1.0000	95	+3.16 [−4.21, +11.58]	1.0000
DS-Flash	Q3.8-Flash	+4.44 [−4.44, +13.33]	1.0000	96	+6.25 [−2.08, +14.58]	1.0000
DS-Flash	Hy3	+4.44 [−3.33, +12.22]	1.0000	95	+5.26 [−2.11, +12.63]	1.0000
DS-Flash	Q3.6-27B	+8.89 [+0.00, +17.78]	1.0000	95	+9.47 [+0.00, +18.95]	1.0000
DS-Flash	Q3.5-35B	+12.22 [+2.22, +22.22]	0.3991	93	+11.83 [+2.15, +21.51]	0.4523
DS-Pro	GLM-5.2	+14.44 [+5.56, +24.44]	0.1296	97	+11.34 [+2.06, +20.62]	0.5550
DS-Pro	Q3.8-Flash	+14.44 [+4.44, +24.44]	0.1296	98	+14.29 [+5.10, +23.47]	0.1190
DS-Pro	Hy3	+14.44 [+6.67, +22.22]	0.0186	97	+14.43 [+7.22, +22.68]	0.0099
DS-Pro	Q3.6-27B	+18.89 [+10.00, +27.78]	0.0044	97	+18.56 [+10.31, +27.84]	0.0024
DS-Pro	Q3.5-35B	+22.22 [+13.33, +32.22]	0.0008	95	+21.05 [+11.58, +30.53]	0.0008
GLM-5.2	Q3.8-Flash	+0.00 [−8.89, +8.89]	1.0000	97	+4.12 [−5.15, +13.40]	1.0000
GLM-5.2	Hy3	+0.00 [−8.89, +8.89]	1.0000	97	+3.09 [−6.19, +12.37]	1.0000
GLM-5.2	Q3.6-27B	+4.44 [−5.56, +14.44]	1.0000	95	+7.37 [−2.11, +16.84]	1.0000
GLM-5.2	Q3.5-35B	+7.78 [−2.22, +17.78]	1.0000	94	+8.51 [−2.13, +18.09]	1.0000
Q3.8-Flash	Hy3	+0.00 [−8.89, +8.89]	1.0000	97	+0.00 [−8.25, +8.25]	1.0000
Q3.8-Flash	Q3.6-27B	+4.44 [−5.56, +14.44]	1.0000	97	+4.12 [−5.15, +13.40]	1.0000
Q3.8-Flash	Q3.5-35B	+7.78 [−3.33, +18.89]	1.0000	94	+6.38 [−4.26, +17.02]	1.0000
Hy3	Q3.6-27B	+4.44 [−3.33, +12.22]	1.0000	95	+3.16 [−4.24, +11.58]	1.0000
Hy3	Q3.5-35B	+7.78 [−2.22, +17.78]	1.0000	94	+7.45 [−2.13, +17.02]	1.0000
Q3.6-27B	Q3.5-35B	+3.33 [−6.67, +13.33]	1.0000	93	+2.15 [−7.53, +11.83]	1.0000

Table D.2: **Threshold sensitivity under two paired task-set definitions.** Directions are relative to Native. Holm count is the number of significant pairs after correction over all 21 tests within each row. A single aggregate ρ is defined only for the common 90-task panel.

Task set	Threshold	N	ρ	Agree / reverse / tie	Holm count
Complete panel	55	90	0.955	19 / 1 / 1	2
Complete panel	60	90	0.927	17 / 1 / 3	3
Complete panel	65	90	0.929	19 / 2 / 0	1
Jointly scored per pair	55	93–98	–	20 / 1 / 0	3
Jointly scored per pair	60	93–98	–	19 / 1 / 1	3
Jointly scored per pair	65	93–98	–	20 / 1 / 0	2

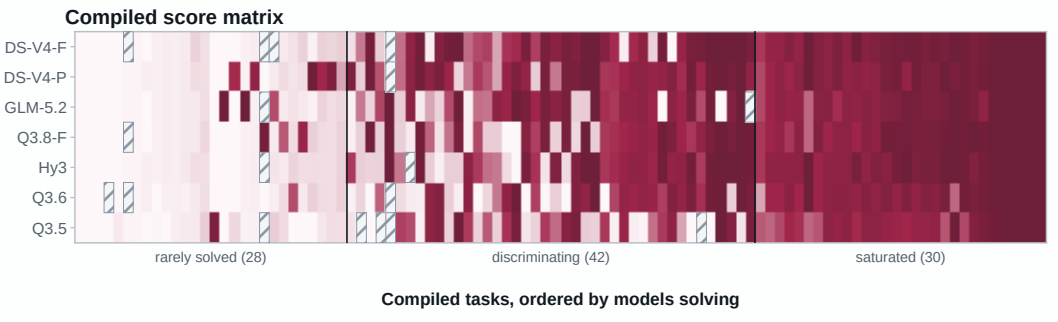


Figure D.1: **Normalized evaluator scores for all seven models on all 100 tasks.** Each column is a task, ordered by the number of models that solve it; 682 of the 700 model and task pairs have a valid measurement. Hatching denotes the 18 missing scores, and the colour boundary marks the completion threshold of 60.

Table D.3: **Compiled outcomes by task group.** TCR@60 pools valid runs of all seven models. Mixed is the percentage of tasks with both passing and failing models. Category ρ uses the overall Native reference; difficulty and size ρ use the full Compiled reference.

Group	N	TCR@60 (%)	Mixed (%)	ρ
TASK CATEGORIES				
Operations	32	69.2	34.4	0.88
Logistics	30	61.8	73.3	0.65
Research	17	64.0	47.1	0.74
Backend	11	58.4	54.5	−0.31
Product	10	51.5	60.0	0.73
DIFFICULTY STRATA				
Easy	14	69.1	57.1	0.355
Medium	54	54.5	55.6	1.000
Hard	32	75.2	46.9	0.865
QUARTILES BY WORKSPACE SIZE				
Q1 (1–4 files)	25	50.9	64.0	0.883
Q2 (4–5 files)	25	61.2	64.0	0.371
Q3 (5–10 files)	25	70.3	40.0	0.857
Q4 (10–327 files)	25	70.7	44.0	0.883

Table D.4: **Non-source model outcomes and ordering under compilation conditions.** TCR@60 pools valid runs of the six models other than the source model. Intervals use task bootstrap within each group. Solved and Mixed count tasks with at least one passing model and with both outcomes, respectively. Order agreement and ρ compare against the full Compiled six-model ordering; agreement excludes tied pairs.

Group	N	TCR@60 [95% CI]	Solved	Mixed	Agreement	ρ
Fraction of used resources synthesized						
None	33	54.9 [41.8, 67.7]	26/33	18/33	13/15	0.886
At most half	15	79.8 [60.7, 95.5]	13/15	3/15	9/9	0.845
More than half	49	61.9 [51.6, 71.8]	44/49	27/49	12/15	0.771
Outcome of the recorded execution						
Recorded success	77	64.1 [55.7, 72.2]	68/77	39/77	15/15	1.000
Recorded failure	23	56.9 [39.9, 73.5]	18/23	11/23	13/14	0.928

The failure group contains one tied pair. The at-most-half group contains six tied pairs, leaving nine strict comparisons. The synthesis and recorded-outcome groups are analyzed separately and may overlap.

E SUPERVISION: VERIFIED SOLUTIONS AND PAIRED CORPUS

The analyses below characterize the paired corpus used for downstream training: the 1,123 original tasks solved in both conditions, with both patches accepted by the original repository tests. Corpus selection and training exposure follow Appendix C.2; Table 2 reports downstream resolved rates.

E.1 REPAIR-FILE COVERAGE

Native repair coverage measures whether Compiled workspaces contain every source file modified by the independently collected Native patch. Table E.1 partitions the 1,123 paired tasks using the 25th, 50th, and 75th percentiles of Compiled text lines: 370.5, 552.0, and 810.5. Tasks with the same size stay in the same group.

Table E.1: **Repair-file agreement across Compiled workspace sizes.** All 1,123 tasks are solved in both conditions. Identical source files counts pairs modifying the same source-file set. Native repair coverage counts Compiled workspaces containing every source file modified by the Native solution. Counts and percentages use the N tasks in each row.

Compiled text lines	N	Identical source files	Native repair coverage
39–370	281	271/281 (96.4%)	273/281 (97.2%)
371–552	283	263/283 (92.9%)	269/283 (95.1%)
553–810	278	261/278 (93.9%)	268/278 (96.4%)
811–37,221	281	260/281 (92.5%)	271/281 (96.4%)

E.2 PAIRED SOLUTION AND TRAJECTORY SUMMARY

Compiled and Native patches modify identical source-file sets in 93.9% of the paired tasks. Their mean source-file Jaccard similarity is 0.967 over 1,119 defined pairs (Figure 7).

The paired successful trajectories average 21.56 assistant steps in Compiled and 25.91 in Native. The paired difference is -4.35 steps, with a 95% task-bootstrap interval of $[-5.08, -3.62]$. SFT exposure is matched in tokenizer tokens as specified in Table C.2.

F COMPILATION CASE STUDIES

Figures F.1–F.6 provide six examples of how a shared task specification connects execution evidence to an unsolved initial state and an evaluator of observable completion. The examples cover exact restoration, filling from partial evidence, replacement that preserves an input's role, distractor preservation, dependencies across directories, and an evaluator that rejects an over-broad result. Displayed paths, observations, and counts come from the corresponding Compiled task; repeated files and tool calls are collapsed, and Chinese labels are translated for display. The score shown in each figure is the recorded execution's Native rubric score.

Together, the examples show how one execution record can support exact reuse or replacement that preserves the role of an input while keeping recorded outputs outside the solver state. The cases cover both successful and failed records; aggregate comparison behavior is evaluated separately in Section 4.2.

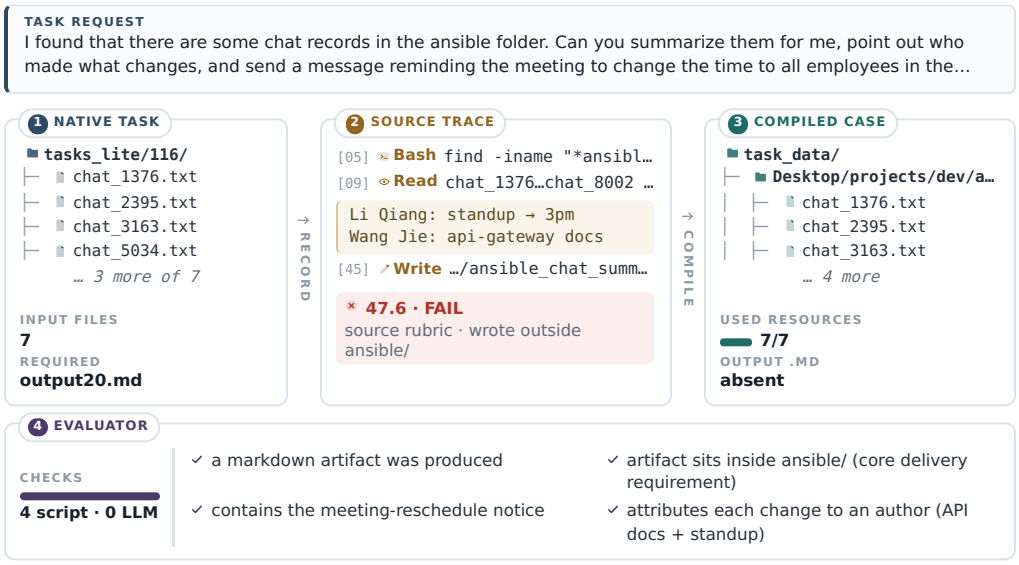


Figure F.1: **Task 116: task-start evidence remains available after a failed execution.** The recorded agent wrote its summary outside the required folder and scored 47.6, but the seven chat logs it read remain available with identical bytes. The Compiled task restores all seven, excludes the recorded answer, and evaluates attribution, citation, meeting details, and output placement.

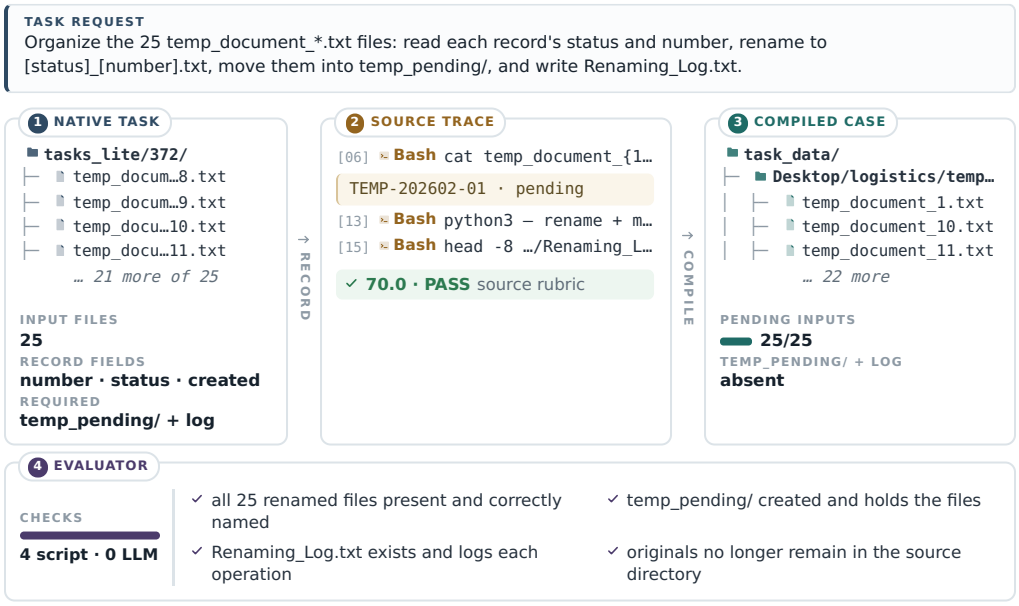


Figure F.2: **Task 372: a completed file transformation is reconstructed as an unsolved task.** All 25 pending documents are present and none of the products are: the renamed files, the target directory and the log are absent. The evaluator checks the complete final state rather than replaying the recorded command sequence.

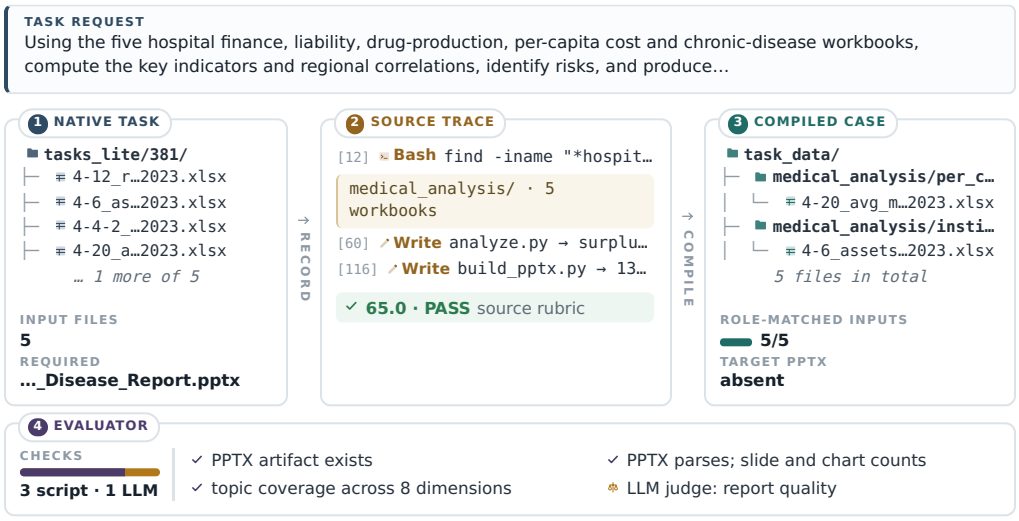


Figure F.3: **Task 381: partial workbook observations support five structured replacement inputs.** The trajectory exposes sheet names, frame shapes, field labels and selected values, but never the workbooks themselves. The compiled task rebuilds five inputs carrying those roles while withholding the requested presentation; the evaluator checks derived quantities and report structure rather than strings.

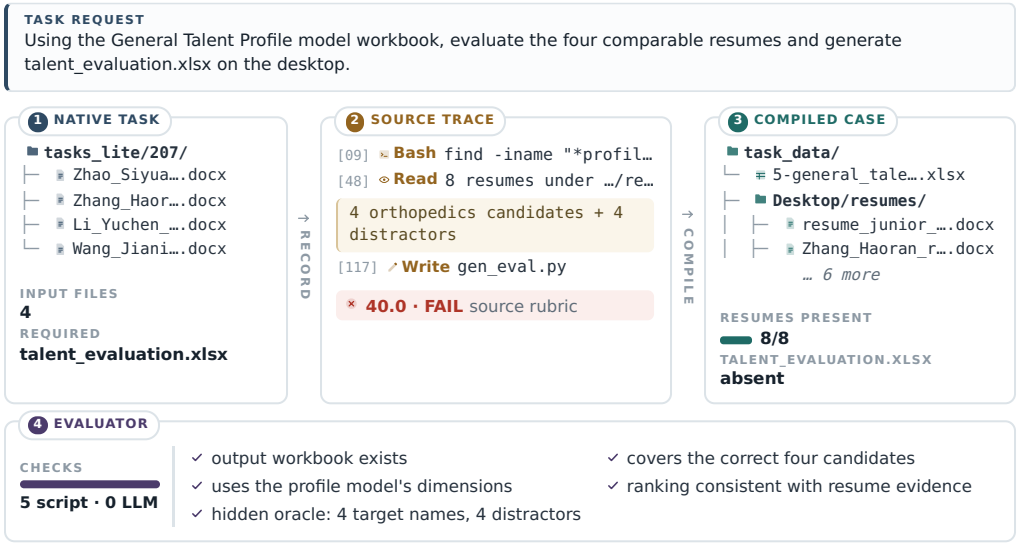


Figure F.4: **Task 207: the Compiled state retains relevant distractors.** Eight resumes are restored, only four of which form the comparable cohort. The evaluator holds the four target names and the four distractors as a hidden oracle and checks the ranking against resume evidence, so a solver that grades the wrong four is penalized rather than rewarded for producing a correctly formatted table.

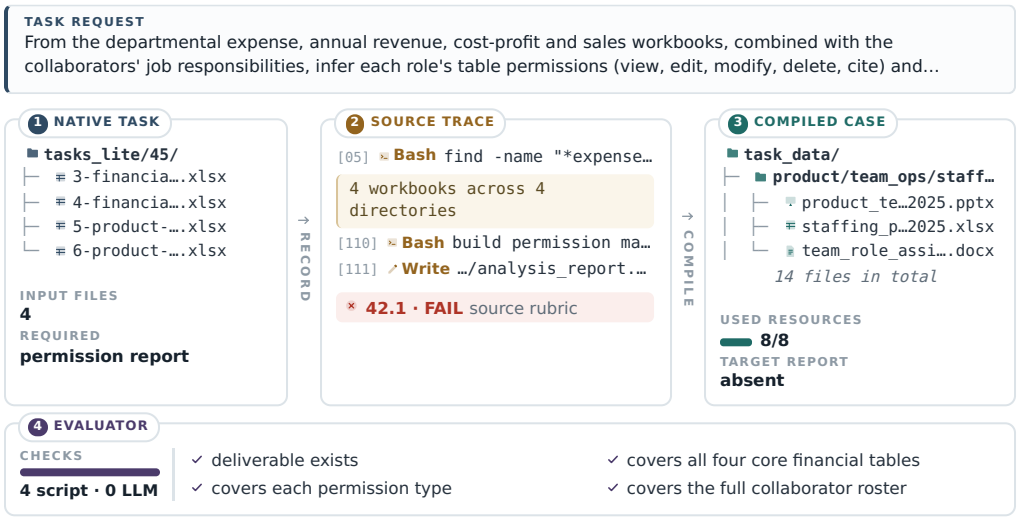


Figure F.5: **Task 45: the Compiled state preserves dependencies across directories.** The four financial workbooks and the directory of job descriptions are spread across four directories in the recorded execution. The Compiled workspace reproduces that layout, so the solver must still join evidence across directories to infer each role's table permissions rather than reading a single flattened folder.

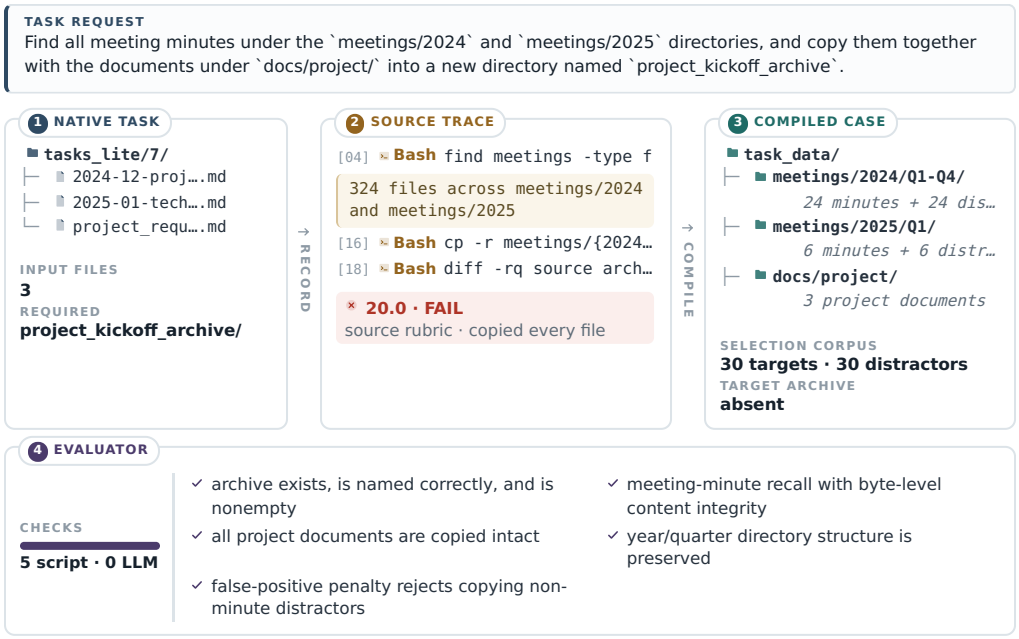


Figure F.6: **Task 7: the evaluator rejects an over-broad completion.** The recorded execution copied all 327 files under the two meeting directories and the project directory, including documents that were not meeting minutes, and scored 20.0. The Compiled state retains 30 target meeting minutes, 30 non-minute distractors, and three project documents while withholding the target archive. Its evaluator jointly checks target recall, byte-level content integrity, directory structure, and false positives, so copying every available file does not satisfy the task.

G REFERENCE PROMPT TEMPLATES

The prompts below implement task inference, unsolved-state construction, and evaluator construction around a shared task specification. They show which information each stage can use: the first prompt infers $\mathcal{B}$, the second constructs $\widetilde{D}_0$, and the third implements $\widetilde{V}$ from the fixed completion conditions Φ. Angle-bracketed fields are populated from the execution record $x = (\tau, E, O)$ or intermediate compiler state. The evaluated solver receives only $(\widetilde{I}, \widetilde{D}_0, \widetilde{R})$ and never sees these prompts or information reserved for the compiler. After construction, the compiler validates the evaluator with the scoring example y_{smk}, the unchanged initial state, negative cases designed for the task, checks that recorded results are excluded, and the recorded outcome when it maps to the Compiled task. Together, these checks implement the validation procedure in Section B.5.

Prompt for Task Specification Inference

ROLE AND GOAL

Infer a task specification and identify the evidence about the state before the request that is needed to complete the task in an offline environment.

COMPILER INPUTS (NOT SOLVER CONTEXT)

Current user request: `<CURRENT_USER_REQUEST>`

Candidate system instruction: `<SYSTEM_TASK_CANDIDATE>`

Indexed prior context: `<INDEXED_CONTEXT>`

Candidate resources and tool events: `<EVIDENCE_CANDIDATES>`

TASK INFERENCE RULES

1. Define the task shown to the solver first. If the current user request already states the operation, preserve it without restating, polishing, or strengthening the user's wording. If the current input is only data, dialogue, retrieved content, or parameters and the system context defines the operation, extract the smallest task instruction, the policy visible to the solver, and the output schema. Do not tie them to a particular product.

2. Preserve every label set, candidate set, threshold, field meaning, decision rule, and output field that changes the expected result. Remove product and model identity, private orchestration labels, proprietary tool names, prescribed call order, and generic instructions for controlling an agent. Describe a tool by its general capability only when using that capability is itself part of the task.

3. Distinguish tasks solvable from fixed local inputs, tasks that require an external read, and tasks that require an external action. Do not replace an external read or action with a written claim that it occurred.

4. Use only evidence about the state before the request. An observation made after the request remains eligible when it describes the version immediately before the first modification. The replaced span of an edit may support the preceding version. Newly created paths, replacement content, observations after a modification, validation outputs, final artifacts, recorded responses, and recorded scores are content produced during the execution and must not define the task. They may identify content that must be withheld and may be matched to an output requirement already supported by the current request, visible context, visible policy, or evidence about the prior state; they cannot introduce a new goal, output requirement, or acceptance criterion.

5. If the request depends on an earlier topic, object, or state, retain the minimum prior context needed to resolve it. Mark the task as needing context only when the available history is genuinely insufficient. If the visible input contains no operation, classify it as not a task.

6. Classify every resource referenced by the task into exactly one of two disjoint sets: `required_resource_ids` for inputs or files edited in place, and `output_resource_ids` for newly created targets or final deliverables. An existing file to be modified is an input; a target directory or file that must be created is an output. Unrelated exploration files may remain unclassified.

7. For each required resource, select how the task uses it: `preserve_original`, `read_structured`, `read_semantic`, `inspect_visual`, or `locate_only`. Also select the required coverage: `presence`, `selected_fields`, or `full`. Choose from the requested operation, not from the file suffix or the recorded agent's exploration strategy.

8. For a required dependency absent from the candidate set, cite the supporting evidence and choose exactly one route: `fixture` when a fixed local equivalent is supported, `source_backfill` when content from the original project must be recovered, `specialized_runtime` for a required browser, service, editor, or tool, and `not_offlineable` for live private state that cannot be fixed locally.

9. Define observable acceptance criteria using only `task_instruction`, `current_input`, `visible_context`, `visible_policy`, or a required `resource`. Each criterion must include an evidence source and a direct evidence quote. Do not add criteria from the recorded solution or later execution.
10. Select the evaluator from the requested deliverable, not from the tools used to produce it: `execute_code` for executable code behavior, `validate_artifact` for documents, data, reports, or images, and `response_only` for explanations or structured answers. Set `output_surface` independently to `natural_response`, `structured_response`, `action_trace`, `workspace_artifact`, or `code_behavior`.

TASK SPECIFICATION

Return a structured task specification with the following conceptual structure:

```
{
 "task": {
 "status": "ready|needs_context|not_a_task",
 "instruction": "minimal instruction without product names",
 "visible_policy": ["rule visible to the solver"],
 "output_surface": "response|action|artifact|code",
 "output_schema": "minimal schema or empty"},
 "resources": {
 "inputs": [{"resource": "input",
 "usage": "preserve|read|inspect|locate",
 "scope": "presence|selected_fields|full"}],
 "outputs": ["output"],
 "missing": [{"name": "...",
 "strategy": "fixture|backfill|runtime|abstain"}]},
 "runtime": {"requirement": "local|read|action",
 "capabilities": ["generic capability"]},
 "evaluation": {
 "family": "execute_code|validate_artifact|response_only",
 "criteria": ["observable completion condition"]}
}
```

Prompt for Task Constructor: Initial State

ROLE AND GOAL

Construct a local workspace in which the compiled task is solvable but remains unsolved.

COMPILER INPUTS (NOT SOLVER CONTEXT)

Fixed solver instruction: `<SOLVER_INSTRUCTION>`

Task specification: `<TASK_SPECIFICATION>`

Construction brief: `<INPUTS R1...Rn/D1...Dn; RELATIONS; OUTPUTS; DIFFICULTY; EVALUATION>`

Evidence view: `<EVIDENCE ABOUT INITIAL STATE; EXCLUSIONS FROM RECORDED ACTIONS AND OUTCOMES; SOURCE EXECUTION FOR VALIDATION>`

STATE CONSTRUCTION RULES

1. Treat the construction brief as an index, not as a replacement for evidence. Inspect the manifest, full recorded trajectory, and context brief for realistic content, relations, boundary conditions, and scale. The recorded execution is available only to the compiler: observations about the state before the request may support construction, whereas recorded actions and outcomes may only identify exclusions, match an already supported output, or validate the evaluator. They cannot introduce a new task requirement and must never enter the solver prompt or initial state. The solver instruction and exact inputs must not be changed.
2. Materialize every required input according to its declared use, required similarity, scope, format, and `preserve_task_properties`. For `reuse_or_materialize_observed`, reconstruct content supported by evidence about the prior state. For partial evidence, preserve every verified fragment and fill only the missing coverage allowed by the specification.

3. Use `synthesize_replacement` only when the specification authorizes replacement. A synthesized resource may change names, bytes, and values, but must preserve its task role, required operations, modality, schema, coverage, and relations. If exact identity, recovery from the original project, a specialized runtime, or live state is required and unavailable, abstain rather than fabricate an input.

4. Preserve the task specification as a whole: the minimum number of inputs, the number of inputs of each modality, the number of outputs, relationships between resources, and difficulty requirements. Do not merge distinct resources, delete joins, reduce a collection to one example, or turn a workflow involving multiple files into a toy task.

5. Produce real, parseable files. Visual fixtures must be valid images with distinguishable content; structured fixtures must satisfy their table or schema relationships; artifacts that must be preserved, moved, or renamed must contain real bytes. Additional support files may establish realistic project context but cannot substitute for a required `Rn`/`Dn` input.

6. Keep all synthesized content internally consistent. Provide an oracle for the evaluator that is hidden from the solver, or make the evaluator derive expected values from the visible fixtures. Never encode the recorded agent's final answer as the oracle.

7. Construct an unsolved initial state. Exclude every target output, content created while solving the task, replacement span, observation after a modification, recorded edit, final artifact, answer, and comment that reveals the solution. Do not place input values or an answer inventory in the prompt to bypass workspace construction.

8. Map each `mapping_required=true` resource or fixture dependency to one distinct existing path. A `mapping_required=false` resource is already materialized by the compiler and must not be remapped or overwritten.

CONSTRUCTED STATE

Write only the resources authorized for the solver under `task_data/`. The following construction map is returned to the compiler and is not shown to the solver:

```
{"version": 2,
 "resources": {"R1": "relative/input.ext",
 "D1": "relative/fixture.ext"},
 "extras": [{"path": "relative/support.ext",
 "usage": "artifact_content"}]}.
```

Every ID with `mapping_required=true` must resolve inside `task_data/`. Do not create `task_fixtures.json`; the compiler derives internal fixture metadata from this map. The solver instruction must remain unchanged. If a required degree of similarity, scope, relation, or runtime dependency cannot be satisfied, do not fabricate a replacement. Record the unsatisfied specification item in `difficulty.json` and stop construction so that the compiler can abstain.

Prompt for Task Constructor: Evaluator

ROLE AND GOAL

Write an evaluator for the task that scores a new solver outcome by observable completion rather than by reproducing the recorded action sequence.

COMPILER INPUTS (NOT SOLVER CONTEXT)

Fixed solver instruction: `<SOLVER_INSTRUCTION>`

Compiled initial workspace: `<TASK_DATA>`

Task specification and acceptance criteria: `<TASK_SPECIFICATION; ACCEPTANCE_CRITERIA>`

Recorded evidence available only to the compiler: `<FULL_SOURCE_TRAJECTORY; EXPECTED_SOURCE_VERDICT; OPTIONAL_TRUSTED_REFERENCE>`

Evaluation family and reason: `<EXECUTE_CODE | VALIDATE_ARTIFACT | RESPONSE_ONLY; EVALUATION_REASON>`

EVALUATION RULES

1. Derive every scoring criterion from the fixed prompt, an acceptance criterion, an explicit task rule, or a resource available to the solver. The full recorded execution, its verdict assigned by the compiler, and an optional trusted reference may be inspected only as validation evidence available to the compiler. A recorded outcome that solves the Compiled task should be accepted; one that does not solve it should be rejected. Neither may introduce hidden requirements, factual targets, exact wording, a unique tool sequence, or a unique implementation.

2. Evaluate the delivered result, not whether the solver copied the recorded process. Prefer observable output, required state changes, parameter constraints, tool errors, and explicitly forbidden risky actions. Accept equivalent tools, different legal action orders, native file operations, and semantically equivalent implementations. Check order only when the visible task establishes a necessary local dependency.

3. Match the evaluator to the task family. For `execute_code`, obtain the candidate source from the submitted workspace and run a focused test, startup check, or API probe in an isolated directory; regex, keywords, AST shape, or an unused test file are not execution. For `validate_artifact`, parse and inspect the final document, report, data file, or image rather than checking only the script that produced it. For `response_only`, score the answer and do not require a workspace artifact.

4. Use a rubric totaling 100 points. Allocate at least 60 points to correctness for the task and at most 20 points to simple properties such as existence, naming, directory structure, or a response containing content. A missing primary output, unparseable artifact, unexecuted core behavior, or fundamentally incomplete result must remain below the completion threshold of 60.

5. Compute expected values independently from visible inputs and explicit rules. Do not trust an existing label that the task asks the solver to correct. For selection, filtering, classification, bulk update, or partial marking tasks, score both recall and false positives so that applying the same state to every record cannot pass.

6. Use the unchanged initial workspace as the lower bound. An empty response with the unchanged initial state must fail. Cases with a missing or corrupted output must lose the affected core credit; when a task requires leaving some records unchanged, applying the change too broadly must score below the correct outcome.

7. Write one executable scoring example that reaches the substantive scoring path. A binary Office or image artifact must contain valid encoded bytes, not a text placeholder. The scoring example need not pass the task, but it must produce a valid nontrivial score rather than zero or an evaluator exception.

8. The compiler will run the scoring example, the unchanged initial state, negative cases designed for the task, checks that recorded results are excluded, and, when mappings are complete, the recorded outcome. Validation with the recorded outcome is conditional and does not add criteria. If no evaluator can satisfy the acceptance criteria and these checks, report the specification gap and stop rather than inventing a weaker rubric.

EVALUATOR INTERFACE

Write `eval/evaluate.py`, `eval/difficulty.json`, and `eval/fixtures/smoke_result.json`. Implement `evaluate(result, eval_dir)` with the following contract:

```
result: {"messages": [...], "workspace_files": [...]}
return: {"score": s, "max_score": 100,
 "status": "completed", "details": d}.
```

The `details` field records scores for each criterion, observable evidence, and execution diagnostics. The compiler applies the completion threshold of 60 and reserves `no-score` for evaluator failures.