OPENTERMINAL: SCALABLE CONSTRUCTION OF EXECUTABLE TASKS FOR OPEN TERMINAL AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Terminal agents enable diverse real-world workflows, yet open-weight models remain unreliable on realistic terminal tasks. A key bottleneck is the scarcity of executable data. Each task must pair an instruction with a runnable environment, the fixtures it uses, and a verifier that evaluates whether the task is completed. Building such data at scale requires broad task coverage, cross-artifact consistency, and a way to recover from construction failures. OpenTerminal addresses these challenges through corpus-anchored synthesis, shared task contracts, and failure-directed repair. Task concepts are synthesized from 307,293 problem patterns across 36 public corpora, where a cluster-smoothed sampler flattens the sampling distribution over semantic regions and increases coverage of underrepresented areas. A hidden task contract resolves implementation choices left unspecified by the instruction and provides a shared specification for environment and verifier generation. When a validation failure persists, Diagnose–Route–Repair identifies a likely source stage, feeds diagnostic evidence, and regenerates the downstream artifacts, recovering tasks that would otherwise be discarded. The pipeline yields 11,713 executable tasks and 7,365 training trajectories. We fine-tune Qwen3.5-4B and Qwen3.5-9B on these trajectories, improving pass@1 over the base models by 56.8% and 63.1% on Terminal-Bench 2.1, and by 31.3% and 23.9% on OpenThoughts-TBLite. Extensive experiments and analyses confirm the contribution of each component and characterize the data the pipeline produces.

1 INTRODUCTION

Terminal environments provide a unified interface to software, data, system services, and computational tools. Recent agents built on large language models (LLMs) use this interface for multi-step workflows across system administration, data processing, debugging, and scientific computing (Anthropic, 2025; OpenAI, 2025; Google, 2025). However, open-weight models remain unreliable on realistic terminal tasks (Merrill et al., 2026). Each command changes the state observed by subsequent actions, so early errors can propagate through the remaining workflow (Yao et al., 2023; Yang et al., 2024). Beyond instruction–response pairs, training terminal agents requires each task description to come with a runnable environment, the fixtures the task operates on, and a programmatic verifier (Pan et al., 2025; Yang et al., 2025; Jain et al., 2025). Constructing these resources at training scale remains costly.

Scaling executable terminal tasks presents three coupled challenges. First, *diversity* is hard to achieve, because model-generated tasks tend to concentrate on familiar patterns and underrepresent the long tail of terminal workflows. This mirrors the mode-collapse problem observed in synthetic-instruction generation (Wang et al., 2023; Xu et al., 2024; 2025). Second, *cross-artifact consistency* is difficult because the instruction, dependencies, initial state, and verifier must jointly define the same feasible problem. Individually plausible artifacts can still be mutually incompatible. Third, *recovery* is complicated by the gap between where a failure occurs and where it originates. A failure may surface downstream from its source, so retrying at the observation stage can leave the underlying inconsistency untouched. Discarding unresolved instances also wastes prior generation effort and can disproportionately remove difficult-to-construct tasks, narrowing the difficulty distribution of the resulting training data.

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.

Recent pipelines draw on taxonomies, domain specifications, repositories, skill resources, and verified seeds (Gandhi et al., 2026; Zhu et al., 2026; Ivison et al., 2026; Peng et al., 2026; Hua et al., 2026; Lin et al., 2026; Wu et al., 2026; Fan et al., 2026; Li et al., 2026). These sources are complementary, but each constrains coverage to the space represented by its initial resource space. A taxonomy or capability structure is limited by the capabilities it enumerates, a repository collection by the domains represented in its repositories, and a verified seed set by the tasks already covered by its seeds. Sequential conditioning and refinement reduce inconsistencies, but they do not remove the need to check whether the generated artifacts actually agree. Existing failure policies also differ in what they trade. A local retry is cheap but may leave the upstream cause untouched, regenerating the entire instance can restore global consistency but discards artifacts that were already correct, and rejection avoids questionable instances but sacrifices potentially recoverable data.

We introduce OpenTerminal, a framework for constructing executable and verifiable terminal tasks at scale for open terminal agents. It addresses the three coupled challenges through three complementary mechanisms. For diversity, *Corpus-Anchored Task Synthesis* (CATS) anchors task synthesis in heterogeneous technical corpora and uses cluster-smoothed sampling to broaden coverage across their semantic space. For consistency, *Shared Task Contract* (STC) provides a common specification that anchors both environment and verifier generation. By fixing the task semantics before either artifact is generated, STC aligns downstream artifacts rather than relying solely on post-hoc reconciliation. For recovery, *Diagnose–Route–Repair* (DRR) treats a persistent validation failure as a routing signal rather than simply retrying at the observation stage. It predicts a likely source stage, forwards diagnostic feedback, and regenerates that stage and all downstream artifacts.

Our key contributions are summarized as follows:

- **Method.** We introduce CATS, STC, and DRR for diversity-aware task synthesis, contract-based coordination of dependent artifacts, and source-stage-directed recovery, respectively.
- **Data.** From 307,293 problem patterns across 36 public corpora, we generate 50,003 task descriptions, of which 16,384 enter full executable-task construction. This yields 11,713 executable tasks and 7,365 eligible training trajectories.
- **Experiments.** Experiments demonstrate that the constructed trajectories consistently improve downstream terminal-agent performance on Terminal-Bench 2.1 and OpenThoughts-TBLite. Ablations confirm the contribution of each component, and analyses trace coverage, difficulty, and yield from seeds to training data.

2 Related Work

Terminal agents and benchmarks. Evaluation has progressed from execution-grounded coding tasks toward realistic, end-to-end agent workflows. InterCode introduced interactive coding evaluation with execution feedback (Yang et al., 2023), while SWE-bench established repository-level issue resolution as a benchmark for software-engineering agents (Jimenez et al., 2024). Terminal-Bench extends this setting to command-line interaction by evaluating agents on multi-step tasks in containerized environments (Merrill et al., 2026). Terminal-Bench 2.1 revises its 89-task suite to improve task reliability (Terminal-Bench Team, 2026), whereas OpenThoughts-TBLite provides a difficulty-calibrated collection of 100 Terminal-Bench tasks for faster iteration and more informative evaluation across model capability levels (OpenThoughts-Agent team et al., 2026). Harbor provides a common execution framework for these benchmarks by standardizing container provisioning, agent execution, and result aggregation, while its Terminus 2 reference agent supplies a reusable interface between models and task environments (Harbor Framework Team, 2026).

Task and environment synthesis. Recent data-construction pipelines differ primarily in how they obtain task diversity and ground tasks in executable environments. TermiGen, Tmax, LiteCoder-Terminal, and CLI-Universe generate from authored taxonomies, domain specifications, or capability structures, sometimes augmented with retrieval and iterative refinement (Zhu et al., 2026; Ivison et al., 2026; Peng et al., 2026; Hua et al., 2026). These structured starting points provide control over task generation but constrain coverage to capabilities anticipated during their design. Repository-grounded approaches instead derive tasks from Dockerized software projects, gaining realism while inheriting the domain distribution of their source repositories (Lin et al., 2026; Wu et al., 2026). Other methods expand the task space compositionally: SkillSynth samples workflows from a skill

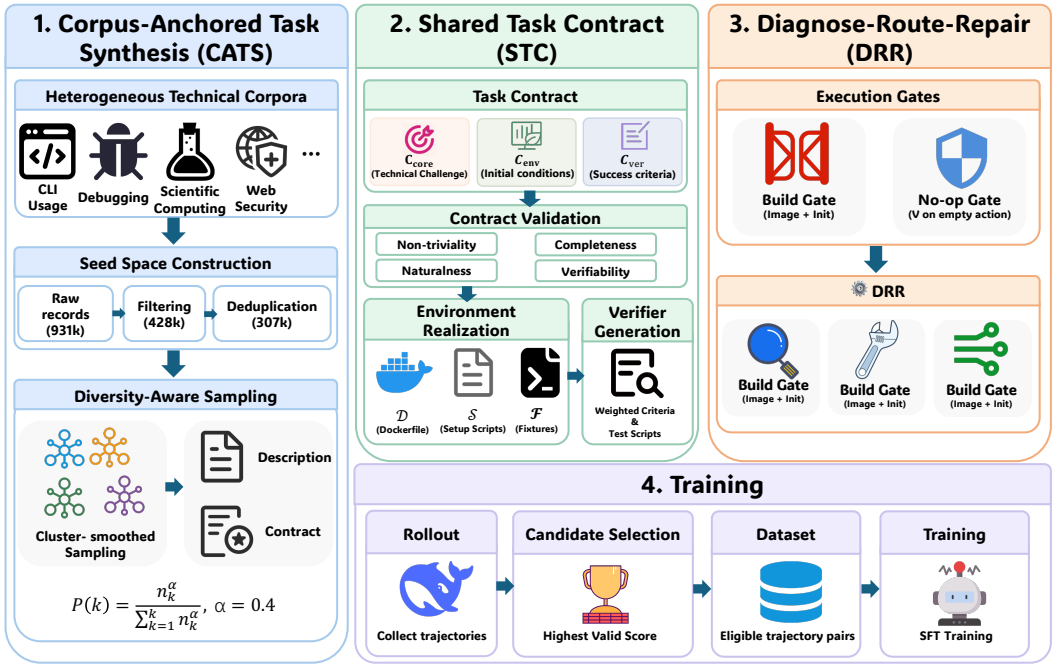

Figure 1: The OpenTerminal pipeline. **(1)** CATS converts heterogeneous technical corpora into a deduplicated seed space and applies cluster-smoothed sampling to generate task descriptions and contracts. **(2)** STC validates each contract and uses it to coordinate environment realization with verifier generation. **(3)** Build and no-op gates validate the resulting artifacts, while DRR diagnoses failures, routes them to a likely source stage, and regenerates the affected dependency suffix. **(4)** Accepted tasks support trajectory collection; the highest-scoring valid trajectories are filtered into the dataset used for SFT.

graph (Fan et al., 2026), while RST recursively extends verified tasks to increase horizon and complexity (Li et al., 2026). Orthogonal work studies how filtering, curricula, and data scale affect terminal-agent training once executable tasks and trajectories are available (Pi et al., 2026).

3 Method

3.1 Overview

OpenTerminal constructs executable terminal tasks $\mathcal{T} = (I, C, E, V)$ from heterogeneous technical corpora, where I is the task instruction, C is a hidden task contract that specifies the intended task semantics and constraints, E is the initialized environment, and V is a programmatic verifier.

As shown in Figure 1, the construction follows the dependency structure $I \to C$, $(I, C) \to E$, and $(I, C, E) \to V$. CATS (§3.2) synthesizes diverse task descriptions from corpus-derived problem patterns. STC (§3.3) then derives a shared contract that conditions environment and verifier generation. When validation fails, DRR (§3.4) diagnoses a likely source stage and regenerates that stage together with all downstream artifacts.

3.2 Corpus-Anchored Task Synthesis

Corpus-derived seed space. We construct the seed space from heterogeneous public technical corpora to expose the synthesis process to a broad range of terminal-relevant workflows. The pool draws on 36 sources covering command-line use, system administration, software engineering, debugging, scientific computing, and tool use, listed in full in Appendix A.1. We normalize each record with a source-specific loader and extract a transferable *problem pattern*. Each pattern records the underlying technical objective, skill and domain labels, and a short source excerpt for later synthesis.

A relevance filter then discards patterns that cannot be instantiated as terminal tasks, either because they require no meaningful execution or because they depend on resources unavailable inside the target container. Starting from 931,964 raw records, abstraction and relevance filtering yield 428,157 terminal-relevant patterns, and deduplication at cosine similarity 0.90 (Reimers & Gurevych, 2019) leaves 307,293 seeds.

Diversity-aware sampling. The seed pool is highly imbalanced, so sampling seeds uniformly would let a few dense semantic regions dominate the resulting tasks. We therefore smooth the sampling distribution over semantic regions. We first organize seeds into local semantic neighborhoods and then aggregate their centroids into $K = 100$ macro-clusters. Let n_k denote the size of cluster k. A single cluster is drawn with probability

$$p(k) = \frac{n_k^\alpha}{\sum_{k=1}^{K} n_k^\alpha}, \qquad \alpha = 0.4, \tag{1}$$

and clusters are drawn sequentially without replacement, with the probabilities of the remaining clusters renormalized after each selection. The sublinear exponent preserves the frequency ordering of the source distribution while flattening over-represented regions, so a sampled group draws from several semantic regions instead of one. Candidate groups are sampled from Eq. 1 and fingerprinted before synthesis to avoid repeated combinations. Appendix A.2 gives the clustering and group-construction details, and Appendix A.3 reports how the domain mix shifts between seeds and tasks.

Coherent task composition. Diversity-oriented sampling alone may combine seeds that do not form a coherent terminal workflow, so we separate retrieval diversity from task coherence. Given a sampled candidate group, the synthesis teacher examines the problem patterns and source excerpts, selects a semantically compatible subset, and distills it into a compact task concept. If no coherent multi-seed composition is possible, it falls back to a single high-value seed. This design broadens the semantic regions from which tasks are drawn without forcing unrelated skills into one task.

3.3 From Contract to Executable Task

Contract fields and acceptance. An instruction alone leaves room for different interpretations of the intended task, which can cause downstream artifacts to encode inconsistent problem specifications. Given an instruction I and its selected seed context, OpenTerminal derives a hidden task contract $C = (c_{\text{core}}, c_{\text{env}}, c_{\text{ver}})$. The fields describe the technical challenge, required initial conditions, and observable success criteria. A model-based judge evaluates non-triviality, naturalness, downstream completeness, and objective verifiability. Any failed judgment triggers a targeted revision for at most two rounds. Revising I also regenerates C, whereas a contract-only revision leaves I unchanged. Once accepted, (I, C) is fixed for the remainder of construction, and C is kept hidden from the agent that executes the task.

Environment realization. Given (I, C), we construct the environment $E = (D, F, S)$, consisting of a Dockerfile D for dependencies, a manifest F for static fixtures, and a setup script S for initializing dynamic state. Because realistic tasks may require many task-specific files, we first generate a manifest that specifies each required file path and its purpose, and then generate the file contents individually from this manifest. The setup script initializes the remaining environment state, including permissions, symbolic links, binary artifacts, and services. For tasks that require privileged operations or specialized services that are difficult to realize in a container, we use task-specific shims or local mocks to provide the required functionality. The resulting environment exposes the intended challenge without preinstalling solutions or answer keys.

Environment-grounded verification. The verifier is generated from (I, C, E), so its checks are grounded in both the intended task semantics and the concrete realized environment. The contract specifies the intended outcome, while E provides the paths, dependencies, and input data against which success can be evaluated. The verifier defines weighted criteria $\{(q_i, w_i)\}_{i=1}^{m}$ and private test scripts. Each case reports a criterion identifier, a Boolean outcome c_{ij}, and a diagnostic message. For the schema-valid cases $\mathcal{C}_i$ assigned to criterion i, we compute

$$s_i = \begin{cases} |\mathcal{C}_i|^{-1} \sum_{j \in \mathcal{C}_i} c_{ij}, & |\mathcal{C}_i| > 0, \\ 0, & |\mathcal{C}_i| = 0, \end{cases} \qquad r = \frac{\sum_{i=1}^{m} w_i s_i}{\sum_{i=1}^{m} w_i} \in [0, 1], \tag{2}$$

with $w_i \geq 0$ and $\sum_i w_i > 0$. The resulting score r provides a measure of task completion rather than a single pass/fail outcome, which also supplies finer-grained evidence for failure diagnosis.

3.4 DIAGNOSE–ROUTE–REPAIR

Construction is validated by two execution gates. The build gate checks image construction and initialization. The no-op gate then runs V on the initialized state with empty agent actions, and requires a well-formed zero score. A failing gate returns build errors or case-level evidence, which forms the input to DRR.

For a construction failure f, we distinguish two stages. The *observation stage* $s_{\mathrm{obs}}(f)$ is where the failure becomes detectable. The *source stage* $s_{\mathrm{src}}(f)$ is where the underlying inconsistency was introduced. The source stage is not directly observable, and the two need not coincide. For example, a positive no-op reward may come from an overly permissive verifier, or from an environment that already contains the requested deliverable. Repairing only the artifact at the observation stage would therefore leave the underlying inconsistency unchanged.

Local repair before diagnosis. DRR first exhausts a bounded local repair budget at the observation stage before invoking cross-stage diagnosis. Construction follows the dependency order ENV $\rightarrow$ VERIFIER $\rightarrow$ BUILD $\rightarrow$ NOOP, where ENV and VERIFIER generate E and V, and the latter two stages execute validation checks. If local repair fails, DRR invokes cross-stage diagnosis.

Source-stage diagnosis and rerouting. Given the current failure report e_t, instruction I, contract C, and gate-specific artifact evidence A_t, DRR predicts

$$(s_{\mathrm{src}}, d_t) = \mathrm{Diagnose}(g_t, e_t, I, C, A_t), \tag{3}$$

where s_{src} is a predicted source stage and d_t contains diagnostic feedback for repair. Build failures route to ENV, while no-op failures may route to ENV, VERIFIER, or BUILD. Diagnosis may also return UNFIXABLE when the failure cannot be reliably repaired within the available budget.

Rerouting retains all stages before the selected entry point and regenerates the dependency suffix from that point onward. Thus, routing to ENV regenerates both the environment and the downstream verifier, while routing to BUILD reconstructs and reinitializes the current environment without regenerating its source artifacts. The failure report and diagnostic feedback are passed to the restarted generator so that subsequent construction is conditioned on the observed failure. We allow at most four construction passes, including the initial attempt. This dependency-aware rerouting retains recoverable task instances that would otherwise be discarded after a failed construction attempt. Algorithm 1 summarizes the control flow, and Appendix B reports the gate and routing statistics.

3.5 FROM EXECUTABLE TASKS TO TRAINING DATA

For each executable task, we collect rollout trajectories through the terminal-agent interface and retain scores and case-level diagnostics, distinguishing invalid or missing results from genuine zero scores. The highest-scoring valid trial becomes a *candidate* trajectory.

Candidate selection is separate from training-data acceptance. We fix the acceptance rule before any benchmark evaluation, and $\mathcal{D}_{\mathrm{SFT}}$ then contains the eligible instruction–trajectory pairs. The objective is

$$\mathcal{L}_{\mathrm{SFT}}(\theta) = -\mathbb{E}_{(I,\tau)\sim\mathcal{D}_{\mathrm{SFT}}} \sum_{t\in\mathcal{M}(\tau)} \log p_\theta(x_t \mid I, x_{<t}), \tag{4}$$

where $\mathcal{M}(\tau)$ selects assistant response tokens, including tool calls, while instructions and tool observations provide context without supervision. The verifier score is used only to select training trajectories. It does not define a policy-gradient objective. A recovered task contributes to training only if it also yields an eligible trajectory, so we report recovery both before and after trajectory filtering.

Algorithm 1 Diagnose–Route–Repair with cross-stage rerouting

Require: Instruction I; contract C; ordered stages $\mathcal{S} = \langle \textsc{Env}, \textsc{Verifier}, \textsc{Build}, \textsc{NoOp} \rangle$; local budgets L; pass budget R
Ensure: Executable task $\mathcal{T}$, or REJECT

$s_{\text{entry}} \leftarrow \textsc{Env}$; $\mathcal{T} \leftarrow \emptyset$; $d \leftarrow \emptyset$
for $r = 1$ **to** R **do**
  $\mathcal{T} \leftarrow \textsc{MarkStaleFrom}(\mathcal{T}, s_{\text{entry}})$
  $(\mathcal{T}, g, e) \leftarrow \textsc{Advance}(\mathcal{S}, s_{\text{entry}}, I, C, \mathcal{T}, d, L)$ {execute stages in order with local repair; stop at the first unresolved failure}
  if $g = \textsc{None}$ **then**
    return $\mathcal{T}$
  end if
  if $r = R$ **or** $g \notin \{\textsc{Build}, \textsc{NoOp}\}$ **then**
    return REJECT
  end if
  $A \leftarrow \textsc{DiagnosticContext}(g, \mathcal{T})$
  $(\text{source}, d) \leftarrow \textsc{Diagnose}(g, e, I, C, A)$
  if source $= \textsc{Unfixable}$ **then**
    return REJECT
  end if
  $s_{\text{entry}} \leftarrow \textsc{Stage}(\text{source})$
end for
return REJECT

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Benchmarks. We evaluate on Terminal-Bench 2.1 (Terminal-Bench Team, 2026; Merrill et al., 2026) and OpenThoughts-TBLite (OpenThoughts-Agent team et al., 2026). Terminal-Bench 2.1 comprises 89 complex, end-to-end tasks executed in container environments, covering a broad range of real-world terminal workflows. OpenThoughts-TBLite is a difficulty-calibrated 100-task subset of Terminal-Bench designed to provide finer-grained evaluation across model capability levels. We report mean pass@1 over five independent trials per task, evaluated with Harbor using the Terminus 2 agent (Harbor Framework Team, 2026).

Dataset. We construct environments for a frozen subset of 16,384 task descriptions drawn from the 50,003 accepted descriptions, resulting in 11,713 executable tasks. For each executable task, we collect three rollouts and retain the highest-scoring valid trajectory, with trajectories scoring below 0.5 discarded, yielding 7,365 trajectories for training. We then apply MinHash LSH (Broder, 1997; Lee et al., 2022) to the task instructions of the training trajectories and remove instances that overlap with either evaluation benchmark.

Training. We fine-tune Qwen3.5-4B and Qwen3.5-9B (Qwen Team, 2026) on the resulting 7,365 trajectories. Both task synthesis and rollout use DeepSeek-V4-Flash-0731. We perform full-parameter SFT in LLaMA-Factory (Zheng et al., 2024) for three epochs with a learning rate of 1×10^{-5}, cosine decay, 3% warmup, and an effective batch size of 64.

Baselines. We compare against four concurrent terminal-data construction pipelines: Endless Terminals (Gandhi et al., 2026), SkillSynth (Fan et al., 2026), LiteCoder-Terminal (Peng et al., 2026), and CLI-Universe (Hua et al., 2026). Every trajectory set is trained on the same two Qwen3.5 backbones, so the comparison isolates differences in the training data.

4.2 MAIN RESULTS

Table 1 shows that training on OpenTerminal consistently improves over the base models. With Qwen3.5-4B, OpenTerminal achieves 21.8 on Terminal-Bench 2.1 and 43.6 on TBLite, correspond-

Table 1: Main results on Terminal-Bench 2.1 and OpenThoughts-TBLite under Harbor with the Terminus 2 agent. Best in each group is **bolded** and second best is underlined.

Method	Trajectories	Terminal-Bench 2.1	OpenThoughts-TBLite
Qwen3.5-4B			
Base	—	$13.9_{\pm 2.6}$	$33.2_{\pm 4.1}$
Endless Terminals (Gandhi et al., 2026)	3,255	$16.2_{\pm 3.4}$	$37.8_{\pm 2.3}$
SkillSynth (Fan et al., 2026)	3,560	$18.4_{\pm 1.9}$	$35.6_{\pm 4.4}$
LiteCoder-Terminal (Peng et al., 2026)	11,255	$17.1_{\pm 2.8}$	$\underline{40.4}_{\pm 3.1}$
CLI-Universe (Hua et al., 2026)	6,000	$\underline{19.1}_{\pm 2.2}$	$38.4_{\pm 1.6}$
OpenTerminal	7,365	$\mathbf{21.8}_{\pm 3.0}$	$\mathbf{43.6}_{\pm 2.7}$
Qwen3.5-9B			
Base	—	$15.7_{\pm 3.3}$	$42.6_{\pm 2.4}$
Endless Terminals (Gandhi et al., 2026)	3,255	$18.9_{\pm 2.0}$	$48.0_{\pm 3.8}$
SkillSynth (Fan et al., 2026)	3,560	$22.0_{\pm 3.6}$	$45.4_{\pm 2.1}$
LiteCoder-Terminal (Peng et al., 2026)	11,255	$19.6_{\pm 1.5}$	$\underline{50.4}_{\pm 2.9}$
CLI-Universe (Hua et al., 2026)	6,000	$\underline{22.9}_{\pm 2.8}$	$48.4_{\pm 4.2}$
OpenTerminal	7,365	$\mathbf{25.6}_{\pm 2.3}$	$\mathbf{52.8}_{\pm 1.8}$

ing to relative gains of 56.8% and 31.3%, respectively. OpenTerminal also outperforms the concurrent datasets on both benchmarks. On Terminal-Bench 2.1, CLI-Universe is the strongest concurrent dataset at 19.1, while OpenTerminal reaches 21.8, a 2.7-point improvement. On TBLite, LiteCoder-Terminal is strongest at 40.4, and OpenTerminal improves this to 43.6, a 3.2-point gain.

These gains persist at the 9B scale. With Qwen3.5-9B, OpenTerminal achieves 25.6 on Terminal-Bench 2.1 and 52.8 on TBLite, yielding relative gains of 63.1% and 23.9% over the corresponding base models. It exceeds the strongest concurrent dataset by 2.7 points on Terminal-Bench 2.1 and 2.4 points on TBLite. The results also indicate that the data contribution can partially offset model-scale differences. On Terminal-Bench 2.1, Qwen3.5-4B trained on OpenTerminal reaches 21.8, compared with 15.7 for the 9B base model. On TBLite, the same comparison gives 43.6 versus 42.6.

4.3 Ablation Studies

As shown in Table 2, we ablate each of the three mechanisms underlying our data construction pipeline. CATS controls coverage by determining which regions of the seed space contribute to task synthesis. STC promotes cross-artifact consistency by fixing a shared interpretation before environment and verifier generation. DRR recovers construction yield by repairing tasks that would otherwise be discarded after failed construction.

Table 2: OpenTerminal ablations. CATS is evaluated by coverage, STC by cross-artifact consistency, and DRR by construction yield. Entries marked — were not measured for that variant.

Component	Method	Coverage		Consistency	Yield	
		$H/\log K \uparrow$	Tail-50 ↑	Mismatch ↓	Task ↑	Traj. ↑
Overall	**OpenTerminal**	0.927	26.9	4.5	71.5	45.0
CATS	w/o corpus anchoring	0.548	3.2	—	—	—
	w/o cluster smoothing	0.739	7.8	—	—	—
STC	w/o shared contract	—	—	49.5	—	—
DRR	Local-only repair	—	—	—	65.2	41.3
	Reject-on-failure	—	—	—	57.6	37.4

$H/\log K$: entropy over K=100 seed clusters, normalized by $\log K$. Tail-50: mass on the 50 rarest clusters, fixed by the seed distribution. Mismatch: disagreement among the instruction, environment, and verifier. Task / Traj.: gate-pass and $r \geq 0.5$ rates over the 16,384 construction attempts.

CATS: corpus sampling. Without corpus anchoring, normalized entropy falls from 0.927 to 0.548 and tail-50 mass from 26.9% to 3.2%. Directly prompting the synthesis model for task intents concentrates the resulting tasks in a small number of dense semantic regions. Without cluster smoothing,

rare clusters remain underrepresented, with entropy dropping to 0.739 and tail-50 mass to 7.8%. The two components therefore serve complementary roles. Corpus anchoring defines a broad seed space, while cluster smoothing redistributes sampling mass across that space.

STC: shared conditioning. Without the shared contract, both the environment and verifier generators condition only on the instruction and independently resolve implementation choices that the instruction leaves unspecified. Rebuilding the same task descriptions under this setting raises the mismatch rate among the instruction, environment, and verifier from 4.5% to 49.5%. Figure 2 reports the mismatch type composition. Missing fixtures are the dominant type in both settings, while pre-solved state remains relatively uncommon, at 9 tasks with the shared contract and 52 without it. The large increase in mismatch rate indicates that shared conditioning is important for keeping independently generated artifacts aligned with the same task specification.

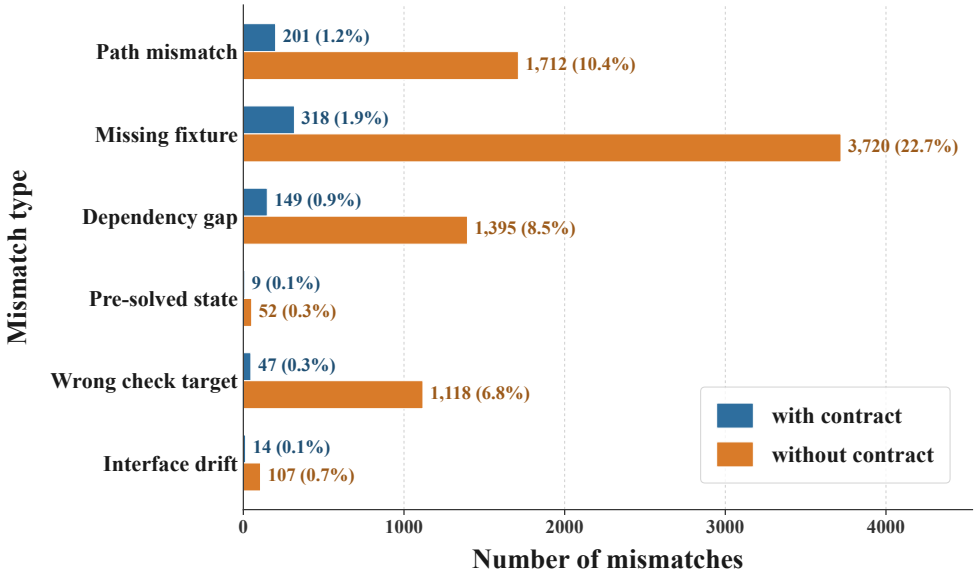

Figure 2: Dominant mismatch type with and without the shared task contract at generation time.

DRR: construction recovery. Of the 16,384 construction attempts, 9,430 tasks pass on the first attempt, 1,249 are recovered by local verifier repair, and a further 1,034 are recovered through cross-stage routing. Without repair, a reject-on-failure policy retains only 57.6% of attempts, compared with 71.5% with DRR. These recovered tasks also increase the amount of usable training data, from 6,126 first-pass trajectories to 6,761 with local repair and 7,365 with DRR. Cross-stage routing therefore recovers an additional 6.3 percentage points of task yield and 3.7 percentage points of trajectory yield beyond local repair alone. Together, these results show that treating construction failures as routing signals allows recoverable instances to be retained rather than discarded.

4.4 Analysis

Difficulty analysis. As shown in Figure 3(a), 31.6% of the verifier scores on executable tasks are 0 and 18.1% are 1, with the remaining scores lying distributed broadly across the interval. The median trajectory takes 12 turns, and 75% of trajectories finish within 15 turns, while a small tail extends substantially longer. Those longer runs are the harder tasks. This distribution indicates substantial variation in both task completion and interaction length, with trajectories ranging from immediate failure to partial and successful completion.

Recovered-task quality. Figure 3(b) compares the quality of tasks that pass on the first attempt with those recovered by local repair and cross-stage routing. Both the mean best score and the fraction of tasks reaching $r \geq 0.8$ decrease progressively across the three groups. First-pass tasks achieve a mean best score of 0.59, with 37.7% reaching $r \geq 0.8$ within the first three trials. Local

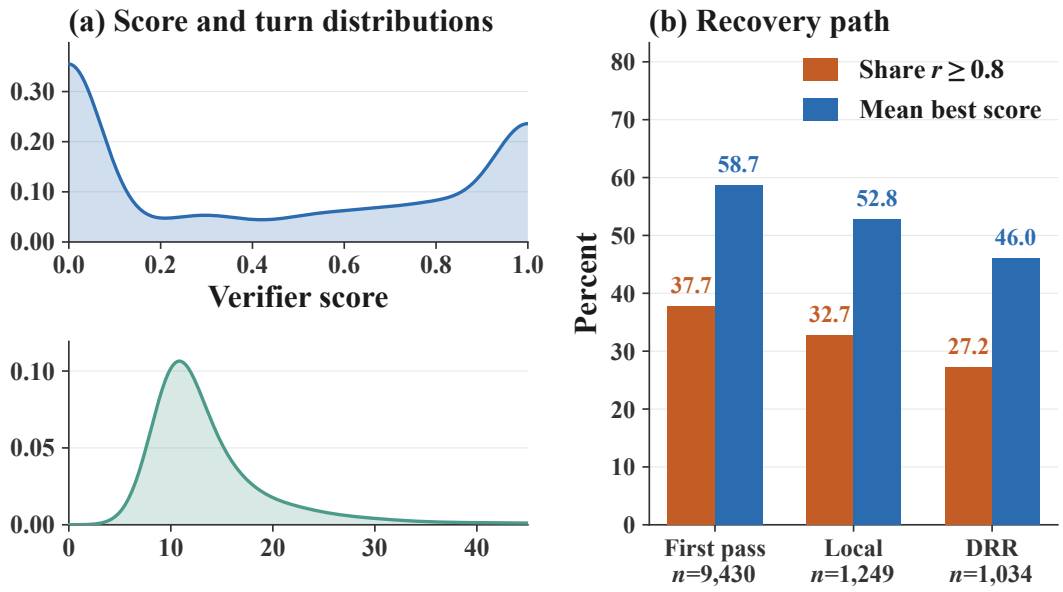

Figure 3: Trajectory scores and recovered task quality. **(a)** Distributions of verifier scores and agent turns over the trajectories of executable tasks. **(b)** Fraction of trajectory scores reaching $r \geq 0.8$ within the three trials and mean best score for first-pass, locally recovered, and DRR-recovered tasks.

recovery lowers these values to 0.53 and 32.7%, while cross-stage recovery lowers them further to 0.46 and 27.2%. The progressive decline suggests that recovered tasks are systematically more challenging than tasks that pass without repair, with DRR retaining the most difficult group. Verifier-gate defect classes and the full routing ledger are reported in Appendix B.

5 Conclusion

OpenTerminal argues that the scarcity of executable terminal data stems from the difficulty of constructing complete, internally consistent tasks rather than from generating plausible task text alone. A usable task requires diverse and meaningful task sources, agreement among its instruction, environment, and verifier, and recovery when construction fails. Our construction pipeline addresses these requirements through corpus-anchored synthesis, shared task contracts, and failure-directed recovery, with each mechanism targeting a distinct bottleneck in the construction process. By separating accepted task descriptions, executable tasks, and training trajectories, we make the attrition at each stage explicit. Experiments show that the resulting data consistently improves terminal-agent performance across model scales and benchmarks, while the ablations confirm that the three mechanisms improve coverage, cross-artifact consistency, and construction yield, respectively.

Limitations. Our construction pipeline relies on LLMs to generate task specifications, environments, verifiers, and repair decisions, so systematic errors in the underlying model may propagate across multiple artifacts. Current corpus and construction budget also cover only a subset of the possible terminal task space, and the resulting distribution may still underrepresent rare or specialized workflows. Finally, our evaluation focuses on SFT and does not establish whether the constructed tasks provide comparable benefits for other training paradigms or for long-horizon agent learning.

AI use statement

AI tools were used for literature retrieval and discovery, including identifying related work and relevant research resources. AI tools were also used to support research ideation, including exploring research directions and refining methodological ideas. Specific uses and their roles are described in the paper.

REFERENCES

Anthropic. Best practices for Claude Code. https://www.anthropic.com/engineering/claude-code-best-practices, 2025.

Andrei Z. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences (SEQUENCES)*. IEEE, 1997. doi: 10.1109/SEQUEN.1997.666900.

Zhiyuan Fan, Tinghao Yu, Yuanjun Cai, Jiangtao Guan, Yun Yang, Dingxin Hu, Jiang Zhou, Xing Wu, Zhuo Han, Feng Zhang, and Lilin Wang. Toward scalable terminal task synthesis via skill graphs. *arXiv preprint arXiv:2604.25727*, 2026.

Kanishk Gandhi, Shivam Garg, Noah D. Goodman, and Dimitris Papailiopoulos. Endless terminals: Scaling RL environments for terminal agents. *arXiv preprint arXiv:2601.16443*, 2026.

Google. Gemini CLI: Your open-source AI agent. https://blog.google/innovation-and-ai/technology/developers-tools/introducing-gemini-cli-open-source-ai-agent/, 2025.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments. https://github.com/harbor-framework/harbor, 2026.

Zhanbo Hua, Yifan Yao, Weihao Xie, Yongchi Zhao, Minghao Liu, Ruizhi Qiu, Zhewei Huang, Zun Wang, Yiyan Ji, Yunhai Ye, Letian Zhu, Xinping Lei, Han Li, Zhiyuan Ma, Zili Wang, Zhaoxiang Zhang, and Jiaheng Liu. CLI-universe: Towards verifiable task synthesis engine for terminal agents. *arXiv preprint arXiv:2606.22883*, 2026.

Hamish Ivison, Junjie Oscar Yin, Rulin Shao, Teng Xiao, Nathan Lambert, and Hannaneh Hajishirzi. Tmax: A simple recipe for terminal agents. *arXiv preprint arXiv:2606.23321*, 2026.

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. *Conference on Language Modeling (COLM)*, 2025.

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? *International Conference on Learning Representations (ICLR)*, 2024.

Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. Deduplicating training data makes language models better. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 8424–8445, 2022.

Zhongzhi Li, Yucheng Shi, Zongxia Li, Ruhan Wang, Anhao Li, Zixun Huang, Junyao Yang, Lei Ke, Ninghao Liu, Haitao Mi, and Leowei Liang. Recursive synthesis for long-horizon terminal tasks. *arXiv preprint arXiv:2608.05466*, 2026.

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. *arXiv preprint arXiv:2602.10999*, 2026.

Mike A Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Kumar Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Kwesi Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai,

Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Jenia Jitsev, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

OpenAI. Introducing codex. https://openai.com/index/introducing-codex/, 2025.

OpenThoughts-Agent team, Snorkel AI, and Bespoke Labs. OpenThoughts-TBLite: A High-Signal benchmark for iterating on terminal agents. https://www.openthoughts.ai/blog/openthoughts-tblite, February 2026. Code and data: https://github.com/open-thoughts/OpenThoughts-TBLite.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-gym. *International Conference on Machine Learning (ICML)*, 2025.

Xiaoxuan Peng, Kaiqi Zhang, Xinyu Lu, Boxi Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, and Le Sun. Litecoder-terminal: Scaling long-horizon terminal environments for learning language agents. *arXiv preprint arXiv:2605.29559*, 2026.

Renjie Pi, Grace Lam, Mohammad Shoeybi, Pooya Jannaty, Bryan Catanzaro, and Wei Ping. On data engineering for scaling LLM terminal capabilities. *arXiv preprint arXiv:2602.21193*, 2026.

Qwen Team. Qwen3.5: Towards native multimodal agents. https://qwen.ai/blog?id=qwen3.5, February 2026.

Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using siamese BERT-networks. *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2019.

D. Sculley. Web-scale k-means clustering. In *Proceedings of the 19th International Conference on World Wide Web (WWW)*, pp. 1177–1178, 2010. doi: 10.1145/1772690.1772862.

Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. MPNet: Masked and permuted pre-training for language understanding. *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Terminal-Bench Team. Terminal-Bench 2.1. https://www.tbench.ai/news/terminal-bench-2-1, 2026.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2023.

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. *International Conference on Machine Learning (ICML)*, 2026. Spotlight.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. Wizardlm: Empowering large pre-trained language models to follow complex instructions. *International Conference on Learning Representations (ICLR)*, 2024.

Zhangchen Xu, Fengqing Jiang, Luyao Niu, Yuntian Deng, Radha Poovendran, Yejin Choi, and Bill Yuchen Lin. Magpie: Alignment data synthesis from scratch by prompting aligned LLMs with nothing. *International Conference on Learning Representations (ICLR)*, 2025.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

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. *Advances in Neural Information Processing Systems (NeurIPS)*, 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. *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. *International Conference on Learning Representations (ICLR)*, 2023.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL), System Demonstrations*, 2024.

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. *arXiv preprint arXiv:2602.07274*, 2026.

Table 3: Data-construction funnel and denominators. Task-description generation uses all 50,382 candidate-group attempts, whereas construction uses the first 16,384 records in the pipeline-results stream. Rollout rows use the 11,713 executable tasks as their denominator and are overlapping summary statistics. The first-judgment rate (†) is estimated from 682 logged generation attempts. Verifier diagnostics are reported in Table 8.

Phase	Stage	Count	Rate
Seed curation	raw records	931,964	—
	terminal-relevant patterns	428,157	45.9%
	seed pool after dedup	307,293	71.8%
Task descriptions	candidate groups sampled	50,382	—
	distinct groups	50,381	—
	estimated first-judgment acceptance†	—	91.1%
	accepted after up to two rewrites	50,003	99.25%
Construction freeze	construction attempts	16,384	—
	executable tasks	11,713	71.5%
	eligible training trajectories ($r \geq 0.5$)	7,365	45.0%
Rollout	best reward > 0	9,178	78.4%
	best reward in $[0.2, 0.8)$	4,249	36.3%

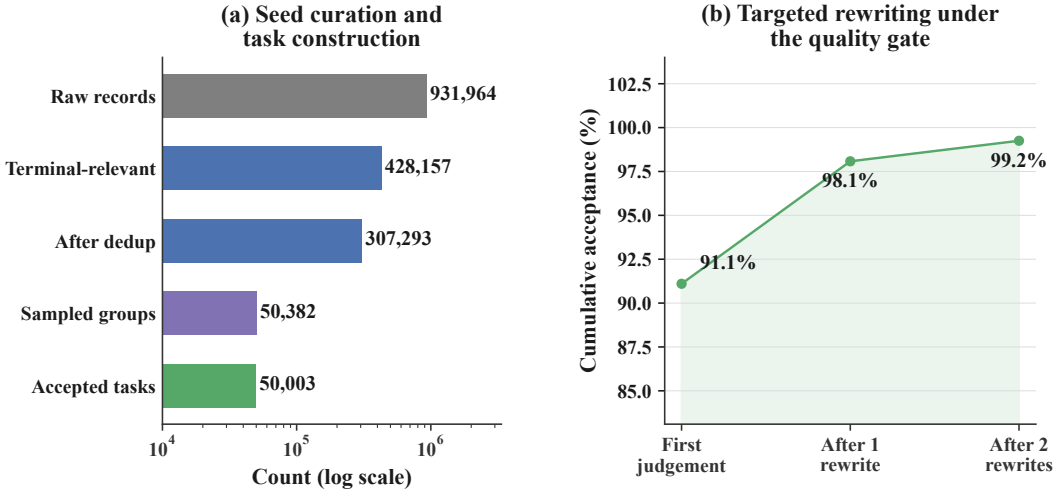

Figure 4: Task-description generation. **(a)** The pipeline reduces 931,964 raw records to 307,293 deduplicated seeds and produces 50,003 accepted task descriptions from 50,382 candidate groups, of which 50,381 are distinct. **(b)** Estimated cumulative acceptance across up to two revision rounds, based on 682 logged generation attempts. The final acceptance rate over all 50,382 candidate-group attempts is 99.25%.

A Seed Corpus and Task Descriptions

This appendix reports the corpus composition, task-description generation, clustering, and domain statistics omitted from the main text. Figure 4 summarizes seed curation and task-description acceptance, while Table 3 separates the denominators used for description generation, construction, and rollout statistics. The snapshot contains 50,003 accepted task descriptions, of which the frozen 16,384-instance subset enters executable-task construction.

A.1 Seed Corpus Composition

Table 4 lists all 36 public sources used to construct the seed pool. The distribution is uneven, with two sources accounting for roughly 70% of all patterns. We retain this distribution because it reflects the availability of public technical material: sources describing routine shell work are substantially

Table 4: Source composition of the deduplicated seed pool and selected seed slots in accepted task descriptions. *Seed share* is the fraction of the 307,293-pattern pool contributed by each source; *task share* is the fraction of selected seed slots originating from that source. All 36 sources are listed. The pool is not rebalanced before sampling.

Source dataset	Seeds	Seed share	Task share
code-exercises	130,642	42.51%	26.85%
stackoverflow-python	83,871	27.29%	35.98%
commitpackft-shell	28,418	9.25%	10.17%
SWE-bench	20,206	6.58%	7.07%
cli-commands-explained	13,395	4.36%	8.17%
NL2Bash	7,165	2.33%	4.22%
tldr	5,151	1.68%	2.33%
glaive-function-calling	4,374	1.42%	1.52%
commitpackft-go	2,622	0.85%	0.54%
commitpackft-cpp	2,517	0.82%	0.47%
commitpackft-java	1,857	0.60%	0.27%
commitpackft-rust	1,707	0.56%	0.41%
CLI-Gym	1,277	0.42%	0.32%
awesome-selfhosted	955	0.31%	0.53%
CTFs-writeups	541	0.18%	0.14%
bash-commands-dataset	439	0.14%	0.25%
TheAlgorithms-Go	431	0.14%	0.08%
TheAlgorithms-Rust	390	0.13%	0.09%
SWE-bench-Verified	298	0.10%	0.16%
devops-exercises	192	0.06%	0.05%
numerical-methods	169	0.06%	0.09%
intercode-nl2bash	134	0.04%	0.08%
mlagentbench	105	0.03%	0.04%
js-algorithms	100	0.03%	0.03%
unix-commands	73	0.02%	0.04%
ToolACE	58	0.02%	0.03%
cryptopals	57	0.02%	0.03%
Agent-FLAN-react	38	0.01%	0.01%
Linux-Bash-Commands	35	0.01%	0.03%
cmake-examples	26	0.0085%	0.0031%
bash-guide	15	0.0049%	0.0019%
awesome-ctf	13	0.0042%	0.0050%
Agent-FLAN-tflan	11	0.0036%	0.0012%
qemu-starter	6	0.0020%	0.0006%
tmux-automation	3	0.0010%	0.0031%
excel-automation	2	0.0007%	0.0019%
total	**307,293**	**100%**	**100%**

smaller than collections of programming exercises and question–answer pairs, and balancing the corpus would discard usable records without expanding its coverage. The task distribution does not inherit this source composition directly. Smoothed sampling and synthesis-teacher selection reshape which seeds enter generated tasks, as quantified by the cluster analysis in §A.2 and the domain analysis in §A.3.

A.2 Clustering Procedure and Its Effect on the Distribution

We embed each abstract pattern with `all-mpnet-base-v2` (Reimers & Gurevych, 2019; Song et al., 2020), producing a 768-dimensional, ℓ_2-normalized representation. Source excerpts are retained for later task synthesis but are excluded from deduplication. An incremental pass removes pairs with cosine similarity above 0.90. Clustering then proceeds in two stages. MiniBatch k-means (Sculley, 2010) first partitions the $N = 307{,}293$ seeds into $K_{\text{micro}} = \min(20K, N/5) = 2{,}000$ micro-clusters. Agglomerative clustering with cosine distance and average linkage subsequently merges their centroids into $K = 100$ macro-clusters. The synthesis teacher labels each

Table 5: Diversity statistics at the cluster, task-composition, and instruction levels. Cluster mass on the task side is computed over selected seed slots; instruction statistics cover all 50,003 accepted task descriptions.

	Statistic	Seed pool	Generated tasks
cluster mass	macro-clusters covered	100	100
	normalized entropy $H/\log K$	0.739	**0.927**
	mass in 5 largest clusters	51.7%	**22.7%**
	mass in 10 largest	61.7%	29.2%
	mass in 20 largest	75.1%	43.5%
	mass in 50 smallest	7.8%	**26.9%**
compos.	tasks using 1 seed	—	8
	tasks using 2–3 seeds	—	37,820
	tasks using $\geq$4 seeds	—	12,175
	tasks spanning $\geq$2 clusters	—	18,471 (36.9%)
instruction text	mean length (tokens)	—	213
	vocabulary size	—	166,126
	distinct-2	—	0.152
	distinct-4	—	0.660
	mean pairwise 4-gram overlap	—	**0.0010**
	instructions naming a concrete tool	—	**97.0%**

macro-cluster from a sample of at most 20 members. Cluster sizes range from 100 to 62,500 seeds; the first quartile, median, and third quartile are 460, 1,104, and 2,558.

During task synthesis, each candidate group contains eight seeds sampled under the smoothed cluster distribution in Equation 1 with $\alpha = 0.4$. A group draws from at most five clusters and uses at most three seeds per cluster; if these constraints leave unfilled positions, the remaining positions are sampled uniformly from the residual seed pool. Candidate-group fingerprints prevent repeated combinations.

Table 5 reports diversity at the cluster, task-composition, and instruction levels, while Table 6 lists all 100 macro-clusters. The generated-task distribution is substantially flatter than the seed distribution without becoming uniform: normalized entropy increases from 0.739 to 0.927, while the 50 smallest clusters under the seed distribution account for 7.8% of seed mass and 26.9% of selected-seed mass in generated tasks. Task composition remains compact. The synthesis teacher selects two or three seeds for 75.6% of tasks; 63.1% of tasks draw all selected seeds from one macro-cluster, while 36.9% span multiple clusters. Thus, the sampler broadens the regions of the corpus used for synthesis without requiring every task to combine seeds from multiple macro-clusters. Across accepted instructions, the mean 4-gram overlap between random pairs is 0.001, and 97.0% name a concrete tool. Figure 5 visualizes the corresponding shift in cluster mass.

A.3 Domain Distribution

Each seed receives a free-form domain phrase during abstraction, which we map to one of 44 coarse categories using an ordered keyword rule set. A task is assigned to the union of the categories associated with its selected seeds, so task shares need not sum to 100%. Table 7 and Figure 6 compare seed-side and task-side shares.

Because each task aggregates several seeds, task-to-seed share ratios reflect both sampling and multi-seed composition. Under this measure, the ratios are $3.8\times$ for system administration, $3.7\times$ for file-system work, $3.6\times$ for networking, $3.2\times$ for databases, and $3.1\times$ for automation and scripting. Generic coding exercises, the largest seed-side category at 12.5%, have a smaller ratio of $1.3\times$. We therefore interpret the contrast between terminal-native categories and generic programming categories rather than attributing each amplification ratio to sampling alone.

Table 6: All 100 seed-pool macro-clusters, ordered by size and labeled by the synthesis teacher from a sample of members. Labels are truncated to fit the column width. *Task share* is the percentage of selected seed slots assigned to each cluster, and *Amp.* is task share divided by seed share. Many smaller clusters receive greater amplification under Equation 1.

ID	Label	Seeds	Task (%)	Amp.	ID	Label	Seeds	Task (%)	Amp.
57	Data Structure Manipulation &	62,500	8.84	0.4	65	SSH Remote Administration & A	1,058	1.14	3.3
0	Numeric Computation & Convers	46,088	5.84	0.4	91	Documentation Generation & Ma	1,034	0.75	2.2
5	Text Processing & Parsing Uti	22,693	4.00	0.5	52	Filesystem Mounting & Managem	951	0.72	2.3
21	Build Script Configuration &	15,990	2.21	0.4	70	System Service Management	934	0.67	2.2
46	Python Environment & Package	11,728	1.79	0.5	43	Algorithmic Data Processing &	930	0.92	3.0
31	Python Debugging & Diagnostic	7,497	0.96	0.4	48	Cryptographic Key & Certifica	904	0.80	2.7
27	Process Management & Monitori	6,316	1.54	0.8	8	Document Conversion & Process	882	0.82	2.9
4	Library Bug Diagnosis & Repai	5,850	1.20	0.6	69	Performance Profiling & Optim	826	0.77	2.9
32	Django Web Deployment & Confi	5,685	1.24	0.7	99	Terminal Output & Display Con	795	0.74	2.9
40	Web Scraping & HTTP Automatio	5,260	1.61	0.9	15	Java Code Compilation & Refac	790	0.13	0.5
13	Geospatial Computation & Geom	5,247	1.95	1.1	83	Disk Usage Inspection & Analy	775	1.18	4.7
23	Software Testing & Debugging	5,016	0.81	0.5	33	File Transfer & Streaming	774	0.75	3.0
6	File Search and Listing Operat	4,905	2.02	1.3	97	Quantum Circuit Debugging & O	770	1.07	4.3
35	Shell Scripting & CLI Configu	4,354	1.22	0.9	11	User Session & Authentication	759	0.39	1.6
9	Data Sorting & Aggregation	3,901	1.66	1.3	41	Graph Algorithms & Data Struc	752	0.61	2.5
56	Numerical Array & Matrix Comp	3,800	1.51	1.2	1	Media Processing & Streaming	745	0.74	3.0
2	Container Orchestration & Man	3,716	1.17	1.0	72	Email Automation & Processing	734	1.10	4.6
16	Network Diagnostics & Monitor	3,662	1.20	1.0	39	Database Provisioning & Scrip	680	0.59	2.7
58	Web Scraping & XML Parsing	3,404	1.49	1.3	81	GUI Application Debugging & D	651	0.34	1.6
20	Database Backup & Migration	3,252	1.25	1.2	94	GUI & Display Automation	641	0.43	2.0
88	Git Repository Automation & M	3,226	1.11	1.1	92	C++ Systems Programming & Ref	633	0.19	0.9
7	Text File Processing & Manipu	3,080	1.55	1.5	78	Node.js Environment Management	530	0.54	3.2
19	CLI Loop & Arithmetic Scripti	3,015	1.22	1.2	90	Color Processing & Analysis	519	1.00	5.9
22	Character Encoding & Text Pro	2,559	1.39	1.7	76	Terminal Input & Serial Contr	503	0.50	3.1
14	Go Development & Testing	2,558	0.48	0.6	77	System Command & Package Disc	460	0.55	3.6
24	Image Processing & Manipulati	2,516	1.34	1.6	30	Localization & Runtime Enviro	428	0.48	3.4
3	File Archiving & Compression	2,182	1.36	1.9	28	Password Generation & Crackin	376	0.65	5.3
64	Tabular Data Processing & Ana	2,122	1.20	1.7	80	Terminal Text Editing & Forma	373	0.38	3.1
93	Binary Data & Cryptography Ut	1,885	1.23	2.0	42	Time Series Data Processing	372	0.64	5.3
50	Date & Time Manipulation	1,819	1.21	2.0	47	System Security & Auditing	323	0.39	3.8
29	JSON Data Processing & Serial	1,733	1.07	1.9	75	Cloud Storage Sync & Manageme	323	0.56	5.3
74	ASCII Art Pattern Generation	1,672	1.30	2.4	25	Scheduled Task & Notification	291	0.69	7.2
17	Task Queue & Process Scheduli	1,630	0.69	1.3	53	Symbolic Mathematics Library B	271	0.77	8.7
26	File Comparison & Integrity C	1,572	0.99	1.9	49	Machine Learning Model Evaluat	255	0.37	4.5
82	Database Optimization & Perfo	1,532	0.86	1.7	55	Desktop File & App Integratio	243	0.29	3.6
36	Random Data Generation & Mani	1,512	1.17	2.4	68	MongoDB Database Operations	142	0.53	11.4
38	Configuration File Processing	1,500	0.82	1.7	71	GPU Computing & CUDA Debuggin	105	0.27	8.0
37	File Renaming & Batch Text Pr	1,487	1.15	2.4	84	Perl Module & Kernel Manageme	100	0.21	6.5
10	Software Library API Maintenan	1,447	0.42	0.9	87	SMB/CIFS Share Administration	100	0.47	14.5
79	File Path Manipulation & Reso	1,424	0.95	2.1	61	Sleep Until Wall-Clock Time	100	0.28	8.7
45	Rust Development & Refactorin	1,411	0.36	0.8	85	Persistent Contact Data Storag	100	0.18	5.6
51	Data Visualization & Plotting	1,362	1.04	2.3	86	SSH Library Repair & Recovery	100	0.13	4.1
34	Network Configuration & Diagn	1,321	1.05	2.4	59	Kaggle Dataset Management	100	0.13	4.0
18	System Resource Monitoring &	1,247	0.69	1.7	98	Browser Automation Setup & Te	100	0.35	10.9
67	Package Management Operations	1,189	0.82	2.1	62	Dynamic Linker Cache Repair	100	0.15	4.6
73	Self-Hosted Web Application De	1,131	0.54	1.5	89	BitTorrent Client & Torrent M	100	0.28	8.6
54	File Ownership & Permission M	1,126	0.94	2.6	60	Plone/Zope Buildout Management	100	0.12	3.8
12	Scientific Computing & Numeri	1,123	0.70	1.9	95	Reinforcement Steel Quantity T	100	0.46	14.0
66	Audio & Media Processing	1,112	0.91	2.5	96	Python Nested List Initializat	100	0.36	11.0
44	Log Analysis & Monitoring	1,104	1.01	2.8	63	XML XPath Query Handling	100	0.34	10.5

B Environment Construction and Recovery

This appendix reports the construction-side statistics underlying §3.3 and §3.4, including verifier defects and the repair-routing ledger. Task-level verifier statistics are computed from the saved configurations associated with the frozen 16,384 construction instances, while routing statistics count events across all repair passes for those instances. Table 8 reports both views.

The no-op check flags 5,095 of the 16,384 saved verifier configurations (31.1%). Among the flagged configurations, 3,462 assign partial credit to the untouched state, with a mean reward of 0.428 over these partial-credit cases. The remaining configurations either crash or contain a free-passing case, fully satisfy the task without agent actions, or yield no usable reward. Passing the no-op check after repair establishes only that the initialized state receives a well-formed zero score; it does not establish that the verifier accepts every valid solution. Figure 7 summarizes these defect classes and the trajectory-reward distribution over executable tasks.

Table 8 separates event-level routing decisions from unique-task outcomes. Among 5,471 tasks with at least one recorded repair event, 2,283 ultimately complete the pipeline and 3,188 do not. The

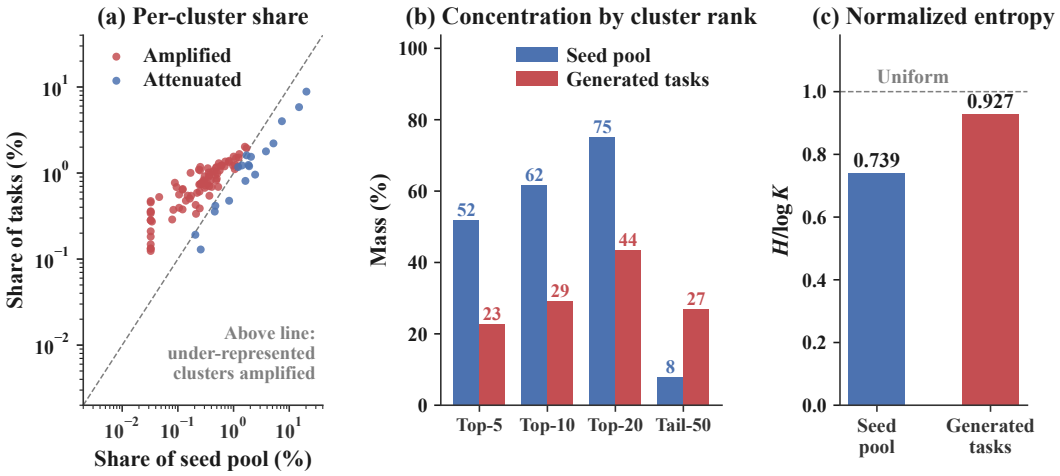

Figure 5: Effect of corpus anchoring on the *generated-task* distribution. **(a)** Seed-pool mass and selected-seed mass in generated tasks for each macro-cluster. **(b)** Cumulative mass in head and tail clusters. **(c)** Normalized cluster entropy before and after task synthesis.

Table 7: Coarse domain shares on the seed and task sides. A task can belong to multiple domains, so task shares need not sum to 100%. *Amp.* is the ratio of task share to seed share. Terminal-native categories generally have larger task-to-seed ratios than generic programming categories.

Domain	Seed (%)	Task (%)	Amp.	Domain	Seed (%)	Task (%)	Amp.
Data Processing	12.35	26.05	2.1	Multimedia	0.44	2.07	4.7
Coding Exercise	12.50	15.87	1.3	Data Structures	0.82	1.97	2.4
Python	10.25	15.57	1.5	Dev Environment	0.54	1.66	3.1
Algorithm	9.14	15.32	1.7	Software Testing	0.86	1.60	1.9
System Admin	3.42	13.04	3.8	API Integration	0.46	1.57	3.4
File System	3.13	11.71	3.7	Data Extraction	0.37	1.55	4.2
Automation/Scripting	3.41	10.69	3.1	Container/Virtual	0.60	1.51	2.5
Web/Backend	4.12	10.48	2.5	Version Control	0.49	1.14	2.4
Shell/CLI	3.49	9.78	2.8	Performance	0.21	0.97	4.7
Text Processing	3.25	7.75	2.4	Cloud	0.24	0.94	4.0
Scientific Computing	3.08	7.33	2.4	Bioinformatics	0.50	0.89	1.8
Programming Basics	4.03	7.04	1.7	ML/AI	0.36	0.89	2.4
Network	1.88	6.86	3.6	DevOps	0.21	0.85	4.0
Build/CI/CD	3.21	6.69	2.1	Email/Automation	0.08	0.82	10.0
Database	1.40	4.53	3.2	Gaming	0.22	0.62	2.9
Data Analysis	1.36	4.30	3.2	Hardware/Device	0.11	0.55	5.1
Software Dev	1.83	4.07	2.2	GUI	0.16	0.55	3.4
Security/CTF	0.74	3.19	4.3	GIS	0.12	0.43	3.7
Image Processing	0.81	3.09	3.8	Log Analysis	0.04	0.35	8.0
Math/Finance	1.25	2.89	2.3	HPC	0.12	0.20	1.7
Document Processing	0.73	2.66	3.7	EDA/Chip	0.00	0.01	4.9
Data Formats	0.42	2.22	5.3				

completed group comprises 1,249 tasks recovered through local verifier repair and 1,034 recovered through cross-stage routing. Verifier-local recovery and executable-task completion use different endpoints: 2,504 saved configurations reach a passing no-op state after repair, whereas 1,249 tasks are ultimately classified as locally recovered executable tasks under the full pipeline outcome and pass budget. Figure 3(b) reports trajectory scores for the first-pass, locally recovered, and cross-stage recovered cohorts.

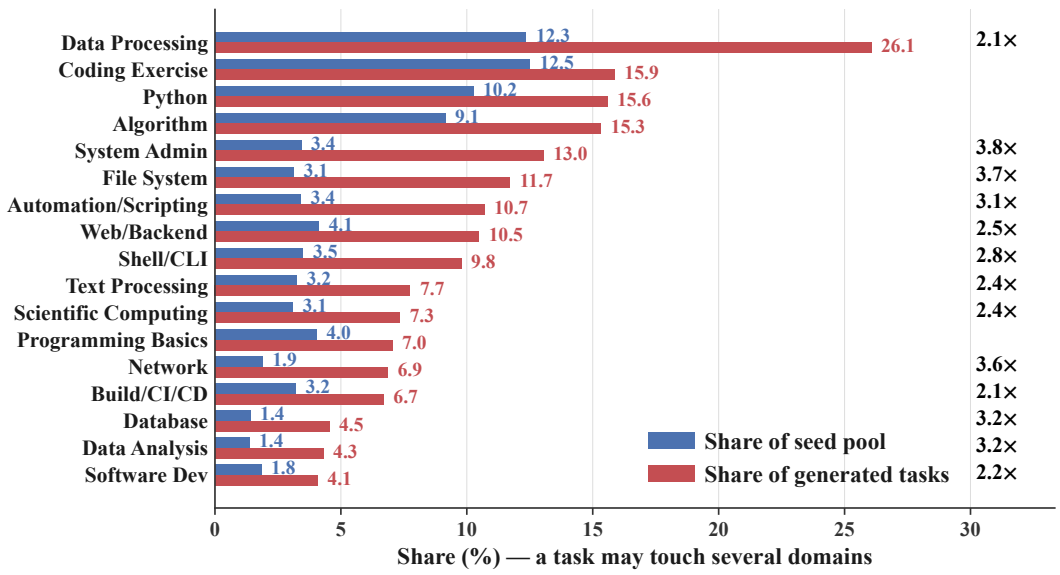

Figure 6: Seed-side and task-side shares for the 17 coarse domains with the largest task shares. Task-to-seed ratios above 2× are labeled. Terminal-native categories generally have larger ratios than generic programming categories.

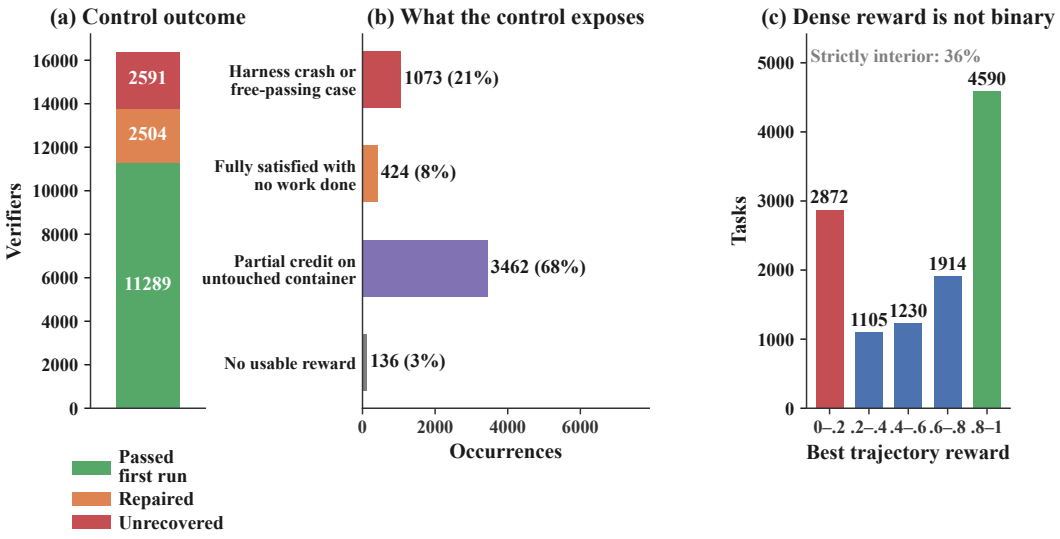

Figure 7: Verifier diagnostics and trajectory rewards. **(a)** The no-op check flags 5,095 of 16,384 saved verifier configurations; local regeneration produces a passing no-op result for 2,504 of the flagged configurations (49.1%). **(b)** Mutually exclusive defect classes among the 5,095 flagged configurations. **(c)** Best trajectory reward for each of the 11,713 executable tasks.

C Snapshot Caveats

Four caveats apply to the statistics in Appendices A and B. All reported counts are computed from a fixed artifact snapshot; later pipeline runs are excluded.

First, the round-by-round acceptance curve in Figure 4(b) is estimated from 682 logged task-generation attempts, whereas the final 99.25% acceptance rate is computed over all 50,382 candidate-group attempts. Failed first judgments are revised for up to two rounds before final rejection. In this snapshot, the protocol therefore operates primarily as a revision stage rather than a single-pass rejection filter.

Table 8: Verifier and repair-routing statistics for the frozen 16,384 construction instances. The verifier block is computed from saved task configurations. Routing rows count events across repair passes, with shares normalized within gate type: 5,714 build-failure events and 10,827 no-op-failure events. Unique-task outcomes are reported separately. An acceptable no-op result after repair is verifier-local and does not imply executable-task completion.

Block	Quantity	Count	Share
Verifier gate	verifiers checked	16,384	—
	passed initially	11,289	68.9%
	flagged by no-op gate	5,095	31.1%
	passing the no-op gate after repair	2,504	49.1% of flagged
	partial credit in initial state	3,462	67.9% of flagged
	crash or free-passing case	1,073	21.1% of flagged
	fully satisfied with no work	424	8.3% of flagged
	no usable reward	136	2.7% of flagged
	mean reward among partial-credit cases	—	0.428
Routing events (events)	build gate → environment	5,714	100.0% of build
	no-op → verifier	8,301	76.7% of no-op
	no-op → environment	1,503	13.9% of no-op
	no-op → image build	1,023	9.4% of no-op
Repaired tasks (unique tasks)	tasks with at least one repair event	5,471	—
	later pipeline completion	2,283	41.7%
	later pipeline failure	3,188	58.3%

Second, construction, verifier, and routing statistics use the first 16,384 records in the pipeline-results stream. This subset is a prefix rather than a random sample, so its retention and repair rates may not transfer unchanged to the complete pool of 50,003 accepted task descriptions.

Third, source coverage remains uneven in absolute terms. Some specialized terminal domains are represented by relatively small seed subsets, and smoothed sampling cannot introduce workflows absent from the underlying corpora. Expanding source coverage in these areas may therefore broaden the reachable task distribution.

Fourth, neither trajectory length nor recovery-cohort scores isolate task difficulty. A longer run, and a lower score from first-pass tasks through local recovery to cross-stage recovery, can reflect intrinsic difficulty, which tasks the gates retain, residual environment defects, or differences in the verifier. The comparisons in Figure 3 are observational, and they do not by themselves establish that longer runs or recovered tasks are harder.