From Issues to Learning Signals: A Survey of Data for Coding Agents

Benchmarks, Executable Task Synthesis, Trajectory Curation, and Post-Training

JLULLM LAB

Independent Research Draft

Coding agents are trained and evaluated on artifacts that are often called datasets but are structurally closer to executable experiments. A useful item contains not only a natural-language task, but also a repository snapshot, a reproducible environment, an action interface, a verifier or rubric, one or more interaction trajectories, and provenance sufficient to prevent evaluation leakage. This survey develops a data-centric account of coding agents around that *executable experience bundle*. We first organize benchmarks from function-level code generation through repository issue resolution, terminal work, long-horizon software evolution, and automated post-training. We then study task synthesis as the joint construction of prompts, environments, and verifiers; trajectory synthesis as search over a rollout graph followed by outcome, behavioral, and causal filtering; and post-training as the routing of curated artifacts into supervised fine-tuning, preference optimization, process-reward learning, and reinforcement learning with verifiable rewards. The review highlights a central tension: executable tests make coding unusually scalable for learning, yet incomplete tests, environment drift, public-task contamination, and scaffold-dependent scores make the same signal brittle. We close with a lifecycle-oriented research agenda for versioned benchmarks, verifier diversity, failure-aware trace pruning, and leakage-resistant data flywheels.

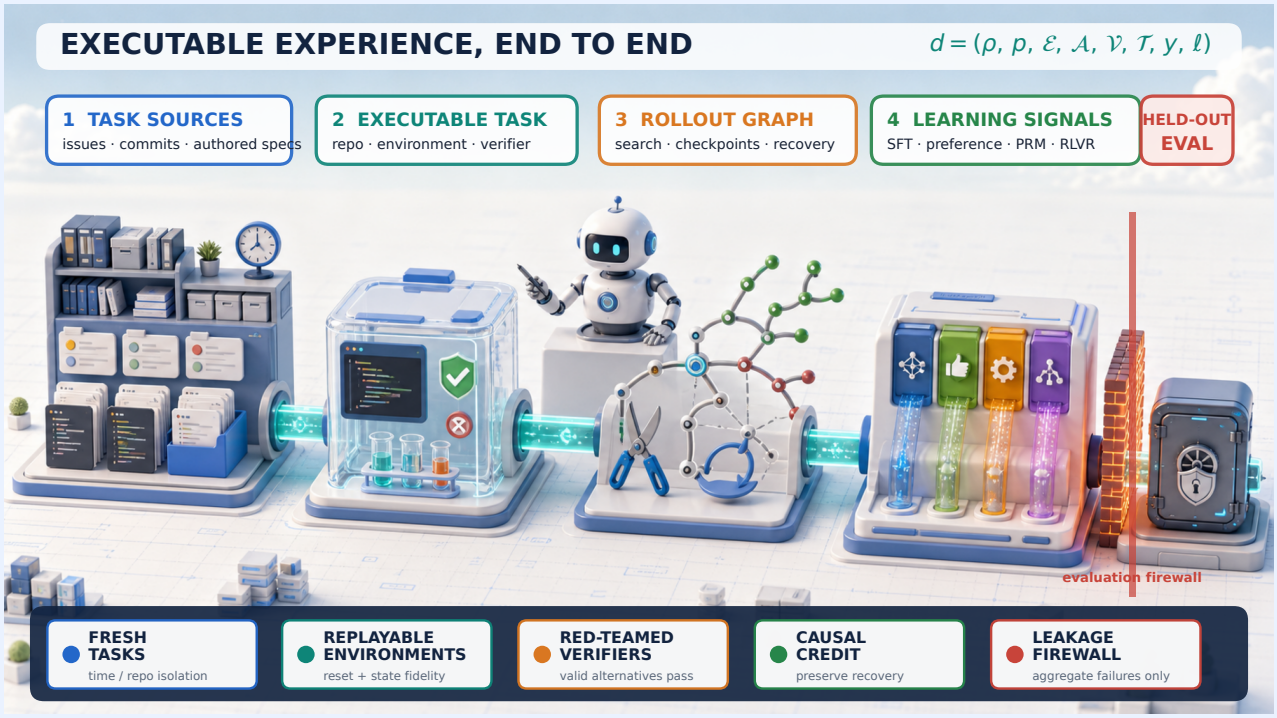

Figure 1: The data engine behind coding agents. Task sources become replayable executable tasks, rollout graphs, and objective-specific learning signals. The visible firewall separates training artifacts from held-out evaluation.

Contents

1 Introduction

Coding agents have moved the unit of code intelligence from a completion to an intervention. A model is no longer judged only by whether it emits a function that passes unit tests; an agent must inspect an unfamiliar repository, form and revise hypotheses, edit several files, execute tools, interpret failures, preserve state across a long interaction, and finally leave the workspace in a verifiable condition. SWE-bench made this transition concrete by packaging real GitHub issues with repository snapshots and executable regression tests (Jimenez et al., 2024). Subsequent benchmarks broadened the setting across languages, terminals, multimodal interfaces, long-lived projects, and machine-learning research (Zan et al., 2025; Merrill et al., 2026; Starace et al., 2025; PostTrainBench Team, 2026). In parallel, training resources such as SWE-Gym, R2E-Gym, and SWE-smith turned repositories into scalable sources of supervised trajectories and reinforcement-learning environments (Pan et al., 2024; Jain et al., 2025; Yang et al., 2025).

The resulting literature is usually narrated through models or algorithms: a new scaffold obtains a higher resolve rate, a stronger teacher generates successful trajectories, or a reinforcement-learning recipe improves a benchmark. This view hides the shared substrate. The reported capability depends on what task was selected, which revision of the repository was reconstructed, what tools and network access were allowed, how the tests map to the written requirement, what actions were retained in the trajectory, and whether the same artifact leaked into training. The *data* is therefore not a prompt with a desired answer. It is an executable experiment whose parts determine both the behavior learned and the claim the evaluation can support.

This survey develops a data-centric account of coding agents. We call the basic unit an **executable experience bundle**: a versioned repository state; a task specification; a replayable environment and action interface; a verifier or rubric; a trajectory or rollout graph; downstream labels or rewards; and provenance linking every component. This shift in unit resolves four recurring category errors.

1. A benchmark is not only a set of prompts. Its main asset is frequently the environment and verifier, and its main liability is frequently the mismatch between the written task and that verifier.
2. Prompt synthesis is not complete when an instruction sounds realistic. For interactive code, task, environment, and verifier must be generated and validated jointly.
3. A trajectory is not intrinsically positive or negative. A failed branch may contain useful localization, a successful trace may include wasteful actions, and an apparent error may be a productive step that is later repaired.
4. "SFT versus RL" is not a sufficient algorithm taxonomy. One must separately identify the source of experience, the learning signal, whether data is collected offline or on-policy, and the granularity of credit.

Why a data survey now? The field has reached an inflection point at which its cleanest signal—executable correctness—also causes its most consequential blind spots. Tests enable cheap, scalable labels and reinforcement learning with verifiable rewards. Yet passing an incomplete test suite can reward an incorrect patch, flaky setup can turn agent quality into infrastructure noise, and a public issue–patch pair can appear in pretraining or post-training corpora. These are not hypothetical caveats. In 2026, OpenAI stopped reporting SWE-bench Verified after finding both contamination and material task-quality problems; its audit of 138 difficult tasks judged at least 59.4% to contain a substantive issue (OpenAI, 2026b). A later audit of the public SWE-bench Pro split estimated that roughly 30% of tasks were broken through overly strict tests, underspecified prompts, low coverage, or misleading requirements (OpenAI, 2026a). The lesson is not that executable benchmarks have failed. It is that benchmark construction, maintenance, and audit are first-class data research.

The same lesson becomes sharper when coding agents are asked to build training data or train models. PostTrainBench gives an agent a base model, compute, a command-line environment, and a time budget, then evaluates the resulting checkpoint on held-out capabilities (PostTrainBench Team, 2026). Here data

choice, contamination policy, model identity, external API access, and evaluation isolation are part of the task environment. The coding agent is no longer merely the object trained on data; it is an actor that searches for, generates, filters, and may inadvertently leak data. This makes a lifecycle view unavoidable.

Scope. We cover data for language-model coding agents that interact with repositories, tools, terminals, build systems, tests, or software-production interfaces. Function-level code benchmarks are included as historical and diagnostic context, but our main subject is repository- and environment-grounded agency. We treat benchmarks, training sets, sandboxes, verifiers, trajectories, and data-generation systems together when they instantiate the same executable unit. We discuss algorithmic changes only where they alter the data required, the label consumed, the credit assigned, or the distribution induced by the policy. General web, GUI, and scientific agents enter only when they clarify a coding-specific design choice.

Relation to prior surveys. Surveys of long-horizon agents provide the broader system framing across model, harness, environment, and evaluation; software-agent surveys organize architectures, applications, and evaluation (RUC-NLPIR and collaborators, 2026; Wang et al., 2024). A recent general survey of agent data synthesis classifies task specifications, trajectories, feedback, and environments across domains (Zhang et al., 2026). Our scope is narrower and more operational: we follow the executable artifacts of repository and terminal work from benchmark construction through task and verifier synthesis, rollout-graph curation, and coding-specific post-training. The focus is therefore the integrity and reuse of one data unit across an end-to-end learning lifecycle.

Contributions. This survey makes five contributions. First, it formalizes the executable experience bundle and distinguishes *task artifacts*, *experience artifacts*, and *learning artifacts*. Second, it organizes benchmarks by software-work unit and verifier boundary rather than by name, including issue repair, professional workflows, terminal execution, release-level evolution, and automated post-training. Third, it reframes prompt synthesis as the co-generation of task, environment, and verifier, with explicit validity gates and repair loops. Fourth, it treats trajectories as rollout graphs and develops a pruning taxonomy spanning trajectory, branch, step, and token levels. Fifth, it maps each resulting artifact to SFT, preference optimization, process-reward learning, and online RL, yielding concrete design patterns for leakage-resistant data flywheels.

The survey in one sentence

Coding-agent progress is best understood as the quality of an executable-data pipeline: source tasks become validated environments, environments produce attributed rollout graphs, graphs are routed into suitable learning objectives, and fresh held-out evaluation returns failures without collapsing the boundary between training and measurement.

Figure 2 provides the visual roadmap for the survey. Its four principal branches answer four different data questions: what is measured and held out, how executable tasks are constructed, which parts of interaction are reusable, and how those artifacts become learning signals. Quality, evidence, governance, and lineage cut across all four branches rather than forming separate data silos.

Organization. Section 2 defines the data unit and lifecycle. Section 3 reviews benchmarks and post-training evaluations. Section 4 covers prompt, environment, verifier, and rubric synthesis. Section 5 studies rollout generation and pruning. Section 6 connects artifacts to SFT, preference methods, process supervision, and RL. Section 7 distills end-to-end patterns; Section 8 identifies open problems.

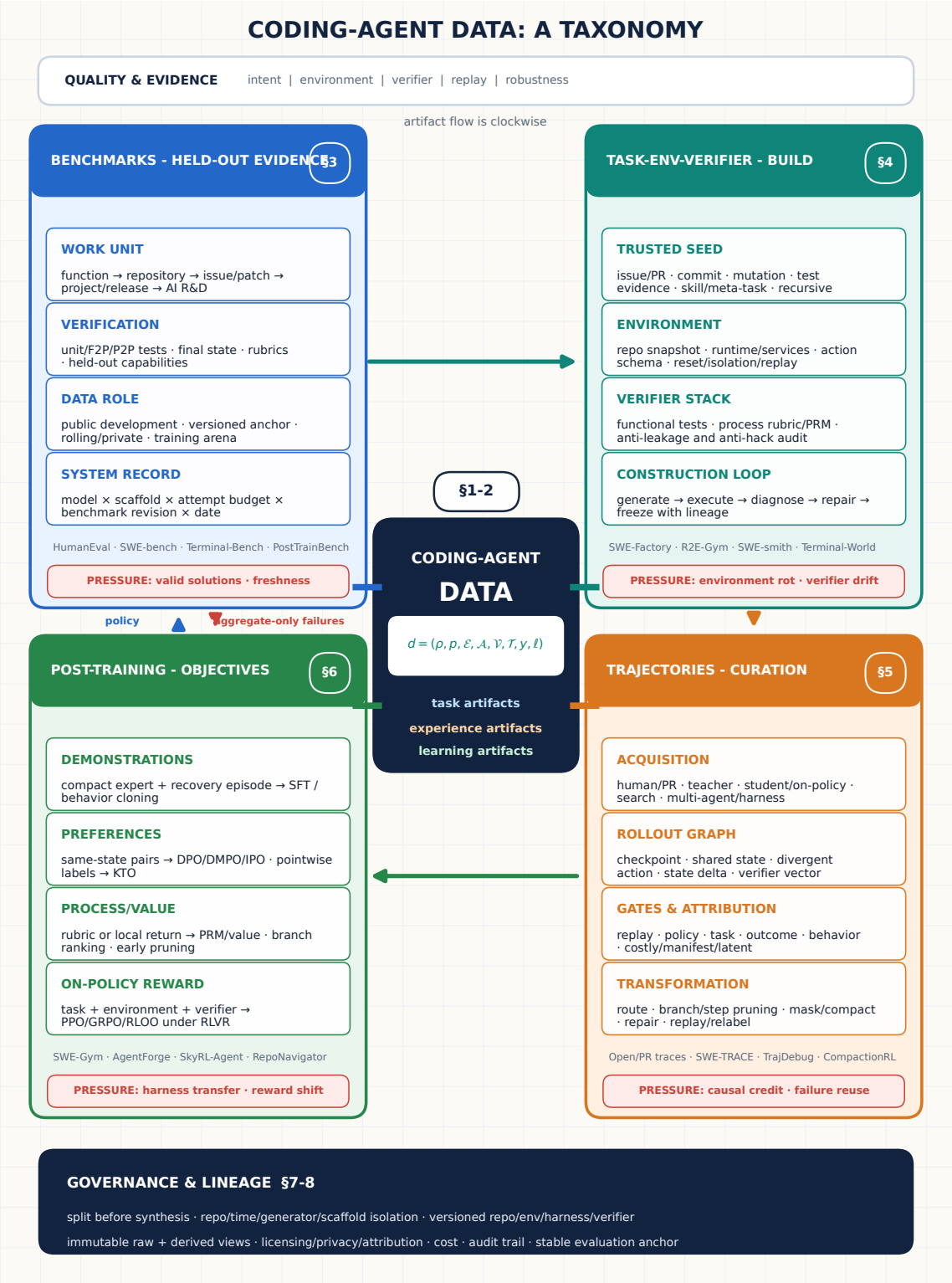

Figure 2: A data-centric mind map of coding-agent experience. The center is the executable experience bundle rather than an isolated prompt–response pair. The four branches organize benchmarks, joint task–environment–verifier construction, rollout-graph curation, and post-training objectives. The upper band records quality and evidence requirements; the lower band records split isolation, provenance, rights, and cost. Solid arrows denote artifact flow. Red pressure points summarize the validity, credit-assignment, transfer, and governance problems developed throughout the survey.

Table 1: The fields of an executable experience bundle. Missing fields do not disappear; they become undocumented assumptions in the harness or evaluation protocol.

Field	Typical contents	What it controls	Common silent failure
ρ: provenance	repo URL, commit, patch lineage, license	identity, chronology, split isolation	future history or gold patch leaks into context
p: task	issue, PRD, dialogue, screenshot, constraints	observable intent and ambiguity	prompt omits behavior enforced by tests
$\mathcal{E}$: environment	container/VM, dependencies, services, fixtures	reachable states and replayability	setup failure is scored as model failure
$\mathcal{A}$: actions	shell, patch tool, editor, browser, GUI, API	agent–world interface and scaffold	models are compared under unequal affordances
$\mathcal{V}$: verifier	F2P/P2P tests, hidden tests, rubric, judge	observable correctness and reward	narrow tests reward brittle or hacked solutions
$\mathcal{T}$: experience	linear trace, branches, checkpoints, recovery	behavioral evidence and policy distribution	final success erases costly or unsafe behavior
y: labels	outcome, pairwise choice, process score, mask	learning objective and credit granularity	a trajectory-level label is copied to every step
ℓ: lineage	generator, model, prompt, cost, date, split	audit, deduplication, governance	synthetic descendants cross the test boundary

2 The Executable Experience Unit

2.1 From examples to versioned experiments

Let one coding-agent datum be

$$d = (\rho, p, \mathcal{E}, \mathcal{A}, \mathcal{V}, \mathcal{T}, y, \ell). \tag{1}$$

Here ρ is repository and dependency provenance; p is the visible task or dialogue; $\mathcal{E}$ is the initial executable environment and transition function; $\mathcal{A}$ is the allowed action interface; $\mathcal{V}$ is a verifier, rubric, or reward program; $\mathcal{T}$ is a trajectory or a graph of rollouts; y contains outcome, preference, process, or token-level labels; and ℓ records lineage, licensing, generation policy, costs, and split membership. A dataset D is a versioned collection of these bundles together with policies that define access, retry budgets, scoring, and contamination boundaries.

Equation 1 is intentionally broader than the records released by any one benchmark. An evaluation set may omit $\mathcal{T}$ and keep hidden portions of $\mathcal{V}$. A supervised set may expose only selected paths and token masks. An online RL arena may store $(\rho, p, \mathcal{E}, \mathcal{A}, \mathcal{V})$ and generate $\mathcal{T}$ on policy. The value of the definition is not that every field must be public, but that every field must be controlled.

2.2 Three artifact layers

The bundle separates into three layers that should be versioned independently.

Task artifacts. The task layer $b = (\rho, p, \mathcal{E}, \mathcal{A}, \mathcal{V})$ defines an executable problem. It can be used for evaluation, on-policy sampling, or repeated generation of expert traces. Its quality is largely independent of whether a particular model solves it. A high-quality task may be hard; a low-quality task may be easy because the verifier is weak.

Experience artifacts. Executing policy π in task bundle b produces a trajectory

$$\tau = (s_0, o_0, a_0, s_1, o_1, a_1, \ldots, s_H), \qquad a_t \sim \pi(\cdot \mid h_t), \tag{2}$$

where the environment state s_t includes files, processes, services, version-control state, and hidden fixtures, while h_t is the agent-visible history after any context management. Multiple retries, search branches, or agents induce a directed acyclic rollout graph rather than a single sequence. Environment checkpoints identify when two branches share the same causal state.

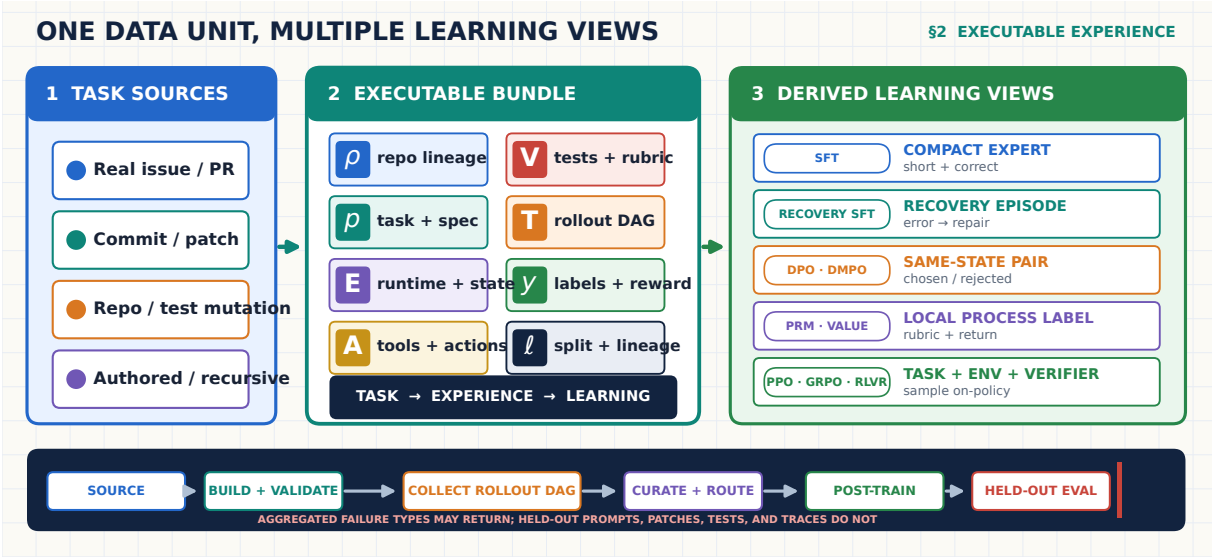

Figure 3: One data unit, three artifact layers. Task sources are packaged into the versioned executable experience bundle, from which several objective-specific learning views can be derived without rewriting the immutable raw record. The footer makes each lifecycle transition and the held-out evaluation firewall explicit.

Learning artifacts. Filtering and annotation map raw experience into training records: successful subsequences for SFT, state-matched chosen/rejected branches for preference learning, stepwise rubric comparisons for a process reward model (PRM), or task bundles for online RL. The distinction matters because the same rollout graph can yield several learning products without pretending they provide the same supervision.

2.3 A lifecycle, not a one-way corpus

Figure 3 expands the title-page overview. Benchmarks and production failures provide task seeds. A builder reconstructs or synthesizes the executable task. Policies then generate rollout graphs under recorded budgets. Validation, failure attribution, and pruning create learning artifacts. Post-training changes the policy distribution, which changes what data is useful. Held-out evaluation identifies new failure clusters, but its tasks and hidden verifiers must remain isolated from the training branch.

2.4 Quality is multidimensional

A binary pass bit is necessary but not sufficient for either task validation or experience curation. We characterize bundle quality along seven dimensions:

$$Q(d) = \left(q_{\text{intent}}, q_{\text{env}}, q_{\text{verify}}, q_{\text{behavior}}, q_{\text{novel}}, q_{\text{replay}}, q_{\text{govern}}\right). \tag{3}$$

Intent validity asks whether the task is sufficiently specified. *Environment validity* asks whether setup is stable and the initial state is faithful. *Verifier validity* asks whether valid implementations pass and invalid shortcuts fail. *Behavioral validity* asks whether the trace is grounded, adaptive, persistent, and non-cheating. *Novelty* measures leakage and redundancy. *Replayability* covers determinism and artifact preservation. *Governance* covers licenses, privacy, and attribution.

These dimensions are not interchangeable. Larger datasets can reduce sampling variance while amplifying a systematic verifier error. More difficult tasks can be difficult only because their environment is broken. Shorter traces can be efficient or can omit necessary recovery. A data pipeline should therefore preserve vector-valued quality metadata and defer scalarization to a documented selection policy.

Boundary condition: benchmark data becomes training data

Once a model or its generator sees a benchmark's prompt, hidden verifier, gold patch, or released trajectory, that artifact no longer provides an unbiased evaluation of the resulting system. Converting evaluation tasks into RL environments is technically convenient, but it must create a new training split and a separately sourced test boundary; a random split over tasks from the same repositories and time period is rarely sufficient.

3 Benchmarks and Evaluation Data

Benchmarks are both measurement instruments and potential data mines. A benchmark defines which work unit counts as a task, which state the agent may observe and change, which behaviors the verifier can distinguish, and which failure modes are invisible. Once its artifacts are used for post-training, however, it stops being an independent measurement instrument for that model. This section therefore organizes benchmarks by *work unit* and *verification boundary*, then separates evaluation assets from training environments.

3.1 Six levels of software work

L1: function-level code. HumanEval, MBPP, BigCodeBench, and LiveCodeBench ask for bounded programs under unit tests. They remain useful diagnostics for syntax, algorithmic reasoning, and test-time sampling, but usually lack a persistent repository, interactive state, and delayed recovery. They measure a base capability rather than the full coding-agent loop.

L2: repository-grounded generation. CrossCodeEval, RepoBench, RepoExec, and DevEval introduce cross-file dependencies, repository retrieval, and executable checks. The agent must ground a local implementation in a larger codebase, but interaction is often shallow and the task boundary is selected in advance.

L3: issue resolution. SWE-bench established the dominant issue-to-patch protocol: a real issue, a pre-fix repository snapshot, and tests that distinguish the buggy state from a candidate repair (Jimenez et al., 2024). Variants extend languages, modalities, scale, freshness, and access boundaries. This level is the center of present coding-agent evaluation and the most common source of RL environments.

L4: professional software workflows. Feature implementation, test writing, refactoring, architectural choice, release evolution, and collaborative clarification require richer artifacts than an isolated bug report. SWE-Lancer maps tasks to real freelance work; SWE Atlas separates codebase question answering, test writing, and refactoring; SWE-EVO packages release-sized changes; SWE-Interact introduces a user simulator; and original-task collections such as DeepSWE reduce dependence on already merged public patches (OpenAI, 2025; Scale AI, 2025; Le et al., 2025; DataCurve, 2026).

L5: terminal and operating-system work. InterCode, Terminal-Bench, and OSWorld move the unit from a patch to a final machine state (Yang et al., 2023; Merrill et al., 2026; Xie et al., 2024). Dependency installation, process control, data transformation, debugging, and recovery become part of the task. The verifier may inspect files, services, databases, or GUI state rather than a single test command.

L6: AI R&D and post-training. MLE-bench, MLGym, PaperBench, and PostTrainBench ask agents to run experiments, reproduce research, or improve a model under compute and time constraints (Chan et al., 2024; Meta AI and collaborators, 2025; Starace et al., 2025; PostTrainBench Team, 2026). At this level the output is a model, experiment record, or research artifact. The agent's data acquisition policy and access to external models become part of what must be evaluated.

3.2 SWE-bench as a benchmark lifecycle

SWE-bench is best understood not as one immutable leaderboard but as a sequence of data interventions. The original benchmark demonstrated that a repository snapshot and fail-to-pass tests could turn real

maintenance into executable evaluation. Lite reduced cost. Verified added human review. Multimodal and Multilingual changed the input and language distributions. Live and rebench emphasized recent, automatically renewable tasks. Pro added longer work and access-controlled splits. Each intervention improves one dimension while opening another source of uncertainty.

The 2026 audits make this lifecycle empirically visible. OpenAI reported that, among 138 difficult SWE-bench Verified tasks it manually audited, at least 59.4% had material issues with the problem statement or tests; it also described increasing evidence that frontier models had encountered benchmark solutions during training (OpenAI, 2026b). In a separate audit of the 731-task public SWE-bench Pro split, automated screening flagged 200 tasks and a human campaign flagged 249, leading to an estimate that roughly 30% were broken (OpenAI, 2026a). The dominant categories—overly strict tests, underspecified prompts, low coverage, and misleading prompts—are exactly mismatches among p, $\mathcal{E}$, and $\mathcal{V}$ in Equation 1.

Audit lesson

Test-passing is a powerful *interface* for evaluation, not a guarantee that the evaluation is valid. A benchmark release should be treated like software: versioned, regression-tested, red-teamed against alternative correct solutions, and accompanied by a change log. Leaderboard entries should name model, scaffold, benchmark revision, attempt budget, and evaluation date.

The lifecycle also clarifies why static and rolling evaluations should coexist. A frozen split supports longitudinal comparison but accumulates contamination and overfitting. A rolling split improves freshness but changes task distribution and complicates comparisons. A robust program reports both: a versioned anchor split for comparability and a time-gated rolling split for current generalization, with repository overlap, model cutoff, network access, and future Git history disclosed.

3.3 Verifier boundaries define what is measured

Most issue-resolution benchmarks use a functional core: fail-to-pass tests must turn green and pass-to-pass tests must remain green. This boundary has three virtues: it is cheap, automatic, and directly connected to executable behavior. It also omits several properties that professional software work values.

- **Specification coverage**: can the tests distinguish multiple plausible but incompatible readings of the prompt?
- **Regression surface**: are tests run beyond the narrow files or modules touched by the gold patch?
- **Maintainability**: does the patch respect interfaces, invariants, style, and future extensibility?
- **Security and safety**: does it introduce a vulnerability, unsafe dependency, secret, or policy violation?
- **Efficiency**: how much time, compute, context, money, and destructive exploration were used?
- **Reliability**: does the same model–scaffold system succeed across repeated runs rather than once?

Different benchmarks expose different parts of this vector. SWE-EVO reports partial fix rate for release-scale work; SWE-Lancer uses end-to-end browser tests or professional choice tasks; PaperBench decomposes research reproduction into rubric nodes; terminal benchmarks inspect final state. The comparison is therefore not a single ladder from "easy" to "hard." It is a matrix of work units and observable properties.

For stochastic agents, pass@k answers whether at least one of k attempts works, while pass^k asks whether all repeated attempts work. The first rewards test-time search; the second measures consistency. Both should be accompanied by token, wall-clock, and tool-call budgets. Otherwise an apparent model improvement may be a sampling or scaffold improvement. Scores belong to the system

$$S = S(\theta, \mathcal{G}, \mathcal{E}, \mathcal{V}, B, v, t), \tag{4}$$

where θ is the model, $\mathcal{G}$ the scaffold, B the attempt and compute budget, v the benchmark revision, and t the evaluation date.

3.4 Evaluation sets versus training arenas

SWE-Gym, R2E-Gym, SWE-smith, Multi-SWE-RL, and SWE-rebench V2 exemplify a second category: collections intended to produce trajectories, verifiers, or online-RL environments (Pan et al., 2024; Jain et al., 2025; Yang et al., 2025; Zan et al., 2025; Badertdinov et al., 2026). They optimize for scale, executable setup success, diversity, and sampling throughput rather than solely for leaderboard stability.

At least two or three independent separation axes should be used between an arena and a formal evaluation: repository-disjoint, time-disjoint, task-type-disjoint, language-disjoint, organization-disjoint, or generator-disjoint. Random instance splits are vulnerable because tasks from the same repository share code, tests, conventions, issue templates, and dependency graphs. Synthetic descendants must inherit lineage from every seed and generator artifact so that deduplication can operate on ancestry, not only on text similarity.

3.5 PostTrainBench: when the agent controls data

PostTrainBench deserves separate treatment because it changes the role of the coding agent. In a run, an agent receives a small base language model, one H100 GPU, ten hours, command-line and permitted web access, and the objective of improving a weighted suite of seven held-out capabilities (PostTrainBench Team, 2026). The agent may inspect data, write training code, launch jobs, analyze intermediate evaluations, and submit a checkpoint. The benchmark therefore tests a compound workflow: data selection and synthesis, training-recipe choice, infrastructure operation, experimental judgment, and compliance with an evaluation firewall.

The setting exposes data hazards that issue repair can often ignore. An agent might train on the test items, query a stronger model through an unauthorized API, submit an instruction-tuned model when the target identity should remain a base model, or search for prior benchmark traces. PostTrainBench's evolving rules and audits illustrate that network policy, model access, task-specific lookup, artifact identity, and logging are components of $\mathcal{E}$ and $\mathcal{V}$, not administrative footnotes.

PostTrain Arena explores the complementary direction: participants contribute task–environment–verifier packages that are used as training arenas, then evaluate whether learning transfers to held-out environments (PostTrain Arena Team, 2026). Together, the two projects suggest an emerging two-sided view of coding-agent data: agents can be evaluated on whether they construct better models, and environment authors can be evaluated on whether their tasks produce transferable learning.

3.6 Recommended benchmark record

A responsible benchmark table should record more than name and score. We recommend the following schema: benchmark and revision; role (evaluation, train–eval, infrastructure); task unit and source; repository, organization, language, and time coverage; input modalities; environment and network policy; action interface and scaffold; verifier layers; metric and partial-credit policy; attempt budget; access tier; freshness cutoff; contamination controls; known audits and repaired tasks; trajectory availability; and licenses. Appendix C gives a compact reporting checklist.

Benchmark design principle

The next generation of coding-agent benchmarks should compete less on the number of public pull requests and more on original or rolling tasks, hidden behavioral verifiers, multiple valid implementations, repeated-run reliability, and explicit revision histories. Public tasks remain useful for development, but strong capability claims increasingly require access-controlled or genuinely fresh evidence.

4 Task–Environment–Verifier Synthesis

Calling this stage "prompt synthesis" understates the problem. A fluent issue is not a usable agent task if the repository does not build, the bug is absent, the tests expose the solution, or valid alternative

Table 2: Representative repository, terminal, and AI R&D benchmarks. Counts refer to the cited paper or named release and should never be separated from that revision. "Hidden" means that a meaningful verifier or task component is not downloadable by the evaluated agent.

Benchmark	Release / scale	Work unit	Environment	Verifier / metric	Data lesson
SWE-bench	2023; 2,294	GitHub issue–PR, 12 Python repos	repo snapshot; package setup	F2P/P2P tests; resolved	Established executable issue repair; public provenance creates contamination risk (Jimenez et al., 2024)
Verified	2024; 500	human-screened SWE-bench subset	official harness	human-reviewed prompt/test fit	Human QA improves a split but does not freeze its validity; retired by OpenAI in 2026 (OpenAI, 2026b)
Multi-SWE-bench	2025; 1,632	issue repair, seven languages	language-specific containers	executable tests; resolved	Language, build system, and harness difficulty remain entangled (Zan et al., 2025)
SWE-PolyBench	2025; 2,110	bug, feature, refactor; four languages	containerized repos	tests; PB500 / Verified subsets	Task-type diversity matters in addition to language diversity (Amazon Science, 2025)
SWE-bench-Live	2025; 1,319 initial	post-cutoff GitHub issues, 93 repos	automated RepoLaunch containers	tests; rolling releases	Freshness is a pipeline property, not a one-time filter (Microsoft Research and collaborators, 2025)
SWE-rebench	2025; 21,336	automatically mined issue repair	automated setup, reruns	executable tests; time split / SEM	Unifies collection and evaluation; setup reliability remains a gate (Badertdinov et al., 2025)
SWE-bench Pro	2025; 1,865	longer public, held-out, commercial tasks	hosted and downloadable variants	tests; resolved	Access tiers reduce leakage; public-split tests still require audit (Deng et al., 2025; OpenAI, 2026a)
SWE-Lancer	2025; 1,488	freelance coding and managerial decisions	offline repo / browser tests	triple-reviewed E2E or choice; dollars	Economic source is valuable, but coding and managerial scores are not commensurate (OpenAI, 2025)
SWE-EVO	2025; 48	release-sized evolution	full project history and test suites	fix rate, regression, completion	Partial progress is needed when all-or-nothing resolution is too coarse (Le et al., 2025)
Terminal-Bench 2	2026; 89	realistic terminal artifact	Docker via Harbor	final-state tests	Benchmark revisions are material: version 2.1 repaired 28 tasks (Merrill et al., 2026)
PaperBench	2025; 20 papers	research reproduction	code, compute, paper artifacts	8,316 rubric nodes	Hierarchical rubrics expose partial research progress beyond pass/fail (Starace et al., 2025)
PostTrainBench	2026; 4 base LMs, 7 targets	autonomous LM post-training	one H100, 10 hours, CLI/web policy	held-out weighted capabilities	Data search, model identity, external APIs, and test leakage become evaluated behavior (PostTrainBench Team, 2026)

Table 3: Evaluation assets and training arenas optimize different properties. A project may release both, but the same instances cannot play both roles for the same model.

Design question	Held-out evaluation	Training arena
Primary objective	unbiased, stable measurement	useful, scalable experience generation
Task visibility	prompt may be visible; critical tests often hidden	full prompt and reward interface available to trainer
Desired difficulty	discriminative across target systems	learnable mixture with non-degenerate reward
Repetition	limited and policy-controlled	many rollouts, checkpoints, and branches
Version policy	frozen releases plus audited patches	frequent repair, relabeling, and curriculum updates
Data isolation	no prompt, gold, verifier, or trace exposure	descendants tracked; overlap permitted only within train lineage
Failure handling	preserve failures for measurement	mine, repair, relabel, or resample failures

implementations fail. For coding agents, task synthesis is the joint construction of

$$b = (\rho, p, \mathcal{E}, \mathcal{A}, \mathcal{V}) \quad \text{such that} \quad \mathcal{V}(\rho) = 0, \qquad \mathcal{V}(\rho \oplus \Delta^{+}) = 1, \tag{5}$$

where Δ^{+} is at least one independently validated solution. The two equalities are necessary but not sufficient: near-miss and adversarial patches should fail, unrelated behavior should remain intact, setup should replay, and the prompt should justify what the verifier enforces.

4.1 Sources and synthesis paradigms

Existing pipelines differ mainly in which artifact is trusted first and which artifacts are generated around it. Table 4 organizes seven recurring paradigms.

Real-issue reconstruction. The highest-fidelity source is often a real issue–PR episode. The builder must recover the pre-fix state, identify dependency versions, reconstruct services and fixtures, separate developer tests from unrelated failures, and confirm that the issue is solvable from visible information. SWE-Factory treats this as an automated factory rather than a one-off benchmark curation exercise, including test restoration and environment setup (Guo et al., 2025). The main advantage is ecological validity. The main weakness is that prompt, patch, and tests share a public history and may be mutually overfit.

Programmatic and model-driven mutation. SWE-smith starts from a repository that already builds, applies transformations that inject executable bugs, and generates issue text around those failures (Yang et al., 2025). This reverses the expensive part of real-issue reconstruction: one validated base environment can support many tasks. Mutations can be distributed across languages, bug operators, and repository regions, but the pipeline must prevent shallow signatures such as suspicious single-line changes or wording that reveals the operator.

Generative environments. Terminal-World elevates a reusable skill—task, environment, verifier, and guideline—to the synthesis primitive and applies a generate–verify–repair loop to thousands of terminal sandboxes (Cheng et al., 2026). LiteCoder-Terminal similarly co-generates instruction, environment, solution, an adversarial verifier, and configuration, then filters both task validity and agent behavior (Peng et al., 2026). These systems demonstrate that data can be grown in domains without a clean GitHub issue history, but they shift trust toward the generator and its validation suite.

Recursive synthesis. Recursive task synthesis uses already validated tasks as seeds for harder executable descendants. RST reports expansion from hundreds of seeds into tens of thousands of tasks over repeated rounds, with instruction–verifier consistency checks and real execution (Li et al., 2026b). A key evaluation is whether difficulty rises in required state changes, dependencies, or solution work, rather than merely in prompt length. Every descendant should retain ancestry so that train–test separation remains valid under recursive generation.

4.2 Environment synthesis and replay

An environment builder should be evaluated separately from the task-solving agent. It receives a repository source and produces a hermetic, versioned state that satisfies a setup contract. The contract includes operating-system image, language runtime, package manager, dependency lock, services, databases, caches, credentials or mock services, hardware requirements, network policy, startup and teardown commands, resource limits, and a health check.

A useful environment qualification protocol has five stages:

1. **Build**: construct the image from declared sources with network access recorded.
2. **Baseline replay**: run the original test command multiple times and separate deterministic failures from flakiness.
3. **State check**: verify that the intended buggy or target state is present and future commits, gold patches, and hidden files are absent.

4. **Reset check**: execute representative actions, restore from a checkpoint or rebuild, and compare a state fingerprint.

5. **Isolation check**: confirm that the agent cannot modify the grader, inspect hidden tests, escape the sandbox, or query prohibited services.

The environment should expose structured failure categories. *Setup failure*, *task-invalid*, *agent timeout*, *agent error*, and *verifier failure* must not collapse into a single score of zero. Repeated execution estimates a task-level environment reliability $r_{\mathcal{E}}$. If a nominally correct patch passes with probability $r_{\mathcal{E}} < 1$, then one-run accuracy confounds policy quality with infrastructure noise. Reproducible snapshots and execution logs are therefore data-quality metadata, not only engineering conveniences.

4.3 Prompt construction and difficulty control

A task prompt should describe externally observable intent without encoding the gold implementation. For a synthetic task, one can derive a structured requirement from the verifier, then render multiple natural-language variants under an information policy. The structured layer may include target behavior, preconditions, examples, allowed interfaces, non-functional constraints, and explicitly hidden details. Rendering is the last step, not the source of truth.

Difficulty should be controlled through executable properties:

$$\delta(b) = f(n_{\text{files}}, d_{\text{dep}}, n_{\text{services}}, h_{\text{feedback}}, c_{\text{tests}}, a_{\text{spec}}, p_{\text{base}}), \tag{6}$$

where the terms measure changed-file scope, dependency depth, services, feedback delay, test coverage, specification ambiguity, and baseline policy success. Prompt length and gold-patch lines are weak proxies. A curriculum should contain a non-degenerate band in which the current policy sometimes succeeds; all-zero tasks provide no group-relative signal for many RL methods, while all-one tasks teach little.

Harness information is part of the task distribution. System prompts, tool schemas, edit protocols, context windows, observation truncation, retry policy, and shell semantics determine which trajectories are reachable. Single-scaffold scaling may fail to transfer across harnesses. Data releases should therefore record harness version and, where possible, augment a task across several semantically equivalent interfaces.

4.4 From tests to verifier stacks

A single test suite should be replaced by a **verifier stack** with three layers.

Functional layer. Build checks, fail-to-pass tests, pass-to-pass regression tests, hidden behavioral tests, static analysis, type checking, security scanners, and final-state assertions establish whether the artifact works. Hidden tests should target behavior rather than reproduce the gold implementation.

Process layer. Rubrics or process reward models can judge localization, evidence gathering, edit minimality, recovery, test discipline, budget use, and tool validity. SWE-TRACE uses issue-specific rubric criteria to rank and prune long software-engineering trajectories while preserving an outcome margin (Han et al., 2026). Process scores are most trustworthy for ordering actions within the same functional-outcome class.

Audit layer. Policy checks detect future-history access, edits to tests or grading scripts, network leakage, vacuous passes, hard-coded fixture answers, dangerous commands, timeout abuse, and environment corruption. This layer should be adversarial: a candidate solution generator actively searches for shortcuts that the verifier mistakenly rewards.

A practical reward composes the layers lexicographically:

$$R(\tau) = M\, r_{\text{func}}(\tau) + r_{\text{process}}(\tau) - \lambda C(\tau) - \gamma r^{-}_{\text{audit}}(\tau), \tag{7}$$

with M large enough that an incorrect but elegant trajectory cannot outrank a correct one, and a hard zero for critical policy violations. The formulation does not imply that all functional passes are equally good; it prevents a learned stylistic judge from overriding executable correctness.

4.5 Rubric construction

A rubric criterion should be *atomic*, *observable*, *requirement-linked*, and *discriminative*. "Uses a good approach" fails all four. "The parser preserves quoted commas in the second field, demonstrated by a named test or equivalent execution evidence" is auditable. Hierarchical rubrics can assign parent weights to major requirements and child criteria to observable evidence, as illustrated at much larger scale by PaperBench (Starace et al., 2025).

Rubrics may be generated from the requirement graph, the gold patch, tests, and code review, but they must be validated against independent alternatives. A minimum verifier-validation set contains:

- the original buggy state, which should fail;
- at least one reference solution, which should pass;
- behaviorally correct alternative solutions, which should also pass;
- near-miss patches and mutations, which should fail the relevant criterion;
- repeated runs under randomized seeds or fixtures, which should remain stable;
- adversarial attempts to modify, bypass, or infer the grader, which should be rejected.

Learned verifiers and LLM judges add semantic coverage but introduce calibration and distribution-shift risks. Execution and learned judgment are therefore complementary: tests anchor correctness, while rubrics densify feedback and expose partial progress. A reward model that ranks best-of-n candidates well is not automatically safe for RL, because an optimizing policy will visit inputs far outside the ranking dataset. Verifier evaluation must include calibration, adversarial robustness, and policy-induced distribution shift, not only static accuracy.

4.6 Generate–verify–repair as the canonical loop

A scalable builder should not discard every invalid task. It should diagnose failure and repair the relevant artifact:

1. Generate a structured task graph, environment recipe, verifier stack, and candidate solution.
2. Execute the buggy and solved states; collect build, test, state-delta, and coverage evidence.
3. Classify invalidity as prompt, environment, verifier, solution, novelty, or policy failure.
4. Repair only the implicated component, rebuild from a clean snapshot, and rerun all gates.
5. Freeze the accepted bundle, hash its artifacts, and record every generation and repair step in lineage.

Filtering is appropriate when repair would change the task's identity or create uncertain provenance. Repair is appropriate for deterministic setup bugs, incomplete fixtures, weak assertions, and wording that can be aligned without revealing the solution. The accepted bundle—not the generator's prose—is the unit passed downstream.

Synthesis principle

The order of trust should usually be: establish an executable behavior and a red-teamed verifier, validate at least one solution and several near misses, then render a natural-language task whose claims are no broader than that evidence. This is closer to test-driven data construction than to unconstrained prompt generation.

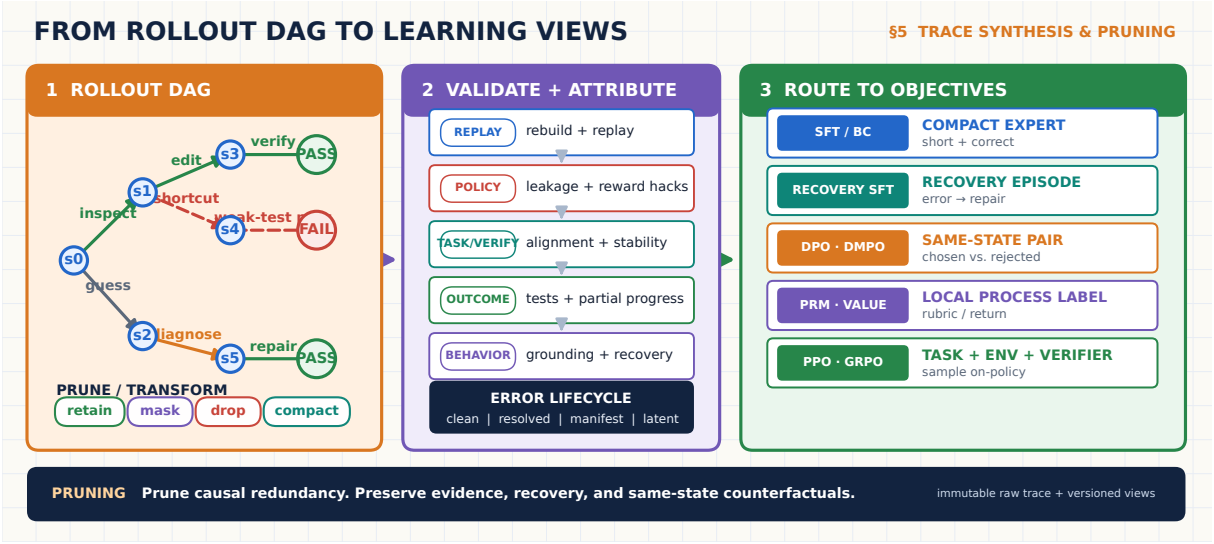

Figure 4: From raw rollouts to causally useful learning views. A rollout DAG retains shared states, divergent actions, verifier outcomes, recovery paths, state deltas, and cost. Replay, policy, task/verifier, outcome, and behavioral gates then route derived views to compact or recovery SFT, same-state preference learning, process/value learning, or on-policy RL. Pruning removes causal redundancy while preserving grounded evidence and counterfactual structure.

5 Trajectory Synthesis and Pruning

Once a task bundle is executable, a policy can generate interaction experience. Raw experience is abundant but heterogeneous: a passing trace may exploit a weak verifier, a failing trace may be one step from a correct repair, and two traces may differ only in repeated file reads. The central trace problem is therefore not "keep success, drop failure." It is to recover which decisions changed the reachable outcome, preserve useful recovery and counterfactual evidence, and route each fragment to an objective that can use it.

5.1 The raw object is a rollout graph

Sampling K independent trajectories discards their shared structure. Instead, represent collection for task b as a directed acyclic graph

$$\mathcal{T}_b = (N, E), \qquad n_i = (\sigma_i, h_i, o_i, c_i), \quad e_{ij} = (a_{ij}, \Delta_{ij}, z_{ij}), \tag{8}$$

where σ_i is an environment checkpoint or content-addressed state fingerprint, h_i is the visible context, o_i the most recent observation, and c_i accumulated cost. An edge records an agent action, its state delta Δ_{ij}, and local execution evidence z_{ij}. Terminal nodes carry the verifier vector and policy-audit results.

The graph can be formed deliberately by beam search, tree search, shared-prefix resampling, multi-agent debate, or teacher correction. It can also be reconstructed by merging independently sampled traces whose environment state and visible history are equivalent. State equivalence must be stronger than text similarity: two nodes with similar dialogue but different uncommitted file edits are not the same decision point.

5.2 Where trajectories come from

Human and repository traces. Pull-request chains, code review, CI history, issue discussion, and developer corrections provide high-fidelity sequences of intent and state transition. They are rarely directly replayable: timestamps, missing local commands, private context, and changes to external services must be reconstructed. daVinci-Agency shows the value of fewer but information-dense PR chains containing dependent changes across stages (Jiang et al., 2026). Human traces should be viewed as evidence of useful decisions, not as exact terminal transcripts to imitate.

Teacher rollouts. A strong model sampled under multiple prompts and harnesses is the dominant source of successful demonstrations. Best-of-n, beam search, and test-time scaling raise yield and naturally

produce rejected branches. Teacher data is off-policy for the student and inherits teacher style, blind spots, contamination, and scaffold dependence. Recording the teacher checkpoint, decoding parameters, reasoning mode, and harness is therefore mandatory.

Student and self-play rollouts. On-policy data exposes states the current model actually visits and avoids the covariate shift of pure behavior cloning. It is essential for online RL and useful for DAgger-like correction. Its drawback is low initial success, repeated failure modes, and rapidly changing data distributions. A replay store should record policy version and relabel old experience when the verifier changes.

Search and multi-agent rollouts. Tree search, sibling resampling, critic–actor loops, and specialized agents produce state-matched alternatives. These are more informative than unrelated successful and failed trajectories because their divergence can often be localized. Search policy and compute budget are part of lineage: an expert path discovered after hundreds of branches does not imply that the base policy could generate it unaided.

5.3 Validation before optimization

Trace curation should apply gates in an order that prevents cheap but misleading preferences:

1. **Replay gate**: rebuild the start state and replay tool actions or their recorded state deltas.
2. **Policy gate**: reject leakage, grader modification, test deletion, future-history access, sandbox escape, and other reward hacks.
3. **Task gate**: confirm that environment and verifier still represent the task and are not flaky.
4. **Outcome gate**: compute functional, regression, partial-progress, and cost vectors.
5. **Behavior gate**: score grounding, adaptability, persistence, refusal, destructive actions, and protocol compliance.
6. **Learnability and diversity gate**: balance repositories, languages, task types, failure modes, harnesses, and difficulty for the target policy.

Length and apparent efficiency should be considered last. Otherwise the selected demonstration may be the shortest trajectory only because it found a shortcut in the verifier. LiteCoder-Terminal's behavior filtering—including adaptability, groundedness, persistence, and refusal—illustrates why a functional pass alone is an incomplete trace label (Peng et al., 2026).

5.4 Four levels of pruning

Trajectory-level selection. The simplest policy keeps traces that pass and are non-cheating. This is suitable for rejection-sampling fine-tuning but overrepresents easy tasks and discards near-success. Better selection assigns a route: compact expert, recovery, preference candidate, process-label candidate, negative safety example, environment failure, or discard. Weighting can correct repository and difficulty imbalance without duplicating text.

Branch and segment pruning. Segments should align with semantic work: localization, hypothesis, edit, test, diagnosis, rollback, and final verification. A branch that repeats an unchanged test is redundant; a branch that runs a targeted test, observes a counterexample, and changes the next edit carries causal evidence even if it temporarily moves away from success. SWE-TRACE uses stepwise candidate generation and rubric-guided selection to construct shorter successful paths and hard negatives for a process reward model (Han et al., 2026).

Step-level pruning. Heuristics include no state change, repeated command over an unchanged state, invalid tool call, duplicate observation, test delta, code-localization overlap, patch applicability, and information gain. Learned process scores can rank siblings, but the most defensible test is counterfactual replay: delete or replace action a_t, restore the checkpoint, replay or regenerate the suffix, and compare the verifier vector. If the same terminal behavior remains reachable at no higher cost, the action is causally redundant. This is expensive and best applied to uncertain high-value segments.

Token-level masking and context compaction. Loss masking can keep an error and its observation in context while avoiding imitation of the error action. Tool syntax, paths, diffs, stack traces, test names, exit codes, and explicit constraints should remain exact. Free-form reasoning and verbose output can be summarized, but summary provenance and access to the underlying artifact should be preserved. Learned context compaction such as CompactionRL moves this decision inside policy optimization; it is related to, but distinct from, offline data pruning (Li et al., 2026a). Inference-time repository pruning methods similarly reduce the visible context rather than change the stored training trajectory (SWE-Pruner Team, 2026; Wang et al., 2026).

5.5 Failure attribution: do not label every detour negative

Final outcome does not determine the sign of every action. A clean trace contains no material error. A costly trace makes an error but resolves it. A manifest failure contains an active error that causes the terminal failure. A latent error remains hidden and may or may not affect the observed result. TrajDebug formalizes an error lifecycle that distinguishes these cases and identifies critical failures rather than marking all actions in a failed trajectory as negative (Qi et al., 2026).

This distinction yields three curation rules:

1. Preserve **costly resolution** as recovery experience: the error, diagnostic evidence, rollback, and corrected action form a coherent positive episode.
2. Turn **manifest active errors** into shared-state negative branches or repair targets, beginning at the earliest attributable divergence.
3. Quarantine **latent or environment-confounded errors** until replay or intervention establishes causality; do not broadcast a terminal label backward by default.

Repair must operate on environment state, not text alone. Replacing an early action while keeping the old observations creates an impossible trajectory. A valid repaired trace restores the pre-divergence checkpoint, executes the replacement, and regenerates every downstream observation and action. Similarly, relabeling after verifier updates should create a new derived view while preserving the original label and verifier version.

5.6 Derived learning products

Two parallel views are especially useful. **Compact expert** traces teach a direct, economical workflow. **Recovery experience** teaches diagnosis, rollback, and self-correction under states the agent is likely to visit. Training only on the first creates a model that appears polished under teacher states but may be brittle after its own mistake; training naively on the second may imitate mistakes. Segment labels and partial loss masks resolve the tension.

5.7 What should be measured

Trace datasets should report more than token count and pass rate. Recommended measures include valid replay rate; success and partial-progress distribution; environment and verifier failure rates; teacher/student/harness mix; unique repository and task lineage; state-changing actions per thousand tokens; test and edit counts; recovery-episode frequency; critical-error coverage; same-state preference-pair count; cost to generate one accepted product; and retention under each pruning stage.

Table 4: Task synthesis paradigms. The first trusted artifact shapes both realism and bias; no paradigm dominates across all quality dimensions.

Paradigm	Construction	Examples	Strength	Dominant risk
Issue/PR-first	reconstruct pre-fix commit, issue, developer patch and tests	SWE-bench; SWE-Gym; SWE-Factory	authentic intent and maintenance distribution	history leakage; setup failure; issue–test–patch mismatch (Jimenez et al., 2024; Pan et al., 2024; Guo et al., 2025)
Commit/patch-first	select a change, back-translate a task, generate or recover tests	R2E-Gym	scales beyond well-written issues; controls changed scope	prompt may reveal the implementation or invent intent (Jain et al., 2025)
Repo/test-first mutation	establish a passing project, inject a rule/AST/LLM bug, describe it	SWE-smith	one environment yields many verifiable defects	mutations differ from organic bugs; easy local artifacts (Yang et al., 2025)
Test-conditioned	select covered behavior, synthesize defect and issue around execution evidence	SWE-TRACE	task, localization target and verifier are tightly coupled	limited by current test coverage; verifier monoculture (Han et al., 2026)
Issue-free evidence	expose failing tests or state while suppressing a noisy issue	SWE-Fuse	teaches independent debugging from execution feedback	loses user intent if used alone; should be mixed (Wen et al., 2026)
Skill/meta-task-first	jointly generate instruction, files/container, verifier, guideline and solution	Terminal-World; Meta-Task	supports domains without historical PRs; generation and repair share a loop	teacher and seed skills bound diversity and validity (Cheng et al., 2026; Pan et al., 2026)
Recursive curriculum	expand validated tasks into harder solutions, requirements and verifiers	RST	difficulty can grow in executable work rather than prompt length	recursive drift, duplicates and same-model verifier bias (Li et al., 2026b)

Table 5: Trace pruning operations and their appropriate evidence. Pruning should produce a derived view while retaining the immutable raw trace and transformation lineage.

Level	Operation	Evidence	Main failure mode
Trajectory	keep, reject, weight, or route complete rollouts	verifier vector, policy audit, cost, diversity	outcome label hides which decisions were useful
Branch / segment	select a path; retain recovery episode; remove dead branch	shared checkpoint, segment return, state delta	deleting diagnostic detours that enable later repair
Step / action	choose among sibling actions; mask or replace one action	local validator, PRM, counterfactual replay	learned judge confuses plausible prose with causal utility
Token / context	loss masking, observation compaction, deduplication	role, tool boundary, exact artifacts, attention budget	corrupting commands, diffs, stack traces, or causal ordering

Table 6: A routing table from rollout-graph structures to learning artifacts. "Same state" refers to an equivalent environment checkpoint and visible context, not merely the same prompt.

Graph view	Data product	Downstream use	Required caveat
Shortest validated success	compact demonstration with action-token mask	SFT / behavior cloning	do not erase all evidence gathering or final verification
Resolved detour	error–evidence–repair segment	recovery SFT	mask erroneous action if direct imitation is harmful
Same-state siblings	chosen / rejected action or segment	DPO, DMPO, IPO	control length and ensure preference is causally local
Independent labeled traces	desirable / undesirable examples	KTO-style learning	weaker counterfactual than a matched pair
Multi-action state	action, local rubric, downstream return	PRM or step-level value learning	judge must be calibrated and outcome-anchored
Raw task bundle	environment plus verifier	online PPO/GRPO/RLVR	isolate formal evaluation and monitor reward attacks
Achieved alternative goal	relabeled task and suffix	hindsight learning (emerging)	new task must be valid and lineage-preserving

We define a simple **credit density** diagnostic

$$\mathrm{CD}(\mathcal{T}) = \frac{\#\{\text{actions with outcome-, process-, or counterfactual evidence}\}}{\#\{\text{agent actions}\}}, \tag{9}$$

reported separately by label type. It does not claim that more labels are always better; it reveals when a very large trace corpus contains little localized supervision.

Pruning principle

Prune causal redundancy, not every detour. The ideal curated corpus contains both direct expert paths and grounded failure–recovery episodes, while state-matched branches provide the counterfactual supervision that a terminal pass bit cannot.

6 Post-Training with Executable Experience

Coding-agent post-training is often summarized as a progression from SFT to DPO to RL. This ordering conflates several independent choices. A method is better specified by four axes:

1. **Experience source**: human/PR, teacher, current policy, search, or synthetic environment;
2. **Supervision**: demonstration, binary outcome, pairwise preference, process score, or state transition;
3. **Policy relation**: offline/off-policy, iteratively refreshed, or online/on-policy;
4. **Credit granularity**: trajectory, semantic segment, turn/action, or token.

Reinforcement learning with verifiable rewards (RLVR) describes a reward regime—rewards come from executable or otherwise checkable outcomes. PPO, GRPO, and RLOO are optimization families. Rejection sampling is a data-selection operator. Keeping these layers separate makes comparisons and hybrid systems easier to reason about.

6.1 Supervised fine-tuning and behavior cloning

Given selected demonstrations τ, supervised fine-tuning minimizes next-token loss on chosen agent outputs:

$$\mathcal{L}_{\mathrm{SFT}}(\theta) = -\mathbb{E}_{\tau} \sum_{t \in \mathcal{M}(\tau)} w_t \log \pi_\theta(u_t \mid h_t), \tag{10}$$

where $\mathcal{M}$ masks system, user, observation, and harmful-error tokens as appropriate. Coding-agent SFT teaches tool syntax, edit protocols, repository navigation, debugging routines, and final verification. It is stable and data-efficient when demonstrations are high quality.

Its characteristic failure is state-distribution shift. The student eventually takes an action the teacher trace did not take, observes a new compiler or test failure, and has no demonstration for recovery. Three data interventions help:

- mix compact successful paths with explicitly annotated recovery episodes;
- collect student rollouts and ask a teacher to correct states the student actually visits, following a DAgger-like loop;
- augment tool and scaffold renderings so the model learns semantics rather than one surface protocol.

SWE-Gym showed that real issue tasks and successful trajectories can support supervised agent improvement (Pan et al., 2024). SWE-smith scaled environment-first synthetic defects and collected thousands of successful teacher trajectories for SFT (Yang et al., 2025). Klear AgentForge uses a much larger agentic SFT stage before multi-turn RL, illustrating the continued importance of protocol and workflow competence before sparse-reward optimization (Wang et al., 2025).

6.2 Rejection sampling and iterative self-training

Rejection-sampling fine-tuning generates n rollouts per task, applies functional and policy gates, and trains on selected successes. It converts an executable verifier into a demonstration generator without changing the SFT objective. Iteration can bootstrap a stronger policy that produces more and harder successes.

The method has three data biases. First, tasks on which the current policy never succeeds disappear, so the curriculum drifts toward easy instances. Second, all failed experience is discarded, even when it contains correct localization or a near-complete fix. Third, best-of-n demonstrations may reflect a search budget unavailable at deployment. Difficulty-aware sampling, recovery extraction, and explicit recording of the generation budget mitigate these biases.

6.3 Preference optimization

Preference methods learn from (τ^+, τ^-) or local $(a^+, a^- \mid s)$ comparisons. Direct Preference Optimization (DPO) removes explicit reward-model and policy-sampling loops by optimizing a contrastive objective relative to a reference policy (Rafailov et al., 2023). KTO accepts independently labeled desirable and undesirable examples, useful when strict pairing is unavailable (Ethayarajh et al., 2024). Multi-turn variants such as DMPO adjust sequence occupancy and length effects that become severe for long agent trajectories (Shi et al., 2024).

Coding traces make pair construction more important than the choice among closely related losses. Two entire trajectories from the same prompt may differ in length, tool affordances, and early state, so a final pass/fail preference provides diffuse credit. Strong pairs share a checkpoint and diverge at one action or semantic segment. They can be mined from search siblings, teacher correction of a student state, or a repair intervention. ETO iteratively converts agent exploration and failures into preference pairs, illustrating the value of unsuccessful experience when aligned with a better continuation (Song et al., 2024).

Preference labels should be lexicographic: policy-compliant functional success first; within the same outcome class, process quality and cost may decide. A shorter failed trace must not outrank a longer correct trace merely because an LLM judge prefers concision.

6.4 Process supervision and reward models

Sparse tests tell the agent whether the final state is acceptable but not where a long run went wrong. A process reward model estimates a score at actions or segments from rubric labels, paired siblings, downstream return, or synthetic critiques. SWE-TRACE constructs issue-specific rubrics and hard negatives, then uses the resulting PRM both for training and heuristic test-time pruning (Han et al., 2026).

Process supervision is useful in three roles: dense shaping during RL, branch selection during data synthesis or inference, and diagnostic attribution after failure. These roles demand different evaluation. A model that selects the best of several completed traces may not be calibrated enough to supply per-step RL reward; an optimizer will search for features that raise the score without improving the program. PRM evaluation should therefore include outcome consistency, same-state ranking, calibration by task type, adversarial patches, and policy-shift stress tests.

The reward stack in Equation 7 should preserve a hard correctness margin. Potential-based shaping or within-outcome ranking reduces the risk that dense process reward changes the optimal task policy. Learned reward should also be versioned: old trajectories may need relabeling after the PRM or rubric changes.

6.5 Online reinforcement learning with verifiable rewards

Online RL samples the current policy in task bundles and optimizes the expected reward

$$J(\theta) = \mathbb{E}_{b \sim D_{\text{train}},\, \tau \sim \pi_\theta(\cdot|b)} \left[R_{\mathcal{V}}(\tau) - \lambda C(\tau) \right]. \tag{11}$$

PPO uses a learned value baseline and clipped policy updates (Schulman et al., 2017). GRPO uses a group of rollouts for the same task to construct relative advantages without a separate critic (Shao et al., 2024).

Leave-one-out and asynchronous variants modify the baseline, sampling, and stale-policy correction. Coding environments make online data valuable because it follows the student's actual failure distribution, but expensive because each trajectory may require minutes of container execution and thousands of tokens.

Agentic RL adds several complications to standard language-model RL:

Sparse and degenerate groups. When every rollout fails, group-relative advantages vanish; when every rollout passes, they provide little direction. A data scheduler should adapt task difficulty, resample promising near-success states, or introduce outcome-anchored process reward. Filtering all timeouts as ordinary failures can also teach the wrong signal if the environment caused them.

Long-horizon credit. Broadcasting terminal reward to every token has high bias: localization, an edit, and redundant narration receive the same sign. A learned critic may reduce bias but introduces long-horizon value error. Shared-state micro-groups, trajectory-graph returns, and step-native formulations aim at the middle ground between whole-trajectory and token credit. In coding, semantic segments and critical divergent steps are especially natural because tools expose state changes.

Reward hacking. The policy can edit tests, alter configuration, exploit a timeout, print an expected string without producing the artifact, inspect Git history, or overfit known fixtures. Sandboxing, hidden behavioral tests, immutable graders, audit rules, and adversarial verifier generation are part of the RL algorithm in practice because they define the reward.

Policy–scaffold co-adaptation. RL trains behavior under a particular tool schema, system prompt, observation truncation, and context policy. Improvements may not transfer to another harness. Multi-harness training and held-out-scaffold evaluation distinguish tool semantics from interface memorization.

Throughput and asynchrony. Container startup, dependency caches, variable episode lengths, and flaky services dominate wall-clock cost. SkyRL-Agent emphasizes an asynchronous dispatcher and backend bridge for on-policy software-engineering RL (Cao et al., 2025). Single-rollout and asynchronous methods seek to reuse heterogeneous episodes without waiting for fixed groups, but must correct stale policies and carefully define returns (Hou et al., 2026). Environment throughput, reward latency, and accepted-experience yield should be reported alongside GPU training compute.

6.6 Coding-specific algorithmic patterns

Several recent systems illustrate how data and algorithms co-design each other.

SWE-TRACE. The pipeline starts from a large synthetic task pool, filters it to executable and useful tasks, constructs stepwise candidates, selects short successful paths, trains an issue-specific rubric PRM with hard negatives, and combines execution and process signals under group-relative RL (Han et al., 2026). Its key contribution for a data survey is the reuse of one rollout graph for SFT, PRM, RL, and test-time pruning.

SWE-Fuse. SWE-Fuse mixes issue-guided and issue-free experience, removes Git-based shortcuts, and uses execution evidence to train debugging rather than only instruction following (Wen et al., 2026). The design shows that prompts themselves can be ablated or mixed as a data variable, and that entropy or difficulty control is needed to keep RL informative.

RepoNavigator and tool abstraction. RepoNavigator compresses repository interaction into an execution-aware tool and trains the policy around that abstraction (Zhang et al., 2025). This reminds us that algorithmic improvement may actually be a change in $\mathcal{A}$: a better action interface reshapes the trajectory distribution, credit horizon, and data format.

Klear AgentForge and staged specialization. AgentForge performs large-scale agentic SFT, then multi-turn RL with outcome and turn-level rewards, followed by model merging for capability preservation (Wang et al., 2025). This staged recipe addresses a recurring tension: aggressive domain RL can improve software behavior while eroding general instruction following or tool use.

Compaction as a learned action. CompactionRL includes context compression inside reinforcement learning rather than treating it as an external heuristic (Li et al., 2026a). In the bundle view, the context manager is part of the harness and its summaries are intermediate data products. Training it jointly may extend the effective horizon, but summary faithfulness and recoverability must be evaluated.

6.7 A practical staged recipe

A mature coding-agent pipeline typically follows:

$$\begin{aligned}&\text{repository / PR mid-training} \rightarrow \text{compact + recovery SFT}\\ &\rightarrow \text{shared-state preference learning (optional)} \rightarrow \text{online RLVR on disjoint arenas}\\ &\rightarrow \text{hard verifier + calibrated PRM} \rightarrow \text{failure mining and verifier red-teaming.}\end{aligned}$$

SFT establishes the action language and tool protocol; preference learning exploits high-quality counterfactual pairs when available; online RL follows the deployed state distribution; process signals densify but do not override correctness; and fresh failures update the training task pool rather than reveal formal evaluation items.

Algorithm principle

There is no data-independent winner among SFT, preference optimization, and RL. The decisive question is whether the available artifact carries demonstration, counterfactual, process, or on-policy outcome information at the granularity the objective assumes. Algorithm names cannot compensate for a mismatched label.

7 End-to-End Data-System Design

The previous sections separate task, experience, and learning artifacts. Real systems must connect them without collapsing evaluation into training or allowing one verifier to dominate the flywheel. This section distills recurring design patterns and anti-patterns.

7.1 Four pipeline archetypes

A. Real-issue distillation. A collector mines issues and pull requests, reconstructs the pre-fix environment, validates tests, samples a strong teacher, and selects successful trajectories for SFT. SWE-Gym and SWE-Factory sit near this archetype (Pan et al., 2024; Guo et al., 2025). It offers high realism and low task-authoring cost after reconstruction, but public history creates contamination and the teacher may rarely solve the hardest tasks.

B. Environment-first mutation. A builder validates a repository once, injects many testable defects, synthesizes prompts, and samples trajectories. SWE-smith is the clearest example (Yang et al., 2025). It provides scale and exact defect control; robustness depends on mutation diversity, linguistic naturalness, and evidence that learning transfers to organic maintenance.

C. Generative skill worlds. An agent or template jointly creates a task, files/container, verifier, guideline, and reference solution, then repairs invalid bundles. Terminal-World and LiteCoder-Terminal instantiate this pattern (Cheng et al., 2026; Peng et al., 2026). It reaches domains without a repository history and produces self-contained arenas. Its risk is generator monoculture: tasks, solutions, rubrics, and failures may all reflect one teacher's worldview.

D. Online failure flywheel. A deployed or training policy samples diverse arenas; failures are clustered by cause; builders generate or retrieve targeted tasks; rollout graphs produce SFT, preference, and PRM views; online RL refreshes the failure distribution. This pattern is implicit in systems such as SWE-TRACE and SWE-Fuse (Han et al., 2026; Wen et al., 2026). It tracks the current policy but requires strict firewalls so formal evaluation failures do not leak as exact training items.

7.2 A multi-view data lake

Raw artifacts should be immutable and content-addressed. Derived views are materialized through versioned transformations rather than overwriting the source. A practical schema has four stores:

- a **task registry** of repository snapshots, environment recipes, prompts, verifiers, access policies, and split lineage;
- a **rollout store** of actions, observations, state fingerprints, checkpoints, costs, policy and harness versions;
- an **annotation store** of outcome vectors, rubric scores, preferences, failure lifecycle, and audit flags, each tied to a verifier version;
- a **training-view registry** that records selection, pruning, masking, weighting, mixing, and export hashes.

This architecture makes several corrections possible without regenerating everything. A repaired test suite can relabel old terminal states. A new policy-audit rule can quarantine traces that edited a grader. A different SFT view can retain recovery episodes while an efficiency-focused view selects only compact paths. A data card can report exactly which raw bundles and transformations produced a checkpoint.

7.3 Split before synthesis

Lineage leakage often enters before task text exists. Suppose the same repository appears in training and evaluation. A synthetic bug in one file may expose conventions, dependency structure, helper functions, and tests needed by an evaluation issue in another. Text deduplication will not catch this. Splits should therefore be assigned at the earliest trusted ancestor, preferably repository or organization, and inherited by every prompt, mutation, solution, verifier, trajectory, summary, and preference pair.

A strong evaluation program uses several orthogonal holdouts:

- **repository/organization holdout** for unseen code and conventions;
- **time holdout** for freshness after the model and data cutoff;
- **task-type holdout** for transfer from bug repair to features, refactoring, or test writing;
- **language/build-system holdout** for interface and ecosystem transfer;
- **generator/verifier holdout** for synthetic monoculture and reward overfitting;
- **scaffold holdout** for semantics beyond one tool protocol.

Not every experiment needs all axes, but random task splits alone should not support broad generalization claims.

7.4 Policy and verifier co-evolution

A fixed verifier becomes an optimization target. The more capable and more frequently sampled the policy, the more likely it is to discover brittle assumptions. Data flywheels should therefore allocate budget to verifier red-teaming:

1. sample high-reward and disagreement cases from the current policy;
2. generate alternative solutions and adversarial shortcuts;
3. compare test, rubric, static-analysis, and human judgments;
4. add hidden behavioral checks or repair requirements;
5. relabel the replay buffer under a new verifier version;

6. retain a stable evaluation anchor to distinguish model change from metric change.

Verifier diversity matters. Independent implementations, metamorphic tests, randomized fixtures, static properties, security checks, and semantic rubrics fail differently. Agreement among correlated LLM judges is not equivalent to independent evidence. For high-stakes or high-reward examples, human review should focus on disagreement and adversarial frontier cases rather than sample uniformly.

7.5 Curriculum as data allocation

Difficulty is dynamic: an environment that yields useful group variation for one checkpoint may be trivial or impossible for another. A scheduler can estimate a task's current success probability $\hat{p}_b$, uncertainty, novelty, failure-cluster importance, rollout cost, and verifier confidence. One allocation objective is

$$w(b) \propto \underbrace{\hat{p}_b(1-\hat{p}_b)}_{\text{learning signal}} \cdot \underbrace{u_b}_{\text{uncertainty/novelty}} \cdot \underbrace{v_b}_{\text{verifier confidence}} \Big/ \underbrace{c_b}_{\text{expected cost}} . \tag{12}$$

This is a scheduling heuristic, not a universal training objective. Some zero-success tasks should be retained for frontier expansion and teacher repair; some high-success tasks should remain as regression anchors. The important point is that "dataset mixture" becomes an adaptive policy over executable bundles, not a fixed token ratio.

7.6 Evaluation of the data engine

The data pipeline itself should be evaluated through ablations and transfer:

- **source ablation**: real issues versus mutations versus generated worlds;
- **bundle ablation**: prompt only versus environment; tests versus verifier stack; one versus multiple harnesses;
- **trace ablation**: compact success, recovery, failed hard negatives, and raw unpruned data;
- **objective ablation**: matched data under SFT, preference learning, and online RL;
- **holdout transfer**: unseen repository, time, language, task type, scaffold, and verifier;
- **cost accounting**: task-build success, accepted experience per dollar, environment utilization, and end-to-end wall clock.

Matched data is crucial. Comparing an SFT model trained on clean teacher successes with an RL model sampled on harder student tasks does not isolate the algorithm. Conversely, holding tasks fixed while changing the scaffold may measure an interface effect. The bundle formalism gives each ablation an explicit field to hold constant.

7.7 Anti-patterns

Prompt-only releases. A list of issues without reproducible repository snapshots and verifiers cannot support comparable agent training or evaluation.

Gold-patch verifiers. Exact-match or structure checks against one reference implementation reject legitimate alternatives and teach imitation rather than behavior.

Passing-only trace stores. They hide the current policy's failure distribution, make credit diffuse, and remove recovery supervision.

Outcome broadcast. Copying one pass/fail label to every step creates false positives in successful traces and false negatives in failed traces.

Benchmark cannibalization. Repeatedly training on a celebrated benchmark and continuing to call it a held-out evaluation turns leaderboard progress into memorization of a task family.

Unversioned harnesses. Changing tool descriptions, context management, retry budgets, or terminal wrappers while attributing gains only to the model makes scores uninterpretable.

System principle

Treat every artifact as versioned software and every transformation as a lineage edge. This makes it possible to repair verifiers, relabel traces, construct multiple learning views, and audit contamination without pretending that a dataset is a static bag of examples.

8 Open Problems

8.1 Measuring validity under multiple correct implementations

Coding admits many behaviorally equivalent patches. Hidden tests reduce implementation overfitting but rarely cover the full requirement. Learned judges add semantic reach but can prefer surface style, and formal specifications are expensive. The open problem is a scalable verifier that balances *faithfulness* to intent, *coverage* of relevant behavior, and *robustness* under optimization. Promising directions combine property-based and metamorphic tests, differential execution, mutation adequacy, static invariants, independent judges, and targeted human review of disagreement.

8.2 Benchmark freshness without losing comparability

Public static tasks allow reproducibility and rapid method development, but contamination and scaffold overfitting accumulate. Private tasks protect the boundary but reduce auditability. Rolling tasks are fresh but change every comparison. A durable evaluation ecosystem needs a three-tier design: public development tasks with complete artifacts; versioned, auditable anchor tasks; and access-controlled rolling tasks with delayed release. Cryptographic commitments, evaluation logs, and scheduled retirement could make freshness claims falsifiable rather than rhetorical.

8.3 Environment rot and semantic replay

Pinned dependencies do not guarantee future replay when package registries, compilers, base images, external services, or hardware change. Containers preserve files but not always kernels, accelerators, clocks, DNS, or remote APIs. Research is needed on semantic environment snapshots, deterministic service mocks, content-addressed dependency archives, and replay confidence intervals. Environment failure should be modeled as missing or noisy reward, not as policy failure.

8.4 The synthetic–real gap

Mutation and generative-world pipelines offer scale and controllability, while real issues offer authentic intent and organizational context. The field lacks strong causal evidence about which synthetic factors transfer: bug operator, repository realism, prompt style, dependency depth, verifier diversity, or interaction horizon. Factorial data generation and held-out real-work transfer are more informative than another aggregate scale curve. Synthetic data should be evaluated on the capabilities it changes, not on its resemblance to developer prose alone.

8.5 Causal credit at the right granularity

Terminal reward is too coarse; token reward is often too noisy and semantically arbitrary. Shared-state branches and semantic segments offer a promising middle layer, but require checkpoints and expensive interventions. Research questions include how to merge nearly equivalent states, how to estimate the value of an information-gathering action whose payoff is delayed, and how to identify the first critical divergence without an oracle. Counterfactual replay, trajectory graphs, and failure-lifecycle annotations are fertile data structures for this work.

8.6 Learning from failure without learning to fail

Failure traces contain localization, negative evidence, and recovery, but naive SFT imitates their mistakes. Preference learning needs a better continuation; outcome broadcast mislabels useful prefixes. The challenge is to extract causal failure–repair episodes and train diagnosis, rollback, and exploration without increasing destructive behavior. Partial loss masks, corrective teacher actions at student states, hindsight tasks, and local process labels deserve systematic comparison.

8.7 Harness generalization

Models learn under a particular action interface and context manager. Tool consolidation can shorten the credit horizon, while verbose general tools expose more transferable semantics. There is no standard way to separate model capability from scaffold competence. Benchmarks should include tool-schema perturbations, equivalent alternative harnesses, held-out editors or shells, and controlled context policies. Training data may need a canonical intermediate action representation that renders into multiple interfaces while preserving environment evidence.

8.8 Verifier co-evolution without moving the goalposts

RL finds verifier weaknesses, so a training verifier must evolve. But continuously changing it can hide regressions and make curves incomparable. The field needs protocols that maintain a stable anchor reward, version new adversarial tests, relabel replay buffers, and distinguish genuine policy improvement from stricter measurement. Verifier changes should be reported like dataset and code changes, including which checkpoints were trained against which revision.

8.9 Beyond functional correctness

Tests mostly capture functional behavior. Software engineering also requires security, performance, maintainability, API stability, licensing compliance, and alignment with product intent. These objectives conflict and are hard to reduce to one scalar. Vector rewards, constraint-based optimization, Pareto reporting, and rubric hierarchies may help. Evaluation must avoid allowing a subjective maintainability judge to erase hard correctness, while still penalizing patches that pass narrowly and damage the system.

8.10 Data rights, privacy, and attribution

Repository code, issues, logs, and pull requests carry licenses, contributor identities, secrets, and organizational information. Synthetic descendants do not erase upstream obligations. Trajectory logs may reveal credentials, proprietary paths, or private test behavior. An executable bundle needs machine-readable licenses, contributor and source lineage, secret scanning, deletion propagation, and access tiers. Training on private environments raises an additional question: whether gradients or released traces can leak protected code.

8.11 Agent-generated training data governance

PostTrainBench illustrates a broader future in which agents choose and generate data for other models (PostTrainBench Team, 2026). Governance must specify which web sources, APIs, teacher models, benchmarks, and prior traces are permitted; how the agent records data provenance; and how contaminated runs are detected. The agent's research freedom and the evaluation firewall are in tension. Sandboxed data acquisition, signed datasets, model-identity checks, and automatic trace audits may become standard components of post-training environments.

8.12 Cost and environmental efficiency

An executable task may require a container build, several long rollouts, a verifier ensemble, and counterfactual replay. Accepted-data cost can dwarf token-training cost. Papers should report build yield, rollout success, replay rate, accepted products per task, CPU/GPU hours, model-call cost, and caching assumptions. Research on learned simulators and execution-free verifiers can reduce cost, but every surrogate introduces a simulation or reward gap that must be measured on real execution.

8.13 A common evidence standard

The literature lacks a standard data card that connects a released agent checkpoint to its executable training lineage. A minimal claim should identify task sources and cutoffs; repository and organization overlap; environment versions; verifier construction and audits; teacher and student policies; harnesses; raw-to-derived trace transformations; learning objectives; evaluation access and attempts; and known contamination. Appendix C proposes a starting checklist.

Research agenda

The next scaling frontier is not simply more trajectories. It is higher task–environment–verifier agreement, fresher and better isolated evaluation, denser causal credit, explicit failure recovery, verifier diversity, and lineage strong enough to audit an agent-generated data flywheel.

9 Conclusion

Coding agents make the structure of agent data unusually visible. Their actions alter repositories and machines; their feedback arrives through builds, tests, and users; and their output can often be executed. These properties create a scalable learning signal, but only when the task, environment, verifier, trajectory, and lineage agree.

This survey argued that the correct unit is an *executable experience bundle*, not a prompt–response pair. Under this view, benchmarks define held-out task bundles and their measurement boundaries; prompt synthesis becomes joint task–environment–verifier construction; trace synthesis becomes search and attribution over a rollout graph; and SFT, preference learning, process supervision, and online RL become different consumers of derived experience. The same view explains why a passing trace may be invalid, why a failed trace may be valuable, why a model score cannot be separated from its scaffold and benchmark revision, and why evaluation data ceases to be held out when it enters the training flywheel.

Three transitions summarize the field. Coding-agent data is moving from static code to executable tasks, from single successful demonstrations to counterfactual and recovery-rich experience, and from fixed tests to verification systems that must evolve with the policy. Progress will depend less on nominal sample count than on executable validity, causal credit density, cross-harness transfer, and trustworthy provenance. The practical goal is a data engine that can grow capability from new experience while retaining a clean, auditable boundary around the evidence used to claim that capability.

Table 7: Post-training families viewed as consumers of executable experience. The same optimizer can behave very differently under another data source, verifier, or credit granularity.

Family	Required artifact	Typical credit	Best use	Coding-specific risk
SFT / BC	validated demonstrations and masks	token under a trajectory	protocol, tools, direct workflows	teacher-state shift; imitation of redundant exploration
Rejection-sampling FT	multiple rollouts plus hard verifier	selected trajectory	inexpensive self-training and distillation	easy-task bias; failure information lost
DPO / DMPO / IPO	chosen–rejected pair, reference policy	trajectory or matched segment	offline use of counterfactual branches	weak pairs and length bias create diffuse preference
KTO-like	independent desirable / undesirable labels	trajectory or segment	large buffers lacking exact pairs	no same-state counterfactual; label confounding
PRM / rubric learning	step/segment comparisons or criteria	step or semantic segment	dense guidance, search, diagnosis	judge hacking, miscalibration under policy shift
PPO	on-policy rollouts, reward, value estimates	token/turn advantage	flexible online optimization	costly critic; long-horizon value error
GRPO / RLOO	groups of on-policy rollouts and reward	group-relative trajectory, then broadcast	critic-free RLVR at scale	all-zero/all-one groups; rollout cost; blame diffusion

Table 8: End-to-end archetypes expose different bottlenecks. "Primary audit" is the check most likely to catch systematic invalidity before scale amplifies it.

Archetype	Trusted seed	Scales through	Primary bottleneck	Primary audit
Real-issue distillation	issue, PR, repository history	automated reconstruction + teacher sampling	setup yield and public leakage	independent prompt–test review; temporal isolation
Environment-first mutation	buildable repository and tests	many injected defects per environment	synthetic–real transfer	mutation holdout and real-issue transfer
Generative skill worlds	seed skills or task templates	joint generation and repair	same-generator bias	independent verifier and cross-generator evaluation
Online failure flywheel	current-policy failures	targeted task generation and on-policy sampling	instability and eval leakage	lineage firewall; frozen external evaluations

A Survey Methodology and Evidence Policy

This manuscript is a structured narrative review with a frozen evidence date of 18 August 2026. The starting vocabulary combined *coding agent*, *software engineering agent*, *repository-level code*, *SWE-bench*, *terminal agent*, *task/environment synthesis*, *trajectory synthesis*, *trace pruning*, *agentic SFT*, *preference optimization*, *process reward model*, *RLVR*, and *post-training agent*. We followed citations and related-work links from landmark executable benchmarks and training-environment papers, then added 2025–2026 work on long-horizon terminal environments, rollout-graph supervision, context compaction, and automated post-training.

Primary sources were preferred for factual claims: papers and their released repositories for dataset scale and method design; benchmark documentation for current revisions; and first-party audit posts for benchmark-quality changes. Claims about SWE-bench Verified and SWE-bench Pro use the 2026 OpenAI audit reports rather than older leaderboard descriptions (OpenAI, 2026b,a). Numerical values are tied to a named paper or release because actively maintained benchmarks drift. Frontier 2026 systems are treated as preprints or technical reports unless an archival venue is explicitly known.

We included a work when it contributed at least one of the following: (i) an executable coding or terminal benchmark; (ii) a scalable construction pipeline for tasks, environments, verifiers, or trajectories; (iii) a post-training method evaluated in interactive software environments; (iv) a trace-analysis or pruning method with a direct implication for coding-agent data; or (v) a benchmark audit that changed how reported results should be interpreted. We excluded ordinary function generation from the main taxonomy unless it clarified the boundary with agents, and excluded generic agent frameworks without a data or evaluation contribution.

The review uses three evidence levels. **Established** refers to widely used peer-reviewed benchmarks or methods with multiple independent follow-ups. **Emerging** refers to 2025–2026 preprints with released artifacts or detailed executable evidence. **Proposed** denotes the survey's synthesis, such as the executable experience bundle, credit-density diagnostic, and lifecycle design principles. These levels prevent a fresh scale claim from carrying the same weight as a repeated benchmark audit.

This is not a systematic meta-analysis. Scores from different models and scaffolds are not aggregated because tool interfaces, budgets, benchmark revisions, and access policies are not commensurate. The emphasis is on data structure, evidence boundaries, and design trade-offs. A future living version should publish a machine-readable inventory with per-claim source locators, revision dates, and independent citation audits.

B Extended Landscape

Table 9 provides a compact routing index. It is not a leaderboard; entries are grouped by the primary artifact they contribute.

Table 9: Extended landscape of representative coding-agent data work.

Work	Year	Primary role	Task unit / source	Distinctive data contribution
SWE-bench	2023	evaluation	real issue–PR	repository snapshots with F2P/P2P executable tests (Jimenez et al., 2024)
SWE-Gym	2024	training arena	real issue–PR	2,438 tasks plus successful trajectories for agent training (Pan et al., 2024)
R2E-Gym	2025	training arena	commit back-translation	more than 8.1K tasks; executable and execution-free verification (Jain et al., 2025)

Continued on next page

Work	Year	Primary role	Task unit / source	Distinctive data contribution
SWE-smith	2025	task / trace synthesis	injected bugs in 128 repos	50K task scale and 5,016 successful teacher trajectories in the paper release (Yang et al., 2025)
SWE-Factory	2025	task synthesis	real issues	automated test restoration, setup, and fail-to-pass construction (Guo et al., 2025)
Multi-SWE-bench	2025	evaluation / RL	multilingual issues	expert-validated cross-language tasks and Multi-SWE-RL (Zan et al., 2025)
SWE-rebench	2025	collection / evaluation	automated issue mining	time-aware, repeatable collection across thousands of repositories (Badertdinov et al., 2025)
SWE-rebench V2	2026	RL environment	multilingual automated mining	executable task supply at broader language and repository scale (Badertdinov et al., 2026)
SWE-bench-Live	2025	rolling evaluation	recent GitHub issues	post-cutoff, continuously updated task collection (Microsoft Research and collaborators, 2025)
SWE-bench Pro	2025	evaluation	public, held-out, commercial issues	longer multi-file tasks and access tiers; public split later audited (Deng et al., 2025; OpenAI, 2026a)
SWE-Lancer	2025	professional evaluation	paid freelance work	executable IC work plus managerial decision tasks (OpenAI, 2025)
SWE-EVO	2025	long-horizon evaluation	project releases	release-scale changes with partial fix rate (Le et al., 2025)
Terminal-Bench 2	2026	terminal evaluation	authored terminal tasks	containerized final-state verification through Harbor (Merrill et al., 2026)
Terminal-World	2026	environment synthesis	generated agent skills	task, environment, verifier, and guideline as one primitive (Cheng et al., 2026)
LiteCoder-Terminal	2026	task / trace synthesis	generated terminal worlds	adversarial verifier and behavior-based filtering (Peng et al., 2026)
Meta-Task	2026	meta synthesis	task creation as a terminal task	generate, execute, repair, and filter task-building traces (Pan et al., 2026)
RST	2026	recursive synthesis	validated seed tasks	recursive growth with executable instruction–verifier validation (Li et al., 2026b)
SWE-Fuse	2026	training	issue-guided and issue-free traces	mixes intent with execution-only debugging and removes Git shortcuts (Wen et al., 2026)

Continued on next page

Work	Year	Primary role	Task unit / source	Distinctive data contribution
SWE-TRACE	2026	trace / PRM / RL	synthetic executable tasks	oracle path construction, rubric PRM, GRPO, and test-time pruning (Han et al., 2026)
TrajDebug	2026	failure attribution	long agent trajectories	error lifecycle separating costly, manifest, and latent failures (Qi et al., 2026)
Klear AgentForge	2025	SFT / RL	large agentic mixture	staged agentic SFT, multi-turn RL, and capability-preserving merge (Wang et al., 2025)
SkyRL-Agent	2025	online RL	R2E-Gym tasks	asynchronous dispatcher and tool-centric on-policy training (Cao et al., 2025)
RepoNavigator	2025	tool / RL	repository localization	execution-aware tool changes action and credit space (Zhang et al., 2025)
CompactionRL	2026	context / RL	long interactive traces	learns when and how to compact context during agent RL (Li et al., 2026a)
PaperBench	2025	AI R&D evaluation	paper reproduction	hierarchical artifact rubric with 8,316 nodes (Starace et al., 2025)
PostTrainBench	2026	post-training evaluation	model-improvement runs	agent controls data, training code, compute, and final checkpoint (PostTrainBench Team, 2026)

C Reporting and Release Checklists

The following checklists translate the survey into auditable release fields. They are intended as minimum documentation, not a substitute for benchmark-specific analysis.

C.1 Benchmark checklist

Table 10: Minimum record for a coding-agent benchmark result.

Category	Report
Identity	benchmark name, exact revision, task count, release date, current errata
System	model checkpoint, scaffold, system prompt, tool schema, context and compaction policy
Budget	attempts, tokens, wall clock, compute, parallelism, tool and network limits
Task source	repository/organization/time/language/task type, authoring or mining procedure
Environment	image and dependency digests, services, hardware, reset policy, replay success
Verifier	visible and hidden layers, F2P/P2P policy, rubric/judge versions, audit checks
Metrics	resolved and partial progress, pass@k, consistency, cost, failure denominator
Isolation	model/data cutoff, repository and lineage overlap, web/Git-history policy
Uncertainty	number of runs, confidence interval or SEM, environment-invalid exclusions

C.2 Synthetic task checklist

Table 11: Validity gates for a synthesized executable task.

Gate	Evidence to retain
Provenance	seed snapshot, license, generator and prompts, ancestry and split
Prompt	structured requirement, rendered variants, ambiguity review, leakage check
Environment	build log, baseline runs, state fingerprint, reset and isolation checks
Verifier	buggy failure, reference success, alternative success, near-miss failure, repeated stability
Difficulty	baseline pass rate, code/dependency scope, feedback horizon, cost estimate
Policy audit	hidden-test protection, immutable grader, no future history, network rules
Acceptance	reason for pass/reject/repair and hashes for every frozen artifact

C.3 Trajectory checklist

Table 12: Minimum lineage for a released or consumed coding-agent trajectory.

Category	Report
Generator	policy checkpoint, teacher/student role, harness, decoding and search budget
Replay	initial state digest, actions, observations, state deltas/checkpoints, terminal artifacts
Labels	verifier version, outcome vector, rubric/process labels, audit flags, failure attribution
Transformations	selection, branch pruning, repair, relabeling, compaction, token loss masks
Routing	compact SFT, recovery SFT, preference, PRM, online-RL seed, quarantine, or discard
Diversity	repositories, languages, task types, harnesses, failure modes, difficulty bins
Cost	generation tokens, tool time, environment time, judge/teacher calls, accepted yield

C.4 Model data card checklist

A coding-agent checkpoint should identify all training task pools and their cutoffs; any benchmark-derived artifacts; repository and organization overlaps with reported evaluations; environment and verifier versions; teacher models and external APIs; harness distribution; proportions of compact, recovery, preference, and online experience; objective sequence; relabeling policy; licenses and deletion procedure; and known contamination. If a field cannot be disclosed, the limitation should be explicit rather than silently treated as zero overlap.

References

Amazon Science. 2025. SWE-PolyBench: A multi-language, multi-task software engineering benchmark. Project repository.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. 2025. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *arXiv preprint arXiv:2505.20411*.

Ibragim Badertdinov and 1 others. 2026. SWE-rebench V2: Scalable multilingual software-engineering environments. *arXiv preprint arXiv:2602.23866*.

Shiyi Cao, Dacheng Li, Fangzhou Zhao, Shuo Yuan, Sumanth R. Hegde, Connor Chen, and 1 others. 2025. Skyrl-agent: Efficient RL training for multi-turn LLM agents. *arXiv preprint arXiv:2511.16108*.

Jun Shern Chan and 1 others. 2024. MLE-bench: Evaluating machine learning agents on machine learning engineering. arXiv preprint arXiv:2410.07095.

Zihao Cheng, Hongru Wang, Zeming Liu, Xinyi Wang, Xiangrong Zhu, Yuhang Guo, Wei Lin, Jeff Z. Pan, and Yunhong Wang. 2026. Terminal-world: Scaling terminal-agent environments via agent skills. *arXiv preprint arXiv:2605.20876*.

DataCurve. 2026. DeepSWE: Original long-horizon software engineering tasks. Benchmark repository.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, and 1 others. 2025. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*.

Kawin Ethayarajh, Winnie Xu, Niklas Muennighoff, Dan Jurafsky, and Douwe Kiela. 2024. KTO: Model alignment as prospect theoretic optimization. *arXiv preprint arXiv:2402.01306*.

Lianghong Guo, Yanlin Wang, Caihua Li, Wei Tao, Pengyu Yang, Jiachi Chen, Haoyu Song, Duyu Tang, and Zibin Zheng. 2025. SWE-Factory: Your automated factory for issue resolution training data and evaluation benchmarks. *arXiv preprint arXiv:2506.10954*.

Hao Han, Jin Xie, Xuehao Ma, Weiquan Zhu, Ziyao Zhang, Zhiliang Long, Hongkai Chen, and Qingwen Ye. 2026. SWE-TRACE: Optimizing long-horizon SWE agents through rubric process reward models and heuristic test-time scaling. *arXiv preprint arXiv:2604.14820*.

Zhenyu Hou, Yujiang Li, Jie Tang, and Yuxiao Dong. 2026. Single-rollout asynchronous optimization for agentic reinforcement learning. *arXiv preprint arXiv:2607.07508*.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. 2025. R2E-Gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents. *arXiv preprint arXiv:2504.07164*.

Mohan Jiang, Dayuan Fu, Junhao Shi, Ji Zeng, Weiye Si, Keyu Li, Xuefeng Li, Yang Xiao, Wenjie Li, Dequan Wang, and Pengfei Liu. 2026. davinci-agency: Unlocking long-horizon agency data-efficiently. *arXiv preprint arXiv:2602.02619*.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*.

Tue Le, Minh V. T. Thai, Dung Nguyen Manh, Huy Phan Nhat, and Nghi D. Q. Bui. 2025. SWE-EVO: Benchmarking coding agents in long-horizon software evolution scenarios. *arXiv preprint arXiv:2512.18470*.

Yujiang Li, Zhenyu Hou, Yi Jing, Jie Tang, and Yuxiao Dong. 2026a. Compactionrl: Reinforcement learning with context compaction for long-horizon agents. *arXiv preprint arXiv:2607.05378*.

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

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, and 1 others. 2026. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*.

Meta AI and collaborators. 2025. MLGym: A new framework and benchmark for advancing AI research agents. Project repository.

Microsoft Research and collaborators. 2025. SWE-bench-Live: A continually updated benchmark for software engineering agents. Project repository.

OpenAI. 2025. SWE-Lancer: Can frontier LLMs earn one million dollars from real-world freelance software engineering? Technical report, arXiv:2502.12115.

OpenAI. 2026a. Separating signal from noise in coding evaluations. Technical audit.

OpenAI. 2026b. Why we no longer evaluate SWE-bench Verified. Technical audit.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2024. Training software engineering agents and verifiers with SWE-Gym. *arXiv preprint arXiv:2412.21139*.

Zhihong Pan, Jiyuan He, Kai Zhang, Yupeng Han, Ze Liu, Yuze Zhao, Yongcong Ye, and Zhaohua Yang. 2026. Meta-task: Turning terminal task synthesis into a terminal task for scalable agent training. *arXiv preprint arXiv:2607.27929*.

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

PostTrain Arena Team. 2026. Posttrain arena. Project website.

PostTrainBench Team. 2026. Posttrainbench: Can LLM agents automate LLM post-training? *arXiv preprint arXiv:2603.08640*.

Yunjia Qi, Zehua Yin, Xintong Shi, Hao Peng, Songyuanyi Lu, Yixian Liu, Richeng Xuan, Yuhong Liu, Zhichao Hu, Xiaozhi Wang, Lei Hou, Bin Xu, and Juanzi Li. 2026. TRAJDEBUG: Tracing error lifecycle to identify critical failures in long-horizon agent trajectories. *arXiv preprint arXiv:2608.06346*.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. In *Advances in Neural Information Processing Systems*.

RUC-NLPIR and collaborators. 2026. Towards long-horizon agents: A survey. Project website and survey manuscript.

Scale AI. 2025. SWE Atlas: A benchmark of professional software engineering workflows. Technical report.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Wentao Shi, Mengqi Yuan, Junkang Wu, Qifan Wang, and Fuli Feng. 2024. Direct multi-turn preference optimization for language agents. *arXiv preprint arXiv:2406.14868*.

Yifan Song, Da Yin, Xiang Yue, Jie Huang, Sujian Li, and Bill Yuchen Lin. 2024. Trial and error: Exploration-based trajectory optimization for LLM agents. *arXiv preprint arXiv:2403.02502*.

Giulio Starace and 1 others. 2025. Paperbench: Evaluating AI's ability to replicate AI research. Technical report, arXiv:2504.01848.

SWE-Pruner Team. 2026. SWE-Pruner: Repository context pruning for software engineering agents. *arXiv preprint arXiv:2601.16746*.

Jingjing Wang, Xiwen Chen, Wenhui Zhu, Huayu Li, Zhengxiao He, Feiyang Cai, Ana S. Carreon-Rascon, Xuanzhao Dong, and Feng Luo. 2026. Context pruning for coding agents via multi-rubric latent reasoning. *arXiv preprint arXiv:2605.15315*.

Qi Wang, Hongzhi Zhang, Jia Fu, Kai Fu, Yahui Liu, Tinghai Zhang, and 1 others. 2025. Klear-agentforge: Forging agentic intelligence through posttraining scaling. *arXiv preprint arXiv:2511.05951*.

Xingyao Wang and 1 others. 2024. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*.

Xin-Cheng Wen, Binbin Chen, Haoxuan Lan, Hang Yu, Peng Di, and Cuiyun Gao. 2026. SWE-Fuse: Empowering software agents via issue-free trajectory learning and entropy-aware RLVR training. *arXiv preprint arXiv:2603.07927*.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, and 1 others. 2024. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems*.

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

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2023. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *arXiv preprint arXiv:2306.14898*.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, and 1 others. 2025. Multi-SWE-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605*.

Hanrong Zhang, Yankai Chen, Shicheng Fan, and 1 others. 2026. Scaling LLM agent learning with data synthesis: A comprehensive survey. OpenReview preprint.

Zhaoxi Zhang, Yitong Duan, Yanzhi Zhang, Yiming Xu, Zhixiang Wang, Kun Liang, and 1 others. 2025. One tool is enough: Reinforcement learning for repository-level LLM agents. *arXiv preprint arXiv:2512.20957*.