# SHEPHERD: A Runtime Substrate Empowering Meta-Agents with a Formalized Execution Trace

**Simon Yu**[*†] **Derek Chong**[*‡] **Ananjan Nandi**[*‡]
**Dilara Soylu**[‡] **Jiuding Sun**[‡] **Christopher D Manning**[‡] **Weiyan Shi**[†]
[†]Northeastern University [‡]Stanford University
{yu.chi, we.shi}@northeastern.edu
{derekch, ananjan, soylu, sunjd24, manning}@stanford.edu
[*]Equal contribution

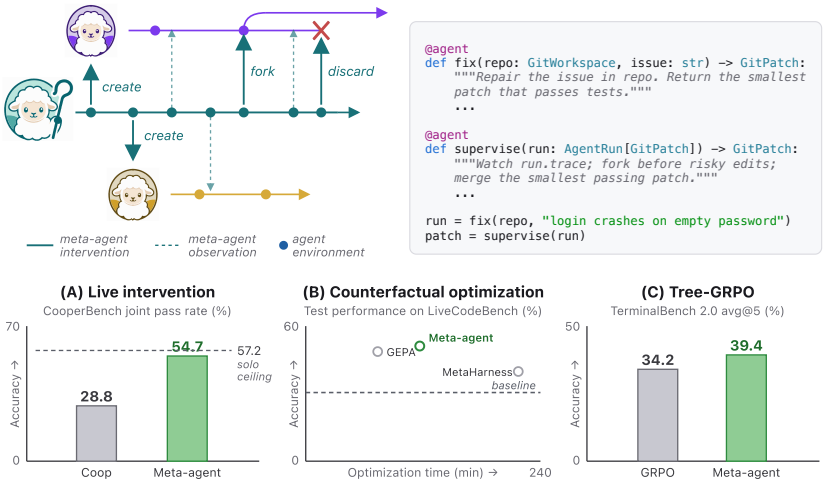


Figure 1: **SHEPHERD meta-agents.** *Top:* A supervisor meta-agent manages code repair agents. *Bottom:* Results from three meta-agents: (A) live supervision; (B) meta-optimization; **(C)** Tree-GRPO

## Abstract

As LLM agent systems take on more complex tasks, they increasingly rely on meta-agents: higher-order agents that operate on other agents, much as managers supervise employees. Whatever a meta-agent does: coordinating agents, halting risky actions before execution, or repairing failed runs, requires manipulation of agentic execution at runtime. Existing agentic substrates make this hard: they give meta-agents only plain transcripts and environment snapshots, requiring it to build it's own tooling to reconstruct and orchestrate execution state. Therefore, we introduce SHEPHERD, a Python substrate grounded in functional programming principles, where an agent's execution is itself a *first-class object* that a meta-agent can inspect and transform. Every model call, tool call, and environment change becomes a structured event in a Git-like execution trace, where any past state can be forked 5× faster than `docker commit` and replayed. Three example use cases show SHEPHERD's versatility: (1) a supervisor agent prevents conflicts among parallel coding agents, lifting CooperBench performance from 28.8% to 54.7%; (2) a counterfactual optimizer repairs agent workflows by proposing edits and replaying runs from the point of changed behavior, outperforming MetaHarness on TerminalBench-2 with 58% lower wall-clock; (3) a meta-agent picks fork points during rollouts to improve credit assignment in long-horizon agentic RL, doubling

AI Agents for Discovery in the Wild (AID-Wild), Workshop at ACM CAIS 2026.

GRPO's gains on TerminalBench-2. We open-source SHEPHERD to empower future meta-agents with principled and efficient operations over agentic execution.

## 1 Introduction

As LLM-based agentic systems mature, we increasingly see the use of agents that act on other agents at runtime. Asawa et al. [3], Lin et al. [23] develop advisor agents that learn intervention policies from execution traces to steer agents away from dead ends; meta-optimizers such as GEPA and MetaHarness optimize agentic workflows to improve performance on a given downstream task [2, 19]; and Hou et al. [10], Ji et al. [12] build tree-search RL that branches rollouts to compare rewards from alternate continuations and improve per-step credit assignment. We call these systems *meta-agents*: higher-order agents that operate over other agents and their execution traces. Meta-agents are becoming increasingly central to extracting capability from agentic systems [45].

Table 1: **Substrate support for runtime meta-agent operations.** ● = fully supported; ◐ = supported only for the agent; ◑ = supported only for the environment; ○ = not supported.

| Meta-agent Operation<br>operating layer | SHEPHERD<br>agent + env | OpenHands<br>agent | AgentGit<br>agent | BranchFS<br>env | Docker<br>env |
|---|---|---|---|---|---|
| **Live action monitoring** | ● | ◐ | ○ | ○ | ○ |
| **Fork execution** | ● | ◐ | ◐ | ◑ | ◑ |
| **Rewind and replay past state** | ● | ◐ | ◐ | ◑ | ◑ |
| **Agent behavior modification** | ● | ◐ | ◐ | ○ | ○ |

Yet, existing agentic substrates are not designed for meta-agents. Consider a supervisor meta-agent that forks a coding agent before a risky write, watches the branch execute, and discards it on failure. To do this, it needs to *monitor* the agent's execution live, *fork* before the write, *rewind* on failure, and *modify* the agent itself to patch the issue, after which execution needs to be *resumed*. Recent work exposes fragments of these operations: OpenHands surfaces a session's event stream [37], AgentGit gives the worker Git-like commit tools [22], BranchFS isolates the filesystem [34] (Table 1). However, these substrates are designed to maintain runtime state for the running agent, not to give a meta-agent the operations it needs to act on that agent. As a result, meta-agent implementations built on them need to reinvent custom tooling to support these operations.

We argue that the solution is to give an agent and its execution the first-class treatment that functions receive in functional programming: *structured data* that a meta-agent can hold, execute, copy, and rewrite. Consequently, this enables *algebraic effect handlers* that intercept and observe execution without modifying it, and *continuations* that enable pause-inspect-decide-resume patterns for meta-agents. We therefore propose SHEPHERD, a principled functional programming model for higher-order agents, instantiated as an intuitive Python substrate (Figure 1, Section 3).

SHEPHERD defines agents as *tasks* with typed inputs and outputs, and records the execution of these tasks in a Git-like *execution trace*: every agent action (filesystem modification, model call, database operation) becomes a commit, every fork is a branch, every past agent-environment state is reachable and replayable through checkouts. Over this trace, a meta-agent *reads* a task's execution by subscribing to its commits; *rewinds* it by checking out an earlier commit, restoring the exact prior agent-environment state; *branches* it by forking a scope, which copies its state; and *modifies* it by rewriting the task definition, after which execution can be resumed. Our principled functional programming roots give meta-agents in SHEPHERD several crucial capabilities: an observing meta-agent does not perturb an agent's trajectory, parallel agents can run as separate processes over forked environments within a shared sandbox without perturbing each other, and rewinding to any intermediate states restores a byte-identical copy of its agent-environment state. We formalize these properties through a small algebraic-effects calculus mechanized in Lean, which provides a precise semantic contract for the execution trace. The implementation is lightweight: for a 5.8 GB docker image, SHEPHERD forks the agent-environment state at $5\times$ the speed of a `docker commit`, and reuses over $95\%$ of the LLM provider's KV cache.

Our design enables the easy implementation of meta-agent applications that previously required substantial bespoke engineering. We showcase three example meta-agents spanning the agent's lifecycle. *During execution*, a *live supervisor meta-agent* (§5.1) watches the execution traces of parallel coding workers and intervenes before they conflict, raising CooperBench [16] joint pass rate from 28.8% to 54.7%. *After execution*, a *counterfactual replay meta-optimizer* (§5.2) diagnoses failures in prior agent runs and validates candidate patches by branching runs at the first point where the patch would change behavior, beating state-of-the-art meta-optimizers such as MetaHarness [19]

by up to 11 points on standard benchmarks such as LiveCodeBench [11] and TerminalBench-2 [24], while cutting wall-clock by up to 58%. *During training*, a *tree-search RL trainer* (§5.3) forks rollouts at meta-agent-chosen turns and samples sibling continuations to improve credit assignment by computing advantages from outcome rewards, outperforming GRPO [29]'s avg@5 performance when used to train Qwen3.5-35B-A3B [28] on TerminalBench-2 by 5.2 points.

In summary, we contribute (i) SHEPHERD, a programming model for higher-order agents whose core operations are grounded in functional programming and mechanized in Lean; (ii) a Python framework instantiating this model with a performant and efficient substrate; and (iii) three meta-agents spanning the agent lifecycle built using the SHEPHERD substrate.

## 2 Related Work

**Meta-Agents.** Meta-agent applications are emerging in recent work [45, 44]. Darwin-Gödel Machines [47] and Group-Evolving Agents [40] maintain dynamic archives of self-modifying code. Hyperagents [48] and Meta-Harness [19] optimize a task agent's problem-solving strategies at the meta level. Other lines refine an agent's context and long-term retrieval through evolutionary search [25, 38] or memory-augmented architectures [50, 21, 51, 42, 39, 30]. Each of these methods reinvents the runtime machinery needed to act on agents – parsing transcripts, building bespoke environment snapshots, re-executing with modified source code. SHEPHERD provides a unified substrate for these operations, letting a meta-agent observe agents without perturbing them, fork their coupled agent-environment state, and replay prior execution byte-identically.

**Agentic Meta-Optimization.** Orchestrating multiple LLM agents is a common strategy for performing complex tasks and inference-time scaling. Standard multi-agent frameworks [41, 9, 1] route natural-language messages between workers; CooperBench [16] shows the coordination failures that can result from this. Pipeline optimizers [15, 6, 52] and test-time scaling methods [17, 20] use parallel rollouts and majority voting, evaluating each candidate by full end-to-end re-execution. GEPA [2] introduces a reflective meta-agent that proposes workflow edits. These methods treat the underlying execution as a black box and re-run candidates from scratch. SHEPHERD adds a different evaluation primitive: a meta-agent can branch a worker's context at the exact commit where an edit first alters behavior and replay only the affected suffix (Section 4), so candidates reuse computation effectively.

**Agentic Runtime and Infrastructure.** A parallel line of research adds infrastructure support for agentic state management, placing the checkpoint primitive at different layers of the software stack. AgentGit [22] exposes version-control operations as cooperative tools the agent can invoke from within a LangGraph workflow. BranchFS [34] adds a kernel-level `branch()` system call that isolates filesystem state, independent of the worker's tool-call structure. AgentSPEX [35] embeds checkpointing into a domain-specific language for agent workflows. Each of these places the checkpoint primitive at a different point in the stack, with different trade-offs between agent autonomy, transparency, and language-level integration. SHEPHERD sits at a different point in this design space: it couples environment states with the agent execution state, so the substrate observes the same events the worker emits without requiring the worker to be rewritten or replayed from scratch. This lets the same substrate support live supervision, post-hoc trajectory optimization, meta-optimization and stateful RL under a unified interface.

## 3 The SHEPHERD Programming Model

The supervisor meta-agent in Figure 1 needs the following operations on worker agents: *monitor* what it is doing as the worker runs, *fork* before a risky write to try an alternative continuation, *rewind* on failure to return to an earlier moment, and *modify* the worker itself to patch the failure, after which execution is *resumed*. These operations are hard to support for agentic

Table 2: Mapping of SHEPHERD primitives to meta-agent operations and FP constructs.

| Primitive | Operation | FP construct |
|---|---|---|
| Task | Agent Behavior Modification | typed function |
| Effect | Action Monitoring and Gating | algebraic effect |
| Scope | Forking Execution | scoped effect handler |
| Trace | Rewinding and Replaying Execution | persistent data structure |

execution because it is full of side effects, such as model calls, filesystem writes, and tool invocations, that resist being treated as inspectable, manipulable values.

Functional programming (FP) has a long tradition of structuring effectful computation by drawing a boundary separating *what* a computation describes from *how* its effects reach the world. Computation inside this boundary becomes substitutable, observable without perturbation, branchable, and replayable. SHEPHERD extends this discipline to agentic execution, treating it as a first-class object. Doing so requires elevating four things to first-class status: what the agent *is* (**tasks**, § 3.1), what it *does* (**effects**, § 3.2), where it *runs* (**scopes**, § 3.3), and what has been *done* (**execution trace**, § 3.4). Each primitive is grounded in a functional-programming construct (Table 2), and the deterministic core of SHEPHERD rests on a small Lean-mechanized semantics for typed effect traces (Appendix B).

### 3.1 Task: Agent Behavior Declaration

For a supervisor meta-agent to modify an agent – by changing it's code, composing them into workflows or synthesizing a fresh one – the agent must be a *value*: something that can be held, passed as an argument, or returned from a call. Functional programming gives this property to functions. A function with a typed input and a typed output is substitutable for any other function with the same type, and a higher-order function can take functions as arguments. Agents in SHEPHERD have the same shape. A **task** is a typed function over agentic execution with typed input and output, and a body that may call LLMs, tools, and other tasks. They are declared using the `@agent` decorator over typed Python functions:

```python
@agent
def fix(repo: GitWorkspace, issue: str) -> GitPatch:
"""Repair the issue in the repo."""

@agent
def supervise(work: Task[GitPatch]) -> GitPatch:
"""Watch work.trace; fork before risky edits; merge the passing patch."""
```

**Tasks are substitutable values.** A task is fully specified by its signature and docstring – SHEPHERD compiles them into an LLM prompt guided by the docstring, where the output is validated against the defined type. As shown above, meta-agents in SHEPHERD are just tasks whose arguments happen to be other tasks, and they hierarchically allow for meta-meta-agents over their execution. Any task with the same typed signature can also stand in for any other, allowing meta-agents to easily edit behavior at runtime. They can also easily synthesize fresh tasks at runtime just by specifying their input, output and expected behavior.

A task is a value, but its body is not yet substrate-visible: when an agent interacts with its environment, the substrate has no handle on it. The next primitive gives every such action a typed value of its own.

### 3.2 Effects: Agent Actions

A supervisor meta-agent needs to see what the worker agent is doing while it executes, without perturbing it. Functional programming solves the analogous problem for effectful programs with *algebraic effects*: any operation that touches the outside world (read a file, send a message) is reified as a typed event, and a *handler* decides what that event means. The same program can be run under different handlers without changing its source: one handler could execute it, while another logs it.

SHEPHERD applies this discipline to agent actions. An **effect** is a typed record of a single action attempted by a worker. They can record LLM calls, tool calls, environment mutations, or even user-defined custom actions. Every effect a worker emits is appended to the immutable *effect stream* of the scope it's running in (§3.3), and a meta-agent can read the stream by subscribing to it.

**An action's intent and outcome are separate events.** Each action emits two effects: an *intent* when the worker issues the action, and an *outcome* when the world responds. Therefore, a meta-agent can read an intent, decide it shouldn't materialize, and act before the outcome arrives, as below:

```python
async for effect in work.trace.live():
    if isinstance(effect, ToolCallIntent) and is_destructive(effect):
        if check(effect) == "deny":
```

```
            work.discard()  # outcome never materializes
```

**Effects are reversible until materialized.** Every effect carries a reversibility tier that determines its behavior when *materialized*, or executed against the world. *Reversible* (such as filesystem writes, sandbox state) and *compensable* effects (such as database writes) are captured by the substrate at emission time and can be rolled back at will by a meta-agent. Reversible effects roll back natively through their scope (§ 3.3), and compensable effects roll back through user-supplied compensation handlers invoked by the substrate. However, *irreversible* effects (such as model calls, emails) materialize on emission, and the stream can only record them for audit.

**Observation is non-perturbing.** The effect stream is append-only and immutable. Therefore, a worker's effect stream is byte-identical whether or not a meta-agent is watching. A meta-agent can read a worker's actions and gate which of them reach the world, without affecting the worker itself. Because intent is decoupled from execution, a meta-agent can also replay a slice of the effect stream under a different handler. For example, a meta-agent can replay an intent under a modified task body to see how the task would have behaved differently.

Reading and reinterpreting actions is enough for an observer. A meta-agent that wants to try an alternative continuation needs to be able to branch the worker and its environment atomically.

### 3.3 Scopes: Agent Environments

When the supervisor meta-agent branches an agent, the alternative must run in its own world. Any effects emitted in the branch must not leak into the parent's effect stream. The functional programming construct enabling this is the *region-scoped effect handler*. A function can open a fresh handler for some sub-region of its execution, run inside it, and then either propagate that region's effects outward (commit) or abandon them (revert). Regions nest cleanly: each level owns its own effect interpretation and cannot contaminate others. A **scope** in SHEPHERD implements this construct for agentic execution. It binds the worker's sandbox handles, model providers, tool surfaces, and effect-stream cursor, and it owns the effects emitted by tasks running inside it.

**Scopes support four primitives.** `emit` writes an effect to the scope's stream. `fork` opens a copy-on-write child scope. `merge` propagates a child's effects into its parent. `discard` abandons a child, leaving the parent untouched. Filling in `supervise`'s body from §3.1:

```
@agent
def supervise(work: Task[Issue, Patch]) -> Patch:
    child = scope.fork()
    result = await work(child)
    if result.failed: child.discard()
    else: scope.merge(child)
    return result
```

**The agent and its environment are forked atomically.** `scope.fork()` captures the worker's filesystem, processes, and bindings in one atomic copy-on-write step. A subsequent `discard` therefore rolls back not just what the worker said or returned, but every trace of what it touched. The substrate realizes this through overlay-filesystem virtualization and the native checkpoint facilities of containerized sandboxes, behind a unified device-layer interface (Appendix C.7).

**Scopes can be parallelized, nested, reverted and resumed.** A meta-agent can create concurrent forks from a parent scope and merge or discard each independently. Scopes also nest: a meta-meta-agent can fork, observe, and resume a meta-agent without contaminating the worker beneath. Discarding a child scope leaves the parent byte-identical to the moment of fork. Resuming a scope also preserves the worker agent's world: a paused worker resumed by a meta-agent sees the bindings recorded in its original scope, not whatever the meta-agent currently holds.

A scope owns the *present* region of execution. However, to *rewind* to an arbitrary past state, execution history itself must be a navigable value. We address this next.

### 3.4 Execution Trace: Agent Execution History

A supervisor meta-agent that wants to *rewind* a worker agent — return to an earlier moment in its execution and either inspect it or run forward from there under different conditions — needs every past state of the worker's execution to be reachable on demand. In functional programming, *persistent data structures* provide this property: every version of the structure remains accessible after modification, with new versions sharing structure with old ones rather than copying. In SHEPHERD, the counterpart to this is the **execution trace**. It is a persistent Git-like commit graph: each scope's effect stream materializes as a sequence of typed commits on a branch of the graph. The four scope operations of § 3.3 then compile to Git-like operations as well:

```
scope.emit(effect)    ~  shepherd commit -m "<effect>"
scope.fork()          ~  shepherd checkout -b <child-branch>
scope.merge(child)    ~  shepherd merge <child-branch>
scope.discard(child)  ~  shepherd branch -D <child-branch>
```

**Every past state is reachable and replayable.** A meta-agent can navigate to any commit by hash and read the exact agent-environment state at that moment, with its full scope intact. Locating the specific commit where a regression first appeared, or where two siblings began to diverge, reduces to graph traversal on this trace. Replaying from a past commit also produces a byte-exact reconstruction of the scope before diverging, so the only cost paid by the meta-agent is the executed suffix.

**Divergent branches share storage.** Just as persistent data structures share substructures across versions, the execution trace shares storage across branches: two siblings forked from the same commit share their entire prefix by content hash. A meta-agent fanning out across many forks pays only for the divergent suffixes, and any set of branches can be diffed based on their captured effects — letting a meta-agent decide which to merge on the basis of their behavior.

## 4 Framework Performance

Three cost properties are crucial for meta-agents in SHEPHERD to be performant: `scope.fork()` must be image-size-independent so branching is affordable; the trace must scale with what the agent writes and not the image so it can persist across long executions; and the byte-identical replay guarantee must reach the LLM provider's prompt cache so re-execution efficiently reuses KV cache. We measure each on real TerminalBench-2 images. Full results are in Appendix C.

Table 3: Fork/revert latency (ms), storage, and host resource cost at $K=4$ concurrent branches across three Terminal-Bench 2.0 images. SHEPHERD's 134–143 ms fork is ∼2–3% of one agent turn (Appendix C.3). Best per group in **bold**; full ±std and protocol in Appendix C.1.

| Image | Method | Fork ↓ | Revert ↓ | Storage ↓ | Branching (K=4) Disk ↓ | Branching (K=4) RAM ↓ |
|---|---|---|---|---|---|---|
| openssl-selfsigned-cert (42 MB image) | Full copy | 5,154 ms | 2,067 ms | 268 MB | 804 MB | 112 MB |
| | Docker commit | 658 ms | 749 ms | 30 KB | 90 KB | 29.8 MB |
| | Modal snapshot | 3,764 ms | 2,260 ms | — | — | — |
| | BranchFS [34] | 266 ms | 360 ms | 12 KB | 48 KB | 22.7 MB |
| | SHEPHERD | **134 ms** | **142 ms** | **10 KB** | **30 KB** | **20.5 MB** |
| caffe-cifar-10 (200 MB image) | Full copy | 5,971 ms | 3,446 ms | 645 MB | 1.9 GB | 230 MB |
| | Docker commit | 692 ms | 761 ms | 30 KB | 90 KB | 38.3 MB |
| | Modal snapshot | 3,291 ms | 2,463 ms | — | — | — |
| | BranchFS [34] | 272 ms | 357 ms | 12 KB | 48 KB | 29.4 MB |
| | SHEPHERD | **135 ms** | **140 ms** | **10 KB** | **30 KB** | **27.1 MB** |
| pytorch-model-recovery (5.8 GB image) | Full copy | 53,462 ms | 25,943 ms | 8.3 GB | 24.9 GB | 910 MB |
| | Docker commit | 725 ms | 828 ms | 30 KB | 90 KB | 30.2 MB |
| | Modal snapshot | 3,160 ms | 2,328 ms | — | — | — |
| | BranchFS [34] | 280 ms | 358 ms | 12 KB | 48 KB | **22.7 MB** |
| | SHEPHERD | **143 ms** | **147 ms** | **10 KB** | **30 KB** | 25.7 MB |

**Fork and revert are fast and image-size-independent.** SHEPHERD fork creates a new copy-on-write layer on top of the existing filesystem instead of duplicating it, so cost is constant regardless of

image size. As a result, forks take 134–143 ms regardless of image size (42 MB to 5.8 GB); on the 5.8 GB image, K forks cost K × 143 ms against K × 53.5 s for full-rootfs copies, a 192× per-branch slowdown (Table 3). The next-fastest alternative, BranchFS [34], branches the filesystem alone via FUSE; like the other methods, it supports no notion of the agent itself (Table 1).

**Replay reuses the LLM provider's prompt cache.** Because SHEPHERD fork preserves the parent's byte-identical LLM message prefix, the provider's prompt cache resolves it without invalidation. On Anthropic Claude Haiku 4.5 across 8 Terminal-Bench 2.0 tasks, cache-hit rate plateaus at ∼95% from $K{=}2$ onwards, within 5% of the byte-identical ceiling. Cache reuse compounds whenever a meta-agent fans out (Tree-GRPO siblings, Section 5.3) or replays completed trajectories (trajectory compression, Section D). We provide further detail in Appendix C.6.

# 5 Experiments

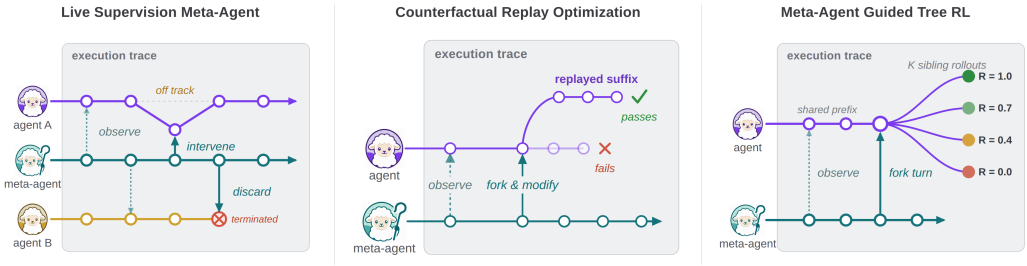


Figure 2: Three meta-agent applications implemented in SHEPHERD

The SHEPHERD substrate enables a wide range of meta-agent applications. In this section, we demonstrate three such applications, *spanning the agent development stack*. (1) *During execution*, a **live supervisor agent** monitors two parallel coding workers and intervenes mid-trajectory, raising the pair pass rate on CooperBench from 28.8% to 54.7% (§5.1). (2) For *post-hoc workflow optimization*, a **meta-optimizer** branches execution traces to test counterfactual workflow edits, outperforming MetaHarness on four of five datasets at up to ∼60% lower wall-clock (§5.2). (3) For *agentic RL training*, a meta-agent selects fork points during RL rollouts to extract per-step credit, lifting Terminal-Bench 2.0 by 5.2% on Qwen3.5-35B-A3B and 3.4% on Nemotron-3-Super-120B-A12B over Flat GRPO (§5.3).

Each application exercises a different SHEPHERD property. § 5.1 requires *non-perturbing observation*: a supervisor that subscribes to both workers' effect streams gains access to their traces without perturbing them. § 5.2 leans on *byte-identical replay*: candidate edits are validated against a fixed baseline by re-executing only the affected suffix. § 5.3 exercises *cheap branching*: SHEPHERD's cheap agent–environment state forking makes per-step credit assignment via sibling rollouts affordable. We report a fourth use case, which is to compress completed trajectories into shorter reruns under meta-agent hindsight in Appendix D. We note that these applications are not exhaustive and discuss further possible applications enabled by SHEPHERD in Appendix A.2.

## 5.1 Meta-Agent for Live Supervision: Sub-Agent Coordination

**Motivation.** CooperBench [16] documents a *curse of coordination*: even when allowed to communicate, parallel coding agents coordinate poorly enough that they succeed less often than a single agent working alone. SHEPHERD's effect stream and scope primitives let a meta-agent close that gap by inspecting both workers' execution in real time and intervening before damage compounds.

**Method.** Two Claude Haiku 4.5 worker agents run in parallel forked scopes, each assigned one complementary feature to implement. A Claude Sonnet 4.6 or Opus 4.7 meta-agent subscribes to both effect streams via SHEPHERD and is provided with three coordination tools: `inject` (push guidance into a worker's session), `handoff` (fork the leading worker's scope as the follower's new root and restart), and `discard` (abort a stuck worker via `scope.discard()`).

**Setup.** We evaluate on CooperBench [16]. Baselines are *solo* (one Haiku 4.5 agent handling both features sequentially) and *coop* (two parallel Haiku 4.5 agents in forked scopes with peer-to-peer messaging via the relay sandbox, no supervisor). Full protocol, dataset construction, and per-condition operational notes are in Appendix E.

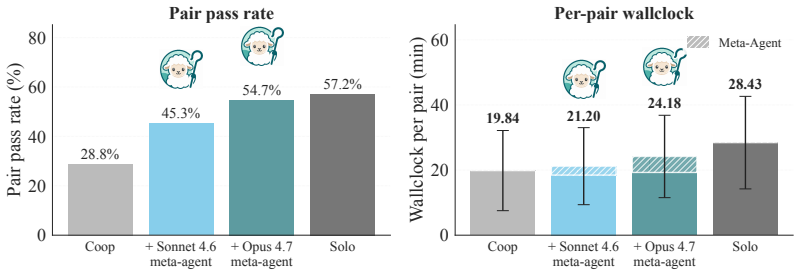


Figure 3: Live intervention experiments on CooperBench, with Claude Haiku 4.5 as worker. **Left (Pair pass rate):** Sonnet 4.6 and Opus 4.7 meta-agents close most of the gap from coop (28.8%) to the solo ceiling (57.2%). **Right (Per-pair wall-clock):** mean wall-clock minutes per pair. Solid bars are the worker wall-clock; the hatched overlay is the additional meta-agent overhead.

**Results.** On the full 479-pair set, the coop baseline lands at 28.8% pair pass rate, reproducing CooperBench's documented coordination penalty against the solo ceiling of 57.2%, showing a 28.4-point gap. A Sonnet meta-agent recovers 45.3%, and an Opus meta-agent reaches 54.7%, closing 91% of the curse-of-coordination gap (Figure 3, left). In the naive coop baseline, each worker sees only what the other chose to send, which is filtered through the sender's imperfect view of the shared state. The supervisor, by contrast, subscribes directly to both workers' effect streams, observing their actions completely without perturbing them and steering them accordingly. Appendix E shows that the supervisor prefers the lightweight `inject` intervention while still taking `handoff` and `discard` interventions when necessary.

The wall-clock cost is modest: solo takes 28.4 min per pair on average, while the supervised conditions stay close to the parallel coop baseline (19.8 min). Sonnet finishes in 21.2 min (1.4 min meta overhead) and Opus in 24.2 min (4.3 min meta overhead), running at roughly 75% and 85% of solo time respectively (Figure 3, right). This is because the meta-agent only observes worker agents once every few seconds, and does not interfere with them unless an intervention is required (Appendix E). We treat this as a proof of existence that SHEPHERD enables effective live supervision; the supervision token-cost vs. task-execution-cost trade-off is discussed in Appendix A.

### 5.2 Meta-Agent for Meta-Optimization: Counterfactual Replay Optimization (CRO)

**Motivation.** When an agentic workflow fails, the failure usually traces to a small set of faulty or missing agent calls [46, 4]. Counterfactual analysis [33] provides standard machinery for diagnosing such failures: would a localized change to a suspect set of calls have fixed the outcome? Answering this in a standard agentic runtime is difficult – workflow re-runs reintroduce stochastic and environmental variation unrelated to the edit, and flat transcripts give little structure to localize failures against. CRO sidesteps both problems by replaying counterfactuals through forks of SHEPHERD's execution trace rather than re-executing changed workflows from scratch.

**Method.** CRO maintains a pool of workflow variants together with their SHEPHERD execution trace on the training set. At each step, the proposer analyzes these traces to identify failure modes, picks a parent candidate and emits a set of candidate edits as counterfactual experiments to patch these failure modes. Every edit is paired with a *fix set* of training examples it should repair and a *guard set* whose performance must not regress. SHEPHERD validates these edits through *counterfactual replay* on the combined fix and guard set: for each edit, it forks the parent's execution trace at the first commit that would be affected by the edit and replays the suffix with the edited workflow. Candidates that outperform their parent on this combined set are evaluated on the dev set and added to the candidate pool; after a set number of iterations CRO returns the highest-scoring member on the dev set as the selected candidate (Algorithm 1, Appendix F).

**Setup.** We evaluate on subsets of HoVer [13], MATH [8], IFBench [27], LiveCodeBench [11], and TerminalBench 2 ( [24]), comparing CRO against the baseline workflow, GEPA (optimizing workflow

Table 4: Test-set performance and meta-optimization wall-clock time across datasets and methods. HoVer/MATH/IFBench/LiveCodeBench report mean ± std, and TerminalBench-2 report average@5 on the test split (Test) over three evaluations and meta-optimization wall-clock in minutes (Wall). Best test mean per dataset in **bold**; second-best underlined.

| | HoVer | | MATH | | IFBench | | LiveCodeBench | | TerminalBench-2 | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Method** | Test | Wall | Test | Wall | Test | Wall | Test | Wall | Test | Wall |
| Baseline | $43.7 \pm 0.0$ | — | $60.7 \pm 1.2$ | — | $42.4 \pm 1.8$ | — | $30.7 \pm 2.1$ | — | 31.2 | — |
| GEPA | $43.7 \pm 0.0$ | 67 | $74.0 \pm 3.5$ | 20 | $50.1 \pm 1.2$ | 50 | $48.7 \pm 1.5$ | 73 | 31.2 | 157 |
| MetaHarness | $77.8 \pm 0.4$ | 235 | $79.3 \pm 1.2$ | 101 | $\mathbf{52.3 \pm 1.4}$ | 126 | $40.0 \pm 3.6$ | 217 | 31.2 | 173 |
| **CRO** | $\mathbf{79.4 \pm 0.2}$ | 120 | $\mathbf{80.0 \pm 2.0}$ | 42 | $51.3 \pm 1.1$ | 82 | $\mathbf{51.0 \pm 1.7}$ | 117 | **35.2** | 73 |

code) [2], and MetaHarness [19]. The executor is `GPT-5.4-mini` and meta-optimizers use `GPT-5.4` (in the Codex harness for MetaHarness and GEPA). We stop each algorithm after 20 candidates on HoVer, MATH, IFBench, and LiveCodeBench, and after 10 on TerminalBench-2. Per-dataset settings, baselines, and full results are in Appendix F.

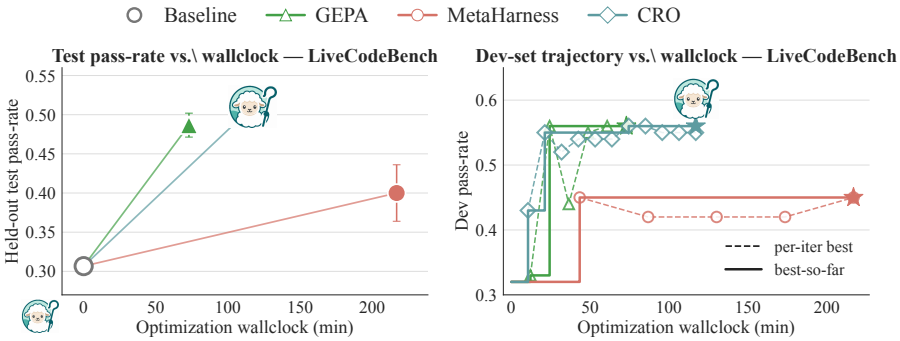


Figure 4: LiveCodeBench comparison. **Left:** held-out test pass-rate versus optimization wall-clock. **Right:** dev-set trajectory for each method across optimization wall-clock. CRO subtask-cache reuse is reported separately in Figure 5.

**Results.** CRO obtains the best performance on four of five datasets (Table 4). It has higher held-out test score and lower wall-clock simultaneously compared to MetaHarness across these datasets, with savings ranging 27–58%. Notably, on TerminalBench-2, the most execution-bound benchmark in the suite, GEPA and MetaHarness both fail to improve over the baseline on the subset under study, while CRO improves performance by 4 pts while also needing the least wall-clock. On IFBench, MetaHarness edges CRO by 1.0 pt on test (within a standard deviation) but still takes 37% longer.

We attribute CRO's downstream performance gains to two sources. First, CRO's counterfactual experiments hold the unaffected part of the computation constant, letting the model isolate and patch failures, compared to the noise introduced by MetaHarness's full-pipeline reruns; we find qualitatively that the generated hypotheses are high-quality (Appendix F). Second, SHEPHERD's Git-like execution trace lets the LLM navigate prior executions more naturally than the flat logs used by other methods. On the other hand, CRO's wall-clock savings come from three sources: counterfactual replay re-executes only the suffix downstream of each edit's first effect via SHEPHERD rather than the full pipeline, the byte-identical prefixes allow for KV-cache reuse from the LLM, and target-set gating evaluates each candidate against a small subset of the training set before committing to a full dev-set evaluation. On LiveCodeBench, computation reuse rises from ∼1% on the first cold proposer session to over 60% later in the run (Figure 5).

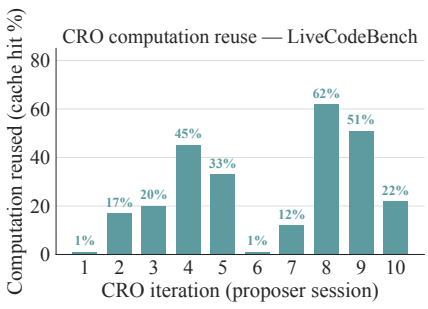


Figure 5: Computation reuse on LiveCodeBench with CRO.

### 5.3 Meta-Agent for Training: Meta-Agent Guided Tree-RL

**Motivation.** RLVR for long-horizon tasks suffers from sparse, episode-level rewards: an agent that takes dozens of steps receives a single binary signal at the end [5, 36, 32]. For fine-grained reward signals, one approach is Tree-search RL [10, 12], which derives step-level advantages from outcome rewards alone via sibling rollouts. In stateless environments, this is trivial [43]. However, with agent tasks that modify real filesystems and services, exact and cheap state-forking is required, which is provided by SHEPHERD.

**Method.** GRPO-style training for long-horizon agents samples $G$ trajectories per prompt and assigns each one a single outcome reward, distributed uniformly across all actions [29]. We recover finer-grained credit by tree-searching intermediate states [31, 53]: along each root rollout, a meta-agent picks a fork turn $t$ and we sample $K$ sibling branches forward from that state, yielding $G(K+1)$ trajectories per task at the cost of only $K$ extra branch rollouts (forking is exact and cheap, Table 3). Credit assignment operates at two levels: prefix actions before $t$ inherit the standard inter-root GRPO advantage across the $G$ roots, while suffix actions take an intra-tree advantage computed within each $(K+1)$-member fork group, surfacing per-step outcome differences without a learned value function or process reward model. The full procedure is given as Algorithm 2 in Appendix F.6, with qualitative examples of the meta-agent's branching decisions in Section F.7.

**Setup.** We use both `Qwen3.5-35B-A3B` and `Nemotron-3-Super-120B-A12B` as the base models [28, 26], training via tinker [18]. We train on a filtered subset of the **Endless Terminals** corpus [7]: starting from the 2,492-task pool, we drop tasks where the base policy passes all 8 sampled rollouts (pass@8=1.0), leaving 442 tasks for Qwen3.5 and 530 for Nemotron-3. We hold out **Terminal-Bench 2.0** [24] as an out-of-distribution test set evaluated every 10 training steps. We compare two RL setups at matched generation compute: **Flat GRPO**, independent root rollouts only, one group baseline across $G$=8 roots; and *Meta-Agent Guided Tree-RL* (**Tree-GRPO**): root rollouts plus $K$=4 sibling branches forked at a meta-agent-chosen turn, with the two-level baseline from Algorithm 2. The full configuration for training is deferred to Section F.6.

**Results.** The results are in Table 5. Meta-Agent Guided Tree-RL is the strongest setting on both base models. First, the uplift of Tree-GRPO over Flat GRPO is consistent across model scale: a ∼3B-active-parameter and a ∼12B-active-parameter model gain 5.2 and 3.4 points respectively. This improvement is the consequence of a training-time pattern: Tree-GRPO's produces higher reward variance on both models (Figure 19). Flat GRPO assigns each action in a trajectory the same outcome-derived advantage. Tree-GRPO's intra-tree baseline instead isolates the suffix actions taken after the fork point: $K+1$ siblings sharing a prefix differ only in what happened after the fork point, so the per-step advantage there reflects local choice quality rather than global trajectory outcome. SHEPHERD makes this affordable, and we find that the intra-tree baseline produces more informative gradient steps as well. This behavior transfers to the Endless Terminals validation split (Figure 18, deferred to Section F.6).

Table 5: Held-out Terminal-Bench 2.0 avg@5 (89 tasks, 5 seeds). Settings in Section F.6.

| Method | Qwen3.5-35B-A3B | Nemotron-3-Super-120B-A12B |
|---|---|---|
| Base | 26.1%$_{\pm 4.21}$ | 30.3%$_{\pm 3.62}$ |
| Flat GRPO | 34.2%$_{\pm 4.05}$ | 33.8%$_{\pm 3.41}$ |
| **Tree-GRPO** | **39.4%**$_{\pm 3.87}$ | **37.2%**$_{\pm 3.19}$ |

## 6 Conclusion

We presented SHEPHERD, a substrate that lets a meta-agent hold, inspect, branch, and rewrite another agent's execution as a first-class object, like functions in functional programming. Meta-agents in SHEPHERD are higher-order agents over agentic execution, as we show across three applications: live coordination of parallel coding agents, counterfactual replay for workflow optimization, and Tree-GRPO for finer-grained credit assignment in long-horizon RL. More broadly, SHEPHERD opens a path toward meta-agent systems that are active operators over agentic execution. Future agents could use the substrate to compress execution traces into reusable workflows, run counterfactual agentic interpretability probes, safely gate irreversible actions, or learn policies for managing agentic execution. As agentic systems become longer-lived, more stateful, and more consequential, we believe this

execution-level control will become a core abstraction for effective meta-agents. **SHEPHERD** is a step toward making that abstraction readily available and programmable.

## References


[1] https://www.anthropic.com/engineering/managed-agents. URL https://www.anthropic.com/engineering/managed-agents.

[2] Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning, July 2025. URL http://arxiv.org/abs/2507.19457. arXiv:2507.19457 [cs].

[3] Parth Asawa, Alan Zhu, Abby O'Neill, Matei Zaharia, Alexandros G. Dimakis, and Joseph E. Gonzalez. How to Train Your Advisor: Steering Black-Box LLMs with Advisor Models, October 2025. URL http://arxiv.org/abs/2510.02453. arXiv:2510.02453 [cs].

[4] Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent llm systems fail?, 2025. URL https://arxiv.org/abs/2503.13657.

[5] Wentse Chen, Jiayu Chen, Hao Zhu, and Jeff Schneider. Context-lite multi-turn reinforcement learning for LLM agents. In *ES-FoMo III: 3rd Workshop on Efficient Systems for Foundation Models*, 2025. URL https://openreview.net/forum?id=6CE5PLsZdW.

[6] Ching-An Cheng, Allen Nie, and Adith Swaminathan. Trace is the Next AutoDiff: Generative Optimization with Rich Feedback, Execution Traces, and LLMs, June 2024. URL https://arxiv.org/abs/2406.16218. arXiv:2406.16218.

[7] Kanishk Gandhi, Shivam Garg, Noah D. Goodman, and Dimitris Papailiopoulos. Endless Terminals: Scaling RL Environments for Terminal Agents, January 2026. URL https://arxiv.org/abs/2601.16443. arXiv:2601.16443.

[8] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset, 2021. URL https://arxiv.org/abs/2103.03874.

[9] Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework, August 2023. URL https://arxiv.org/abs/2308.00352. arXiv:2308.00352.

[10] Zhenyu Hou, Ziniu Hu, Yujiang Li, Rui Lu, Jie Tang, and Yuxiao Dong. Treerl: Llm reinforcement learning with on-policy tree search, 2025. URL https://arxiv.org/abs/2506.11902.

[11] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024. URL https://arxiv.org/abs/2403.07974.

[12] Yuxiang Ji, Ziyu Ma, Yong Wang, Guanhua Chen, Xiangxiang Chu, and Liaoni Wu. Tree search for llm agent reinforcement learning, 2026. URL https://arxiv.org/abs/2509.21240.

[13] Yichen Jiang, Shikha Bordia, Zheng Zhong, Charles Dognin, Maneesh Singh, and Mohit Bansal. Hover: A dataset for many-hop fact extraction and claim verification, 2020. URL https://arxiv.org/abs/2011.03088.

[14] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

[15] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines, October 2023. URL https://arxiv.org/abs/2310.03714. arXiv:2310.03714.

[16] Arpandeep Khatua, Hao Zhu, Peter Tran, Arya Prabhudesai, Frederic Sadrieh, Johann K. Lieberwirth, Xinkai Yu, Yicheng Fu, Michael J. Ryan, Jiaxin Pei, and Diyi Yang. CooperBench: Why Coding Agents Cannot be Your Teammates Yet, January 2026. URL https://arxiv.org/abs/2601.13295. arXiv:2601.13295.

[17] Joongwon Kim, Wannan Yang, Kelvin Niu, Hongming Zhang, Yun Zhu, Eryk Helenowski, Ruan Silva, Zhengxing Chen, Srinivasan Iyer, Manzil Zaheer, Daniel Fried, Hannaneh Hajishirzi, Sanjeev Arora, Gabriel Synnaeve, Ruslan Salakhutdinov, and Anirudh Goyal. Scaling Test-Time Compute for Agentic Coding, 2026. URL https://arxiv.org/abs/2604.16529.

[18] Thinking Machines Lab. Tinker, 2025. URL https://thinkingmachines.ai/tinker/.

[19] Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-End Optimization of Model Harnesses, March 2026. URL https://arxiv.org/abs/2603.28052. arXiv:2603.28052.

[20] Yoonsang Lee, Howard Yen, Xi Ye, and Danqi Chen. Agentic Aggregation for Parallel Scaling of Long-Horizon Agentic Tasks, 2026. URL https://arxiv.org/abs/2604.11753.

[21] Hanchen Li, Runyuan He, Qizheng Zhang, Changxiu Ji, Qiuyang Mang, Xiaokun Chen, Lakshya A. Agrawal, Wei-Liang Liao, Eric Yang, Alvin Cheung, James Zou, Kunle Olukotun, Ion Stoica, and Joseph E. Gonzalez. Combee: Scaling Prompt Learning for Self-Improving Language Model Agents, April 2026. URL http://arxiv.org/abs/2604.04247. arXiv:2604.04247 [cs].

[22] Yang Li, Siqi Ping, Xiyu Chen, Xiaojian Qi, Zigan Wang, Ye Luo, and Xiaowei Zhang. AgentGit: A Version Control Framework for Reliable and Scalable LLM-Powered Multi-Agent Systems, November 2025. URL https://arxiv.org/abs/2511.00628. arXiv:2511.00628.

[23] Fulin Lin, Shaowen Chen, Ruishan Fang, Hongwei Wang, and Tao Lin. Stop wasting your tokens: Towards efficient runtime multi-agent systems. *arXiv preprint arXiv:2510.26585*, 2025. URL https://arxiv.org/abs/2510.26585.

[24] Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

[25] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

[26] NVIDIA. Nvidia nemotron 3: Efficient and open intelligence, 2025. URL https://arxiv.org/abs/2512.20856. White Paper.

[27] Valentina Pyatkin, Saumya Malik, Victoria Graf, Hamish Ivison, Shengyi Huang, Pradeep Dasigi, Nathan Lambert, and Hannaneh Hajishirzi. Generalizing verifiable instruction following, 2025. URL https://arxiv.org/abs/2507.02833.

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

[29] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

[30] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning, March 2023. URL https://arxiv.org/abs/2303.11366. arXiv:2303.11366.

[31] RS Sutton and AG Barto. Reinforcement learning: An introduction 1st edition. *Exp. Psychol. Learn. Mem. Cogn*, 30:1302–1321, 1998.

[32] Hui-Ze Tan, Xiao-Wen Yang, Hao Chen, Jie-Jing Shao, Yi Wen, Yuteng Shen, Weihong Luo, Xiku Du, Lan-Zhe Guo, and Yu-Feng Li. Hindsight credit assignment for long-horizon llm agents, 2026. URL https://arxiv.org/abs/2603.08754.

[33] Sandra Wachter, Brent Mittelstadt, and Chris Russell. Counterfactual explanations without opening the black box: Automated decisions and the gdpr. *Harvard Journal of Law & Technology*, 31(2):841–887, 2018.

[34] Cong Wang and Yusheng Zheng. Fork, Explore, Commit: OS Primitives for Agentic Exploration, February 2026. URL https://arxiv.org/abs/2602.08199. arXiv:2602.08199.

[35] Pengcheng Wang, Jerry Huang, Jiarui Yao, Rui Pan, Peizhi Niu, Yaowenqi Liu, Ruida Wang, Renhao Lu, Yuwei Guo, and Tong Zhang. AgentSPEX: An Agent SPecification and EXecution Language, 2026. URL https://arxiv.org/abs/2604.13346.

[36] Ruiyi Wang and Prithviraj Ammanabrolu. A practitioner's guide to multi-turn agentic reinforcement learning, 2025. URL https://arxiv.org/abs/2510.01132.

[37] Xingyao Wang, Jiayi Pan, Binyuan Hui, et al. OpenHands V1: Event-sourced state management for multi-agent coding systems. *MLSys*, 2026. URL https://arxiv.org/abs/2511.03690. arXiv:2511.03690.

[38] Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, Xinyu Yang, Zeyi Huang, Xuehai He, Luyao Ma, Baolin Peng, Hao Cheng, Pengcheng He, Weizhu Chen, Shuohang Wang, Simon Shaolei Du, and Yelong Shen. ThetaEvolve: Test-time Learning on Open Problems, 2025. URL https://arxiv.org/abs/2511.23473.

[39] Tianxin Wei, Noveen Sachdeva, Benjamin Coleman, Zhankui He, Yuanchen Bei, Xuying Ning, Mengting Ai, Yunzhe Li, Jingrui He, Ed H. Chi, Chi Wang, Shuo Chen, Fernando Pereira, Wang-Cheng Kang, and Derek Zhiyuan Cheng. Evo-Memory: Benchmarking LLM Agent Test-time Learning with Self-Evolving Memory, 2025. URL https://arxiv.org/abs/2511.20857.

[40] Zhaotian Weng, Antonis Antoniades, Deepak Nathani, Zhen Zhang, Xiao Pu, and Xin Eric Wang. Group-Evolving Agents: Open-Ended Self-Improvement via Experience Sharing, 2026. URL https://arxiv.org/abs/2602.04837.

[41] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation, August 2023. URL https://arxiv.org/abs/2308.08155. arXiv:2308.08155.

[42] Peng Xia, Kaide Zeng, Jiaqi Liu, Can Qin, Fang Wu, Yiyang Zhou, Caiming Xiong, and Huaxiu Yao. Agent0: Unleashing Self-Evolving Agents from Zero Data via Tool-Integrated Reasoning, 2025. URL https://arxiv.org/abs/2511.16043.

[43] Yuxi Xie, Anirudh Goyal, Wenyue Zheng, Min-Yen Kan, Timothy P. Lillicrap, Kenji Kawaguchi, and Michael Shieh. Monte carlo tree search boosts reasoning via iterative preference learning, 2024. URL https://arxiv.org/abs/2405.00451.

[44] Alex L. Zhang, Tim Kraska, and Omar Khattab. Recursive language models, 2025. URL https://arxiv.org/abs/2512.24601.

[45] Alex L. Zhang, Zhening Li, and Omar Khattab. The Mismanaged Geniuses Hypothesis, 2026. URL https://alexzhang13.github.io/blog/2026/mgh/. Blog post.

[46] Guibin Zhang, Junhao Wang, Junjie Chen, Wangchunshu Zhou, Kun Wang, and Shuicheng Yan. Agentracer: Who is inducing failure in the llm agentic systems?, 2025. URL https://arxiv.org/abs/2509.03312.

[47] Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel Machine: Open-Ended Evolution of Self-Improving Agents, 2025. URL https://arxiv.org/abs/2505.22954.

[48] Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents, March 2026. URL https://arxiv.org/abs/2603.19461. arXiv:2603.19461.

[49] Jiayi Zhang, Simon Yu, Derek Chong, Anthony Sicilia, Michael R. Tomz, Christopher D. Manning, and Weiyan Shi. Verbalized sampling: How to mitigate mode collapse and unlock LLM diversity. In *arXiv preprint arXiv:2510.01171*, 2025. URL https://arxiv.org/abs/2510.01171.

[50] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models, 2025. URL https://arxiv.org/abs/2510.04618.

[51] Shengtao Zhang, Jiaqian Wang, Ruiwen Zhou, Junwei Liao, Yuchen Feng, Zhuo Li, Yujie Zheng, Weinan Zhang, Ying Wen, Zhiyu Li, Feiyu Xiong, Yutao Qi, Bo Tang, and Muning Wen. MemRL: Self-Evolving Agents via Runtime Reinforcement Learning on Episodic Memory, 2026. URL https://arxiv.org/abs/2601.03192.

[52] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language Agent Tree Search Unifies Reasoning Acting and Planning in Language Models, October 2023. URL https://arxiv.org/abs/2310.04406. arXiv:2310.04406.

[53] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. Monte carlo tree search: a review of recent modifications and applications. *Artificial Intelligence Review*, 56(3):2497–2562, 2022. ISSN 1573-7462. doi: 10.1007/s10462-022-10228-y. URL http://dx.doi.org/10.1007/s10462-022-10228-y.

# A Limitations and Future Works

## A.1 Limitations

**Proof-of-existence framing.** Each of our three case studies is reported as a proof of existence: the substrate primitives suffice to drive a meaningful uplift on a representative dataset, given a meta-agent we wrote for that case. We do not claim optimality of the meta-agent policies, robustness across model

families and benchmarks at the scale of, e.g., a benchmark sweep, or that the headline numbers cannot be matched without SHEPHERD. Establishing a full burden of proof, in the sense of head-to-head comparisons against every plausible alternative substrate and policy, is its own paper per case study and is outside the scope of this work.

**Supervision and proposer cost.** The live-supervision and CRO results assume access to a meta-agent strong enough to act usefully (Sonnet 4.6 / Opus 4.7 in §5.1; GPT-5.4 in §5.2). For short tasks, the meta-agent's token cost can exceed the worker's, just as for existing meta-optimisers [2, 19]. We report total dollar cost per arm in Appendix F.4; the regime where this trade-off is favourable depends on task length and on the cost ratio between the worker and the meta-agent, which we do not characterise here.

**Counterfactual replay assumes weak coupling between edits and side effects.** CRO replays a candidate edit's suffix from the first event whose dependencies are affected by the edit. When an edit touches a component whose effects propagate widely (e.g. the system prompt of a tool used in every step), the suffix is the entire trajectory and the cache buys nothing. We observe this regime on the cold first proposer session of every dataset (Appendix F.4); it amortises away within two to three sessions on the benchmarks we study.

### A.2 Future Works

**Agent interpretability.** Mechanistic interpretability of agentic systems today is largely observational: a transcript is annotated, a probe is fit, conclusions are drawn. SHEPHERD's coupled fork turns these into testable counterfactual interventions. Editing a single component (a tool, a system-prompt span, a sub-task definition) and replaying the suffix from the first commit it affects holds every other source of variance fixed, isolating the contribution of that component to the eventual outcome. CRO's propose-and-replay machinery is one instance of this loop driven by a task-success objective; a natural next step is to run the same loop under an interpretability objective, e.g. minimal edits that flip a specific decision, or the smallest prompt span whose removal preserves task success, turning CRO into a tool for explanation rather than optimisation.

**Reversible environments for continual learning.** The Tree-GRPO results in §5.3 work because forking the agent's filesystem and processes is essentially free. The same primitive applies to any domain whose state lives on disk or in a sandboxed process tree: computer-use, web browsing, and large-codebase edit tasks all fit. Once forks are cheap, a single agent can train over weeks of interaction rather than a single rollout episode, with the execution trace maintaining a coherent history of every revisit, every backtrack, and every retried tool call. Continual learning then becomes a question of scheduling reads against this graph rather than re-engineering the substrate.

**Post-training agents to use the substrate.** The case studies in §5 fix the meta-agent and let it govern a base worker. The dual question is whether the worker itself can be post-trained to use SHEPHERD's primitives natively: an action space that includes `fork`, `discard`, and `replay` over its own typed effect stream, not just tool calls into the environment. This is strictly harder than tool-use post-training because the model has to learn when to back up, when to retry, and when to spawn a sibling, not just which tool to invoke. The substrate provides exactly the typed state needed to define these actions and to compute exact returns over them.

**Reversible sandboxes as a safety property.** Every effect on the substrate carries a reversibility tier (§3.2): reversible filesystem mutations, compensable side effects, and irreversible external calls each have a different rollback contract. This makes SHEPHERD a candidate substrate for safety-critical agent deployments: an external override can preempt the materialisation of any irreversible effect, downstream checks can fire a rollback up to the last materialisation point, and the execution trace gives auditors a content-addressed record of what actually happened rather than what the agent reported. The substrate does not solve the alignment problem, but it makes the standard halt-and-inspect loop something a deployment system can implement against a stable interface rather than bespoke per-agent plumbing.

## B Mechanized Core and Proof Envelopes

**SHEPHERD** separates the production runtime from the semantic object that is mechanized in Lean. The production framework executes ordinary Python tasks, provider SDK calls, shell commands, sandbox operations, retries, scheduling, and carrier storage. Those executions are not themselves verified. The verified artifact is a small algebraic-effects trace machine, together with proof-backed profiles for static fragments whose lowered traces fall inside its boundary.

**Claim tiers.** Each inspectable run may carry a proof envelope with a profile and an explicit strength. The profile is one of `runtime_only`, `reference_core_a`, `core0`, `core_a`, `core0h`, or `extension`; the strength is one of `runtime_only`, `reference_validated`, `forward_simulation`, or `semantic_adequacy`. Ordinary Python runs default to `runtime_only`. A trace becomes `reference_core_a` when it is accepted by the executable kernel-v3 reference validator but does not yet claim Lean theorem coverage. A trace becomes `core0` or `core_a` only when the static lowering evidence, generated-trace validation, and Lean-side fragment assumptions all match the completed generated trace. Core-0H is sidecar-gated: it can receive a `forward_simulation` envelope only when a validated content-addressed Core-0H sidecar manifest is attached, and the classifier does not infer it from arbitrary structured handlers. Core-0/Core-A envelopes claim `semantic_adequacy`. Core-A proof-backed envelopes currently require exactly one selected direct abort capture, no selected abort-path resume/return, and no selection-closure suffix; handler-side effects, sequencing before abort, multiple abort captures, or unused abort-only handler definitions remain reference-validatable. Incomplete or live prefixes remain reference-validatable unless a future prefix certificate is attached. Publication controls such as forwarding, terminal delay/fork, and replay remain `extension` unless a future proof envelope names a stronger theorem.

**Lean theorem surface.** The Lean development builds with `lake build` in the kernel-v3 proof artifact. The current theorem surface used by the proof envelope is:

Table 6: Mechanized theorem surface used by the proof-envelope claim.

| Profile | Representative Lean theorem | Meaning |
|---|---|---|
| Core-0 | `source_eval_to_machine`; `core0_machine_eval_to_source` | Forward source-to-machine simulation and restricted reverse simulation for the ordinary callable-resumption fragment. |
| Core-A | `source_eval_to_machine`; `coreA_machine_eval_to_source` | Core-0 plus the direct abort-without-resume handler boundary currently admitted by the envelope. |
| Core-0H | `core0h_source_eval_to_machine` | Forward simulation for deterministic two-phase handler bodies with matching evidence; this profile is currently forward-only. |
| Trace monotonicity | `trace_monotonic`; `core0h_trace_monotonic` | Machine execution extends traces by appending records rather than rewriting prior trace prefixes. |
| Branch replay skeleton | `single_child_branch_replay_sound` | One-child structural replay soundness for an exact suffix replay model; this is not a production replay refinement proof. |

**Executable reference boundary.** The Python `agentic-kernel-v3-reference` package in the submitted code package sits between the production runtime and the Lean development. It validates Core-0/Core-A trace lifecycles, reruns a static `KernelProgram` to check exact generated-trace agreement, and emits an explicit `ProofEnvelope`. Public `Run[T]` values carry a `proof` field; current production Python executions default to `runtime_only`. The envelope records a content-addressed evidence identifier over the proof authority, validator, program digest, and trace digest. Kernel envelopes derive `proof_backed` from `proof_strength`, so profile names alone do not upgrade a trace. Runtime metadata is carrier metadata: non-runtime strengths must cite kernel-v3 reference provenance and a `proof-evidence:sha256` identifier, but public `Run[T]` metadata exposes such imports as `claimed_proof_backed` rather than runtime-verified proof authority. Lean theorem ids are centralized in a proof-surface ABI table checked by a Lean module with `#check` commands and typed signature wrappers, so stale theorem names or materially changed theorem statements fail the artifact build. Live prefixes and structured handler bodies without validator-issued Core-0H two-phase sidecar manifests are classified as reference-validatable rather than proof-backed. The artifact gate is `make verify-proof-envelope-claim`. This makes the formal claim inspectable without implying that all executable Agentic programs are proof-backed.

**Non-claims.** The proof envelope does not verify arbitrary Python control flow, provider SDK behavior, model outputs, prompt-cache state, shell commands, filesystem mutation correctness, Docker or sandbox implementations, meta-git carrier storage, scheduling, cancellation, retries, recovery, or multi-branch replay. The meta-agent applications in the main paper rely on the production substrate plus empirical validation; the Lean artifact supplies the semantic core and the boundary discipline for proof-backed fragments.

# C Framework Performance: Extended Results

This appendix groups all extended results that support Section 4: the measurement protocol (Appendix C.1), the mutation-size sweep that defuses the large-file concern (Appendix C.2), the agent-turn latency reference behind the "2–3% of one turn" claim (Appendix C.3), depth scaling under stacked overlay layers (Appendix C.4), the effect-stream observation overhead (Appendix C.5), the KV-cache reuse breakdown (Appendix C.6), and the cross-backend portability check (Appendix C.7).

## C.1 Measurement Protocol

**Hardware.** The Table 3 measurements for SHEPHERD, `docker commit`, and full rootfs copy run on the same Vultr cloud instance (2 vCPU, 16 GB RAM, SSD, Ubuntu 22.04, Docker 29.3.1, overlay2 storage driver on extfs). Modal numbers come from Modal's hosted gVisor runtime (separate hardware) and are reproduced from a fresh benchmark whose latency, therefore, additionally includes network round-trip from the host-side agent. The Table 8 measurements use E2B Firecracker microVMs. Cross-backend numbers (Appendix C.7) use E2B (Firecracker), Modal (gVisor), and Daytona (managed Linux containers).

**Workloads.** Table 3 uses three real Terminal-Bench 2.0 Docker images spanning two orders of magnitude: *openssl-selfsigned-cert* (42 MB), *caffe-cifar-10* (200 MB), and *pytorch-model-recovery* (5.8 GB). Full copy tars the entire container rootfs, excluding `/proc`, `/sys`, `/dev`, and `/tmp`, and is $O(n)$ in image size. `docker commit`, Modal `snapshot_filesystem()`, and SHEPHERD's overlay delta are all $O(1)$ in image size on the overlay2 driver. Table 8 additionally uses three synthetic working directories: *small* (10 files, 30 KB), *medium* (100 files, 100 MB), *large* (100 files, 1 GB). KV-cache experiments use real Terminal-Bench 2.0 tasks with a Haiku 4.5 agent.

**Per-step mutation pattern.** The agent's writes inside the workdir are simulated with a fixed 5+3 random-file pattern: 5 baseline writes establish the parent state *before* the checkpoint, then each of the $K-1{=}3$ sibling branches adds 1 more random write *after* the fork, all written into the measured workdir. Each write defaults to 10,KB. Table 3's storage columns count only the post-fork branch deltas: the **Storage** column reports a single sibling's overlay delta (1 write = 10,KB); the **Disk @ K=4** column reports the sum of the three siblings' overlay deltas ($3 \times 10, \text{KB} = 30, \text{KB}$). The 5 baseline writes (50,KB) sit in the parent's overlay and are not counted there. Table 7 in Section C.2 reports the *full* overlay upper directory (baseline + branches, $8 \times 10, \text{KB} \approx 80, \text{KB}$ at this default size) and sweeps the per-write size from 1,KB to 100,MB to verify the substrate scales 1:1 with what the agent writes and adds no multiplier on top.

**Pattern A: agent-on-host, sandbox-in-container.** The fork/revert latencies in Table 3 measure agent-perceived wall-clock under Pattern A, which matches how production agents (mini-swe-agent, SWE-Agent, smolagents) use sandboxes today. A Python agent process on the host imports `litellm` + `tenacity` (matching Terminus-2's startup), warms an OpenAI `gpt-5.4-mini` client with one call, then drives the sandbox container through a tmux session. Each operation is timed from "operation issued" until the new sandbox's tmux returns to a probe bash command (`echo READY_<uuid>` + `capture-pane` until the marker appears). Method-specific operations:

- `docker commit`: Fork = `docker commit` + `docker run -d` new container + tmux ready in new container. Revert = `docker rm -f` + `docker run -d` from saved image + tmux ready. Both ops are required; measuring only `docker run` understates by the rm/commit cost.
- Modal: Fork = `snapshot_filesystem()` + `Sandbox.create(image=snap)` + tmux ready. Revert = `terminate()` + `Sandbox.create(image=snap)` + tmux ready.
- SHEPHERD: Fork = overlay-layer fork (in-container, same tmux session). Revert = overlay umount/remount of saved layer (in-container, same tmux session).

- BranchFS [34]: BranchFS daemon runs on the host with a FUSE mount; the TB2 image is bind-mounted into a Docker container at `/workspace` so the agent's tmux can read/write through the FUSE layer. Fork = `branchfs create`, which creates a copy-on-write branch and auto-switches the mount, then tmux probe. Revert = `branchfs abort`, which discards the leaf branch's delta and switches the mount back to the parent, then tmux probe. BranchFS branch ops are O(1) in base directory size, so fork/revert latency is image-independent (confirmed across 42 MB / 200 MB / 5.8 GB images: ~270 ms fork, ~358 ms revert).
- Full copy: latency reported in Table 3 is computed (not measured) as rootfs `tar`/`restore` (Vultr) plus an empirically observed ∼340 ms container-startup + tmux-probe overhead. Running the full agent-revert protocol takes hours per cell on the 5.8 GB image (each fork moves the full rootfs tar across two `docker cp` boundaries), so we report the computed values.

**Disk and RAM at $K{=}4$.** Disk @ K=4 is the host disk delta over $K{-}1$ added branches: commit-layer deltas for `docker commit`, full rootfs tars for `full copy` (computed = $(K{-}1)\times$ per-branch tar size), overlay-layer files for **SHEPHERD**, BranchFS storage tree (`/var/lib/branchfs`) over $K$ branches. RAM @ K=4 is summed RSS via `docker stats -no-stream` over the alive containers ($K$ for `docker commit` / `full copy`, where each branch needs a separate restarted container; 1 container reading for **SHEPHERD**, which forks the worker process per branch but reuses one host container as the overlay-layer host (the K forked worker processes are aggregated under that one RSS reading); 1 container + BranchFS daemon RSS for BranchFS). Modal's per-sandbox host RAM/disk is hidden behind Modal's runtime, so its $K{=}4$ cells are marked —.

**Protocol.** Each benchmark begins with 2–3 warm-up iterations (discarded) to prime filesystem caches and JIT paths. Measurements are wall-clock `time.monotonic()` around the operation under test. For Table 3 fork/revert latency: 10 repetitions per cell for **SHEPHERD** / `docker commit` / Modal, 5 repetitions for full-copy storage measurements; 3 repetitions for the $K{=}4$ resource columns. Error bars are $\pm 1\sigma$. Storage is measured via `du -sb` on the overlay upper directory (for **SHEPHERD**) or the checkpoint artifact (for baselines). Bench code is shared under `exp/framework-perf`, please refer to the readme.

### C.2 Substrate scaling under varying mutation size

A natural reviewer concern about the 10 KB delta in Table 3 is whether the substrate stays small when the agent writes large files. We sweep the per-step write size from 1 KB to 100 MB on the 5.8 GB pytorch-model-recovery image, keeping the same 5+3 mutation pattern but changing how many bytes each write puts into the workdir. Table 7 reports the resulting overlay disk delta and fork latency across three repetitions per size.

Two findings: (a) the disk delta tracks what the agent writes 1:1 (8 KB at 1 KB/step up to 800 MB at 100 MB/step), so the substrate adds zero overhead beyond the agent's actual emissions; (b) fork and revert latency stay flat at ∼340 ms across all sizes, since both operations only swap overlay metadata, not file contents. We additionally verify revert correctness across the same sweep: in 12 reps spanning all four sizes, the workdir after revert exactly matches the pre-fork state (5 baseline files preserved, 3 post-fork branch files cleanly discarded; 12/12 PASS). Absolute latencies here are about 200 ms higher than Table 3 because this sweep ran on Mac/Docker Desktop's Linux VM, which adds `docker exec` overhead per call; the scaling shape is what the table claims.

Table 7: Substrate scaling under varying per-step mutation size on the 5.8 GB pytorch-model-recovery image (5+3 random-write pattern, $K{=}4$ branches, 3 reps; medians). The disk delta scales 1:1 with what the agent writes; fork and revert latency stay flat. Revert correctness (post-revert workdir matches pre-fork state) is 12/12 PASS across the four sizes.

| Per-step write | Total written | Disk delta ↓ | Fork (ms) ↓ | Revert (ms) ↓ |
|---|---|---|---|---|
| 1 KB | 8 KB | 8.4 KB | 339 | 348 |
| 10 KB | 80 KB | 80.4 KB | 386 | 346 |
| 1 MB | 8 MB | 8.0 MB | 331 | 385 |
| 100 MB | 800 MB | 800 MB | 343 | 366 |

### C.3 Agent per-turn latency reference

To anchor the claim that SHEPHERD's fork is small relative to a typical agent turn, we instrumented the bench harness with per-turn wall-clock timing and ran two Terminal-Bench 2.0 tasks for up to 20 turns each (Anthropic Claude Haiku 4.5, E2B sandbox). Across 17 measured turns we observe:

- **LLM call** (Anthropic API round-trip): mean 5.36 s, median 5.56 s, p10/p90 = 2.69 / 7.64 s.
- **Tool call** (sandbox bash execution): mean 0.20 s, median 0.17 s, p10/p90 = 0.08 / 0.61 s.
- **Per-turn total**: mean 5.50 s, median 5.81 s, p10/p90 = 2.70 / 7.77 s.

The LLM call dominates ( 98% of per-turn wall-clock); the tool call is small here because Terminal-Bench tasks involve mostly light bash (file reads, package installs). Heavier tool actions (compilation, training) push per-turn time well above the median. SHEPHERD's 134–143 ms fork (Table 3) is therefore around 2–3% of a typical Haiku 4.5 turn and well below the noise floor of one LLM call's response-time variance.

### C.4 Scaling Behaviour

Table 8 reports how checkpoint/revert latency behaves as the number of stacked overlay layers grows (left) and as the effect stream accumulates events (right). SHEPHERD's overlay checkpoint remains in the 157–252 ms band (on E2B) through 50 stacked layers; `docker commit` is roughly constant as well but at a 2.8× higher baseline (451–558 ms). The OverlayFS lower-directory chain is bounded by the kernel's page-size limit at approximately 60 layers; trajectories exceeding this depth require periodic compaction of frozen layers.

Per-event effect-stream overhead (record and observe) is constant at approximately 120 ms on E2B (network-dominated) through 200 steps. Stream size grows linearly at ∼130 B/event.

Table 8: **Left:** Checkpoint/revert latency (ms) as overlay layers stack (E2B), compared to `docker commit` (Vultr). **Right:** Per-event effect-stream overhead (E2B) as trajectories grow.

| | Agentic | | Docker commit | |
|---|---|---|---|---|
| **Depth** | **Ckpt** | **Revert** | **Commit** | **Revert** |
| 1 | 173 | 255 | 451 | 300 |
| 5 | 157 | 170 | 524 | 255 |
| 10 | 252 | 167 | 481 | 258 |
| 25 | 179 | 170 | 558 | 260 |
| 50 | 237 | 167 | 503 | 297 |

| **Steps** | **Record** | **Observe** | **Stream** |
|---|---|---|---|
| 1 | 106 ms | 107 ms | 134 B |
| 10 | 127 ms | 178 ms | 1.3 KB |
| 50 | 109 ms | 177 ms | 6.5 KB |
| 100 | 197 ms | 117 ms | 13.1 KB |
| 200 | 171 ms | 115 ms | 26.4 KB |

### C.5 Observe Overhead Detail

Table 9 compares effect-stream recording throughput on a local Docker host (no network roundtrip) versus E2B Firecracker (remote API). The local overhead is 3.1 ms per event (5%); the E2B figure (113 ms, 87%) is dominated by the network roundtrip for each `exec` call and does not reflect framework serialization cost. We separately verified that subscribing a supervisor to the effect stream adds exactly zero tokens to the worker's context by comparing the worker's message list with and without a supervisor attached: the two lists are byte-identical across a 10-step trajectory.

Table 9: **Left:** Effect-stream recording throughput, local vs. remote. **Right:** Context inflation test: supervisor subscription adds 0 tokens to the worker.

| **Metric** | **Local** | **E2B** |
|---|---|---|
| Raw throughput | 17 evt/s | 8 evt/s |
| Logged throughput | 16 evt/s | 4 evt/s |
| Overhead / event | 3.1 ms (5%) | 113 ms (87%) |
| Observe latency | 64 ms | 104 ms |

| **Condition** | **Worker context** |
|---|---|
| Without supervisor | 21 msgs, 1449 chars |
| With supervisor | 21 msgs, 1449 chars |
| Context inflation | **0 chars (0.0%)** |

### C.6 KV-Cache Reuse Detail

Table 10 reports the per-task, per-fork-depth view behind the §3 summary. Each task is run once on the Anthropic API with Claude Haiku 4.5 to generate an initial trajectory, with checkpoints saved at fork depths step 10, step 25, and step 50 (the third only fires when the trajectory reaches that depth). At each saved fork point we then run $K$ branches: each branch reverts the sandbox to the checkpoint, restores the LLM message prefix with `cache_control: {"type": "ephemeral"}` on the last prefix message, and continues sampling. The provider's prompt cache (5-minute TTL) charges $0.10\times$ the input-token rate for resolved-prefix tokens and $1.25\times$ the rate for the first branch's cache write; branches 2 through $K$ amortise the write across additional reads, which is why savings climb sharply $K{=}1 \rightarrow K{=}2$ and stabilise after.

Table 10: Per-task KV-cache reuse on the Anthropic API (Claude Haiku 4.5) across 8 Terminal-Bench 2.0 tasks. Each cell reports *savings% / hit%* for $K$ branches forked and replayed from a checkpoint at the listed step depth. The hit rate is the substrate-fidelity check (does `revert` restore the LLM message prefix byte-for-byte); the plateau at $\sim$95% from $K{=}2$ onwards is within 5% of the 100% ceiling. Savings climb sharply $K{=}1 \rightarrow K{=}2$ as the cache-write penalty amortises across one extra branch, then stabilise as per-branch suffix generation grows linearly in $K$. Empty cells are fork-depth/$K$ combinations not reached within the per-task wall-clock budget.

| | | **Branching factor $K$** | | | | |
|---|---|---|---|---|---|---|
| **Task** | **Fork** | $K{=}1$ | $K{=}2$ | $K{=}4$ | $K{=}8$ | $K{=}16$ |
| openssl-selfsigned-cert | step 10 | 61 / 83 | 72 / 96 | 70 / 95 | 72 / 96 | 71 / 95 |
| | step 25 | 60 / 79 | 79 / 98 | 79 / 97 | 79 / 98 | 80 / 98 |
| nginx-request-logging | step 10 | 59 / 87 | 61 / 93 | 63 / 93 | 62 / 93 | 62 / 93 |
| | step 25 | 66 / 88 | — | — | — | — |
| build-cython-ext | step 10 | 60 / 87 | 67 / 95 | 67 / 95 | 67 / 95 | 67 / 94 |
| | step 25 | 68 / 89 | 78 / 97 | 77 / 97 | 77 / 97 | — |
| configure-git-webserver | step 10 | 62 / 88 | 68 / 94 | 67 / 94 | 68 / 95 | 68 / 95 |
| | step 25 | 62 / 82 | 76 / 97 | 79 / 97 | 77 / 97 | — |
| feal-differential-cryptanalysis | step 10 | 57 / 86 | 63 / 93 | 63 / 93 | 63 / 93 | — |
| llm-inference-batching-scheduler | step 10 | 55 / 83 | 63 / 92 | 63 / 92 | 64 / 93 | 64 / 93 |
| make-doom-for-mips | step 10 | 56 / 84 | 68 / 93 | 64 / 93 | 59 / 91 | 62 / 92 |
| | step 25 | 64 / 88 | 73 / 96 | 75 / 97 | 74 / 97 | 74 / 97 |
| | step 50 | 74 / 89 | 84 / 99 | 85 / 99 | 84 / 99 | — |
| pytorch-model-recovery | step 10 | 58 / 86 | 65 / 93 | 64 / 92 | 64 / 92 | 64 / 93 |
| | step 50 | 67 / 85 | 82 / 98 | 82 / 98 | — | — |
| **Mean** | | **62 / 86** | **71 / 95** | **71 / 95** | **70 / 95** | **68 / 94** |

All Anthropic measurements use `cache_control: {"type": "ephemeral"}` on the last prefix message; the provider's prompt cache (5-minute TTL) serves the prefix at 10% of the normal input-token price. Tasks whose initial-prompt prefix falls below Haiku 4.5's 4,096-token minimum cacheable threshold are excluded; in our Terminal-Bench 2.0 sample this filter drops one task (*fix-git*, 157-character instruction).

### C.7 Realization across Sandbox Backends

The primitives of Section 3 are realized over the overlay-filesystem and checkpoint facilities exposed by modern containerized sandboxes. A single device-layer interface abstracts backend differences; application code written against the abstraction runs unchanged across providers. Table 11 summarizes compatibility and measured fork latency where available.

**Docker (local / Vultr).** Privileged containers with kernel OverlayFS on tmpfs. Checkpoint unmounts the overlay, freezes the upper directory as a named layer, and remounts with the frozen layer in the lower stack. Measured at 72 ms median (50 reps, 2 vCPU / 4 GB).

**E2B Firecracker.** Micro-VM sandboxes with OverlayFS via `sudo`. Semantics are identical to local Docker; measured latency is higher (159–169 ms) due to the remote API roundtrip. The `metacopy=on` mount option avoids full-file copy-up on `chown`.

**Modal (gVisor).** gVisor blocks `mount`/`umount` syscalls, so checkpoint uses Modal's `snapshot_filesystem()` API (935–1137 ms). Revert terminates the sandbox and spawns a new one from the snapshot image (75–79 ms).

**Daytona.** Cloud development environment with root access. OverlayFS works without `sudo`. Preliminary validation confirms all scope operations pass. Since the underlying primitive is identical to local Docker's OverlayFS (72 ms, size-independent in our `local_overlay` bench), Daytona's measured fork latency is dominated by the remote API roundtrip; we estimate $\sim$150 ms by analogy with E2B's measured +91 ms RTT (Table 11).

**Prime Intellect (gVisor).** `umount` is blocked even though `mount` succeeds, so the framework falls back to `cp -a` copies. This is $O(n)$ in working-directory size: from our `real_docker_images` bench, a small workdir ($\leq$5 MB) takes $\sim$100 ms to clone, and the same primitive scales to 2.3 s on a 44 MB rootfs and 57 s on a 6 GB rootfs (`storage_fix` bench). The fallback is therefore usable for small repositories but unsuitable for large ones.

Table 11: Cross-backend compatibility. All backends support the same scope API. Latency is wall-clock median for `Scope.fork`; 50 reps except where noted.

| Backend | Mechanism | Fork | Revert | Notes |
|---|---|---|---|---|
| Docker (local) | OverlayFS on tmpfs | 72 ms | 70 ms | Primary benchmark platform |
| E2B Firecracker | OverlayFS + `sudo` | 163 ms | 157 ms | +90 ms network roundtrip |
| Modal | `snapshot_filesystem()` | 935 ms | 79 ms | gVisor; no OverlayFS |
| Daytona | OverlayFS (root) | 150 ms | 140 ms | Validation passed; remote managed container |
| Prime Intellect | `cp -a` fallback | 100 ms | 110 ms | gVisor; $O(n)$ in workdir size |

## D Trajectory Compression: Extended Results

**Motivation.** Many real-world agent tasks are *repeatable*: a class of bug fixes, a class of data-processing scripts, a class of build-environment errors. The first solution an agent finds is typically full of exploration it did not, in retrospect, need: redundant probes, dead-end hypotheses, trial-and-error before convergence. We ask whether a meta-agent reading the completed trajectory through the effect stream can identify a fork point and a hint such that the worker, restored to the SHEPHERD scope at that point and given the hint as a system-prompt addendum, reaches the same task outcome in strictly fewer steps; i.e., *compresses* the trajectory. SHEPHERD's per-step snapshots make this cheap: the meta-agent need not commit to a fork point at trajectory-collection time, and the rerun pays only the suffix cost. Whether a compressed trajectory generalises to a reusable workflow for future invocations of the same task class is a follow-up question we leave open.

**Setup.** We evaluate on full Terminal-Bench v2.0 (88 tasks) [24] and SWE-Bench Verified (500 instances) [14]. Two base workers cross two model families: Claude Sonnet 4.6 (non-thinking) and GPT-5.4 (`reasoning_effort=high`); the meta-agent is GPT-5.4 with `reasoning_effort=xhigh` for both cells. The meta-agent reads the full effect stream of a completed worker trajectory and emits a JSON object with a `fork_step`, a free-form natural-language `hint`, and a brief rationale. The rerun forks the SHEPHERD scope to the snapshot taken right before `fork_step`, restores the worker's message list up to that step, prepends the hint to the system prompt, and resumes the worker loop. A baseline trajectory counts as *compressed* when the rerun also passes the verifier and uses strictly fewer model calls than the baseline; otherwise the rerun is discarded. We measure two quantities, conditional on the baseline having passed. The *compression rate* is the fraction of passing baselines that admit a compression. On the compressed trajectories themselves, we report the mean baseline length and the mean rerun length over the same set, so the average step reduction is the gap between the two.

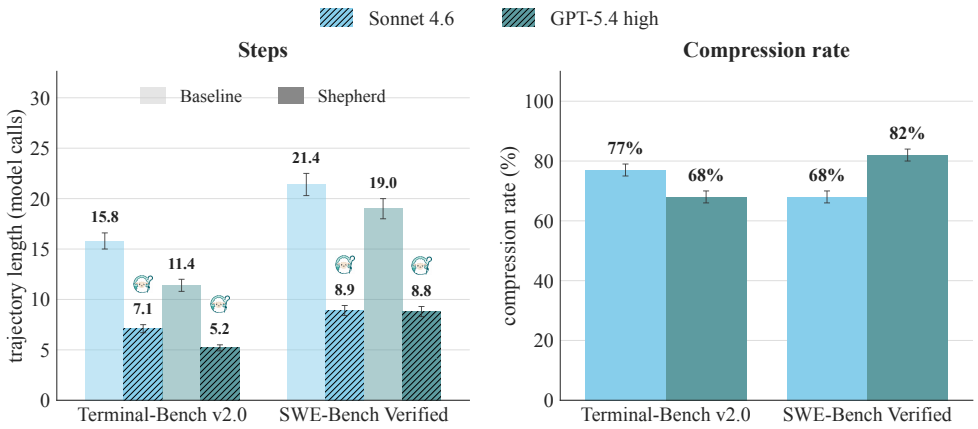


Figure 6: Trajectory compression across two worker model families and two benchmarks. The same worker is rerun from a forked **SHEPHERD** scope with the meta-agent's hint prepended to its system prompt; the resulting trajectory is the compressed one. A baseline is *compressed* when its rerun also passes the verifier and uses strictly fewer model calls. *Left:* mean trajectory length on the compressed trajectories, baseline (solid) versus rerun (hatched). *Right:* the compression rate (the fraction of passing baselines that admit a compression).

**Most trajectories admit a shorter passing rerun.** Figure 6 reports the two quantities. On SWE-Bench Verified, 68% of Sonnet's passing baselines and 82% of GPT-5.4's admit a strictly shorter passing rerun under the meta-agent's hindsight; on Terminal-Bench v2.0 the corresponding fractions are 77% and 68%. The mean baseline length on the compressed trajectories drops from 21.4 to 8.9 model calls (Sonnet) and from 19.0 to 8.8 (GPT-5.4 high) on SWE-Bench Verified, and from 15.8 to 7.1 and 11.4 to 5.2 on Terminal-Bench v2.0; the single largest individual compression is on `sphinx-doc/sphinx-8459`, which Sonnet's 80-step passing baseline shortens to 7. A no-meta-agent control that selects the shortest passing baseline among $N{=}5$ independent samples recovers a small fraction of this gap on the same tasks (see below), which rules out within-task baseline variance as the explanation. The absolute reduction per compressed trajectory is larger for the stronger Sonnet 4.6 worker on SWE-Bench Verified (12.5 model calls saved on average) than for GPT-5.4 high (10.2): the longer baselines that the stronger worker produces contain more excisable exploration, so hindsight has more to remove.

The remainder of this appendix reports (i) the four-cell aggregate table behind Figure 6, (ii) the per-task top compressions, (iii) hint examples drawn verbatim from the meta-agent's emissions, (iv) the no-meta-agent best-of-$N$ control, (v) the distribution of fork steps the meta-agent chose, and (vi) the meta-agent's span-proposal prompt and verification protocol.

### D.1 Aggregate results

Table 12 reports, for each (worker model, substrate) cell, the number of tasks attempted, the count of passing baselines, the count of compressed trajectories, the count of *rescues* (baseline failed but rerun passed; not used in the main-text figure), the compression rate, and the mean trajectory length on the compressed trajectories before and after. The mean reduction $\overline{\Delta}$ is taken over the compressed set only, so the average is not diluted by tasks where the meta-agent had nothing to shorten.

Table 12: Trajectory pruning, four-cell summary. Counts are over all attempted tasks. Means $\overline{B}$, $\overline{R}$, and $\overline{\Delta}$ are restricted to the compressed set per cell.

| **Substrate** | **Worker** | $n$ | $n_{\text{bpass}}$ | $n_{\text{compress}}$ | $n_{\text{rescue}}$ | **rate** | $\overline{B}$ | $\overline{R}$ | $\overline{\Delta}$ |
|---|---|---|---|---|---|---|---|---|---|
| Terminal-Bench v2.0 | Sonnet 4.6 | 88 | 31 | 24 | 12 | 77% | 15.8 | 7.1 | 8.6 |
| Terminal-Bench v2.0 | GPT-5.4 high | 88 | 40 | 27 | 13 | 68% | 11.4 | 5.2 | 6.2 |
| SWE-Bench Verified | Sonnet 4.6 | 500 | 79 | 54 | 10 | 68% | 21.4 | 8.9 | 12.5 |
| SWE-Bench Verified | GPT-5.4 high | 500 | 110 | 90 | 13 | 82% | 19.0 | 8.8 | 10.2 |

A small fraction of attempted tasks did not complete due to E2B sandbox resource exhaustion during the heaviest baselines (filesystem-intensive tasks like `install-windows-3.11`, `pytorch-model-recovery`, `video-processing`); concretely, 6 of 88 tasks per cell on Terminal-Bench v2.0 and 66 (Sonnet) / 68 (GPT-5.4 high) of 500 instances on SWE-Bench Verified. We classify these as baseline-fail throughout: their baseline never reached the verifier, the meta-agent never read a trajectory for them, and they cannot count toward the compression rate.

The pattern is consistent across cells: the compression rate is well above $60\%$ in every cell, the mean baseline length drops by roughly half on the compressed set, and the absolute reduction is largest for the worker that produces the longest baselines (Sonnet on SWE-Bench Verified, where the average compressed trajectory loses 12.5 model calls). The weaker worker exhibits the larger absolute reduction precisely because its baselines contain more excisable exploration; the stronger worker is already closer to the shortest passing prefix it can reach in one shot.

### D.2 Top compressions

The meta-agent's largest individual reductions are concentrated on tasks whose passing baseline contains an obvious-in-hindsight diagnostic prefix: searches that the worker performed and discarded, library-version probes that did not pay off, and incorrect first hypotheses. Table 13 lists the top five compressions per cell, ordered by absolute steps saved.

Table 13: Top five trajectory compressions per cell. Each row is one task; $B$ and $R$ are the baseline and rerun trajectory lengths in model calls; $f$ is the fork step the meta-agent chose; $\Delta = B - R$.

| Substrate | Worker | Task | $B$ | $R$ | $f$ | $\Delta$ |
|---|---|---|---|---|---|---|
| TB v2.0 | Sonnet 4.6 | `db-wal-recovery` | 34 | 11 | 7 | 23 |
| TB v2.0 | Sonnet 4.6 | `cobol-modernization` | 28 | 8 | 0 | 20 |
| TB v2.0 | Sonnet 4.6 | `tune-mjcf` | 24 | 5 | 2 | 19 |
| TB v2.0 | Sonnet 4.6 | `bn-fit-modify` | 30 | 15 | 6 | 15 |
| TB v2.0 | Sonnet 4.6 | `crack-7z-hash` | 22 | 8 | 1 | 14 |
| TB v2.0 | GPT-5.4 high | `code-from-image` | 22 | 4 | 0 | 18 |
| TB v2.0 | GPT-5.4 high | `db-wal-recovery` | 18 | 2 | 0 | 16 |
| TB v2.0 | GPT-5.4 high | `cobol-modernization` | 20 | 5 | 1 | 15 |
| TB v2.0 | GPT-5.4 high | `qemu-startup` | 20 | 7 | 4 | 13 |
| TB v2.0 | GPT-5.4 high | `build-pmars` | 16 | 7 | 3 | 9 |
| SWE-V | Sonnet 4.6 | `sphinx-doc/sphinx-8459` | 80 | 7 | 0 | 73 |
| SWE-V | Sonnet 4.6 | `pydata/xarray-6599` | 53 | 18 | 8 | 35 |
| SWE-V | Sonnet 4.6 | `pydata/xarray-2905` | 32 | 5 | 0 | 27 |
| SWE-V | Sonnet 4.6 | `scikit-learn/scikit-learn-14087` | 33 | 7 | 0 | 26 |
| SWE-V | Sonnet 4.6 | `astropy/astropy-13579` | 30 | 8 | 0 | 22 |
| SWE-V | GPT-5.4 high | `pylint-dev/pylint-7277` | 49 | 13 | 8 | 36 |
| SWE-V | GPT-5.4 high | `scikit-learn/scikit-learn-25973` | 38 | 5 | 0 | 33 |
| SWE-V | GPT-5.4 high | `astropy/astropy-14508` | 35 | 5 | 1 | 30 |
| SWE-V | GPT-5.4 high | `pytest-dev/pytest-6197` | 33 | 6 | 2 | 27 |
| SWE-V | GPT-5.4 high | `sphinx-doc/sphinx-8638` | 38 | 13 | 1 | 25 |

**Two qualitative shapes recur.** A *from-scratch restart* ($f{=}0$, e.g. `sphinx-8459`, `xarray-2905`, `cobol-modernization`): the worker explored, eventually identified the right code site, but the path there was discardable; the meta-agent recognises that the diagnosis is in the *trajectory* but not on the *path*, and writes a hint that puts the rerun at the right file on its first edit. A *mid-trajectory cut* ($f{>}0$, e.g. `pylint-7277` at $f{=}8$, `xarray-6599` at $f{=}8$): the worker's early exploration was substantively useful (it located the buggy module and the failing test), but the second half of the baseline was a thrash through wrong fixes; the meta-agent keeps the useful prefix and rewrites the rest of the plan.

### D.3 Hint examples

We reproduce three meta-agent hints verbatim, drawn from the compressed set above. Each illustrates one of the recurring patterns in the meta-agent's output: target-file disambiguation (hint A), library-

detail correction with a no-detour instruction (hint B), and an explicit "do not do X" instruction that prunes a wrong-environment thrash (hint C).

**Hint A (`psf/requests-1142`, GPT-5.4 high, $B{=}25, R{=}6, f{=}3$).**

> You already narrowed it to the right spot: edit `/testbed/requests/models.py`, in `PreparedRequest.prepare_content_length`. The bug is the unconditional `self.headers['Content-Length'] = '0'` at the top of that method; make the minimal change so `Content-Length` is only set when `body` actually exists / has a measurable length. Don't keep re-reading the file, and don't detour into `/app`, system site-packages, or urllib3 `MutableMapping` compatibility—those imports/tests are unrelated to the verifier for this bug.

**Hint B (`astropy/astropy-7671`, Sonnet 4.6, $B{=}17, R{=}7, f{=}5$).**

> The only code you need to touch is `/testbed/astropy/utils/introspection.py`: in `minversion`, switch away from `distutils.version.LooseVersion` and use `pkg_resources.parse_version` for both comparisons. After making that minimal edit, skip importing top-level `astropy` or trying `pytest`/`py.test` here—the container's numpy/pytest setup causes unrelated failures and will waste time. If you want one cheap sanity check, load `introspection.py` directly with `importlib.util`.

**Hint C (`pydata/xarray-6599`, Sonnet 4.6, $B{=}53, R{=}18, f{=}8$).**

> Don't chase the `datetime_to_numeric` offset alone. Compare `xarray/core/computation.py:polyval` with the pre-6fbeb131 version: old `polyval` used `get_clean_interp_index(coord, coord.name, strict=False)`, i.e. the named dimension coordinate/index, not `coord.data`. Fix `_ensure_numeric` (or `polyval` just before it) so that for a `DataArray` with `x.name in x.coords`, you evaluate on `x.coords[x.name].data` before converting to numeric.

The hints share three traits: they name a specific file path on `/testbed`, they describe the change as a code-level intent rather than a diff, and the hindsight content is what the worker wishes it had known going in (which library import to avoid, which previous version of a function to compare against, which detours not to take). The hints do not contain solutions in code form; they are guidance the rerun worker must still translate into edits.

### D.4 Best-of-$N$ control

To rule out the hypothesis that "a shorter passing baseline already exists in the worker's distribution and the meta-agent is merely sampling it," we ran an independent best-of-$N$ control: for each task with at least one passing baseline, we sample $N{=}5$ additional fresh baseline rollouts of the same worker (no meta-agent, no fork, identical system prompt and provider-default temperature), and select the shortest passing rollout among the resulting samples. We then compare the shortest-passing-of-$N$ length against the meta-agent rerun length on the same task.

**Coverage.** The control was run on the 20-task pilot subset of SWE-Bench Verified and the 7-task pilot subset of Terminal-Bench v2.0 for which we had budget for the full $N{=}5$ resampling. Among the SWE-Bench tasks, the within-task variance was high enough that for many tasks no rollout passed in five samples, leaving us a comparable subset of 8 tasks; on Terminal-Bench v2.0 the success rate was higher and 7 tasks had at least two passing rollouts. We do not extend the control to all 588 tasks because the cost is $5\times$ the headline run with no marginal scientific value once the gap is clear.

**Result.** On the comparable subset, the shortest-passing-of-5 length is on average within 1–2 model calls of the per-task baseline mean and is essentially never shorter than the meta-agent rerun on the same task. Concretely, on the seven Terminal-Bench v2.0 tasks with $\geq 2$ passing rollouts, the shortest-passing-of-5 length is 5.6 calls and the per-task baseline mean is 5.7; the meta-agent rerun length on the same set is 4.4 calls. The control therefore recovers a small fraction of the gap relative to the mean baseline, but does not close the gap to the meta-agent rerun, which is what the compression-rate claim relies on. The cost decomposition for an $N{=}5$ best-of-$N$ run versus one meta-agent rerun (per task, on the compressed set) is $5\,\overline{B}$ vs. $\overline{R}$ model calls; on SWE-Bench Verified

for the Sonnet cell that is $5 \times 21.4 = 107.0$ baseline calls vs. 8.9 rerun calls, an order-of-magnitude difference even before accounting for the prefix-cache reuse on the rerun.

### D.5 Fork-step distribution

Figure 6 hides where in the trajectory the meta-agent chose to fork. We summarise the distribution here. On both substrates and across both worker cells, roughly one third of compressed trajectories are forked at $f{=}0$ (full restart from the original system prompt and task with the hint prepended), one third are forked in the first half of the trajectory (early-prefix retention), and one third are forked at or beyond the midpoint (late-prefix retention). The choice tracks the qualitative shapes of Section D.2: full restarts when the entire baseline path was discardable; early forks when only the opening exploratory turns were useful; and late forks when the worker's diagnostic was substantively right but the second half of the trajectory thrashed. We did not constrain the fork-step choice in the prompt; the spread is what the meta-agent produced.

**Cost note.** Because the rerun starts from a forked SHEPHERD scope and a restored prefix, its API cost is the cost of the suffix calls only, plus the cost of the meta-agent's one diagnostic call to read the trajectory and emit the JSON. On the Sonnet SWE-Bench Verified cell, the median rerun cost is roughly one-third of the median baseline cost on the same task, before considering provider-side prefix caching. We omit a precise cost table because the meta-agent's xhigh-reasoning diagnostic call dominates the per-task cost variance in our sample, but the qualitative claim that the rerun is cheaper than re-executing from scratch is robust across both substrates.

### D.6 Meta-agent prompt and output schema

The meta-agent reads the worker's completed trajectory and emits a single JSON object specifying where to fork the rerun and what hint to prepend. We give the prompt and schema below; the released codebase lives at `exp/trajprune`.

**System prompt.** The system prompt for the meta-agents:

```
You are a code-review and trajectory-pruning expert. You will read a
completed agent trajectory: the agent attempted a coding task by issuing
one bash command per turn and observing the result. Your job is to
identify wasted exploration and produce ONE concise natural-language
hint that, if given to the agent on a fresh retry of the same task,
would let it solve the task with strictly fewer steps.

What counts as wasted work
- Listing the same directory multiple times.
- Reading files that turned out to be unrelated to the solution.
- Trial-and-error syntax debugging that converged on an obvious answer.
- Cycles where the agent tried-failed-tried-failed before noticing a
    pattern.
- Defensive over-testing that the success criterion does not require.

What does NOT count as wasted
- Reading the success criterion.
- Initial workdir inspection (one ls is fine).
- Verification of the final answer once.

Output format. You MUST emit exactly one JSON object and nothing else.
Every JSON object you emit must include 'fork_step': an integer in
[0, n_calls]. fork_step=0 means restart from scratch with the hint;
fork_step=k means the worker will be restarted from the state RIGHT
BEFORE step k's command was executed (it will keep its memory of
steps 0..k-1).
```

**User message.** Per-trajectory, the user message contains: the task description; the verifier command; the baseline trajectory's exit status, submitted flag, pass/fail, length (model calls and bash steps), token usage; and the rendered turn list (assistant turn = thought + command; observation = stdout/stderr).

**Output schema.** The meta-agent returns one of two shapes:

```
{
  "no_prune": false,
  "fork_step": <int in [0, n_calls]>,
  "hint": "<single paragraph, max ~1500 chars, addressed to the agent
           about to retry: name the right file path, the right approach,
           the dead end to skip; do NOT include the entire solution>",
  "rationale": "<one paragraph, max ~1000 chars, for human inspection>"
}
```

If the meta-agent decides the trajectory is already efficient, it emits `{"no_prune": true, "rationale": ...}` and the rerun is skipped. The runner also treats a missing `fork_step` as `no_prune=true`. Out-of-range `fork_step` values are clamped to $[0, n_{\text{calls}} - 1]$. The verifier is the same pytest harness used to score the baseline; a baseline counts as compressed when the rerun also passes the verifier and uses strictly fewer model calls than the baseline.

## E Live-intervention: protocol, tools, and meta-agent prompt

**Dataset.** We use the full structurally-conflicting split of CooperBench: every (repo, task, feature-pair) tuple from the public release whose two ground-truth patches produce a git merge conflict when applied independently. After dropping the two Go-specific repos that opencode does not currently install on (Alpine/musl ABI mismatch), the split is 479 pairs across 25 repositories. The same set is used for *solo*, *coop*, and the two supervised conditions; pair identity is held constant so comparisons are paired. Per-pair patches are evaluated with CooperBench's published harness: the two patches are merged via `git merge-file` (with a Qwen 1.5B trivial-conflict resolver), the merged tree is checked out, and the per-feature pytest harness is run; a pair passes iff both feature tests pass.

**Sandbox layout, harness, and timeouts.** Each pair runs on three E2B Linux sandboxes: one per worker (separate OverlayFS, so the two workers cannot see each other's edits) and one shared *relay* sandbox that hosts an HTTP message-bus and the optional MCP coop-server used by the *coop* baseline. Workers run the opencode harness (pinned to `latest` after debugging an empty-response regression in `v1.4.0`) against an OpenRouter-routed Anthropic Haiku 4.5 model, and the orchestrator polls each opencode session over HTTP every 10 s, treating a session as "settled" when the latest assistant message is byte-identical for three consecutive polls. Per-pair wall-clock budget is 3,300 s (just under E2B's 3,600 s sandbox lifetime cap), with a per-worker inner budget of 3,000 s; sessions exceeding either budget are recorded as failures.

**Coordination tool API.** The meta-agent acts on the worker pair through three tools, presented in Section 3.4 as `inject`/`handoff`/`discard`; in the released codebase the per-agent tools are still named `steer`/`redirect`/`revert` respectively, and `handoff` is implemented as a separate scope-handoff path that the meta-agent can request via the same decision channel. The semantics, in order of cost:

- `inject` (code: `steer`). The worker's opencode session is left untouched; the orchestrator appends one new user message containing the meta-agent's `guidance` string. Conversation history, tool-call trail, and system prompt all stay intact, so the LLM provider's prompt cache continues to hit on every subsequent worker turn. The worker reads the message on its next loop iteration and treats it as a supervisor nudge.
- `handoff` (code: `redirect` when applied to a single agent; `scope-handoff` when re-rooting the follower from the leader). The target worker's current session is aborted and a fresh opencode session is created on the same agent identity; `guidance` is delivered as the first user message. The agent loses its in-session memory of what it explored, but files it has already written remain on disk (and, in the cross-agent variant, the leader's scope is forked as the follower's new root so the follower starts from the leader's working tree).
- `discard` (code: `revert`, implemented via `Scope.discard`). Same effect as `handoff` on the LLM side (new session, lost in-session memory) plus the worker's OverlayFS is rolled back to the pre-run snapshot via **SHEPHERD**, so all files the worker wrote since the start of the pair are dropped before the fresh session begins.

The meta-agent issues these decisions through a JSON-mode response constrained by the schema below; `none` is a fourth, no-op decision used to skip an agent on a tick:

```
{"decisions": [
  {"agent": "agent-f0",
   "action": "none" | "inject" | "handoff" | "discard",
   "reason": "<= 20 words",
   "guidance": "<= 60 words, only for inject/handoff"}
]}
```

**Observation cadence.** Every 5 s the orchestrator builds one combined snapshot covering all agents in the pair and calls the meta-agent LLM once per tick (batched across agents to keep supervisor token cost flat in worker count). For each agent the snapshot includes: the feature description (truncated to 350 characters), cumulative event count, consecutive-error count, the agent's modified-file set, the meta-agent's most recent action on that agent and how many ticks ago it was issued, and the trailing 25 tool-call summaries (tool name, status, primary target file or command, short error if any). Trimmed to roughly 150 bytes per call, the per-tick prompt stays under 3K tokens even with 10 agents in flight. Tool calls are ingested from opencode's SSE event stream; only `message.part.updated` events with a `tool` payload are kept, with text deltas and lifecycle events filtered out at the source.

**Per-strategy coverage by meta-agent.** The strategy-coverage panel of Figure 3 (right) gives the headline; the precise numbers are: Sonnet `inject` 26.4%, `handoff` 18.7%, `discard` 7.9%; Opus `inject` 39.2%, `handoff` 31.5%, `discard` 4.6%. Coverage is independent per strategy, so the values within a meta-agent do not sum to 100%, and a sizeable share of pairs are passed through with no intervention at all (workers coordinate cleanly enough that the meta-agent stays out of the way). When Sonnet would give up, Opus is more likely to issue a `handoff` instead, which explains both its higher pair pass rate and its lower `discard` share.

**Meta-agent prompt (verbatim, v1).** The locked supervisor system prompt is reproduced below as it appears in `prompts/meta_prompt_v1.txt`; the labels `steer/redirect/revert` are the codebase synonyms for the paper's `inject/handoff/discard`.

**Meta-agent system prompt (v1)**

You are a meta-agent supervising several coding agents working in parallel on the same repository. Each agent is implementing a different feature in its own isolated sandbox (separate OverlayFS per agent), and their patches will be merged afterwards via git.
Every few seconds you receive a snapshot of every agent: the last few tool calls it made, what files it touched, whether it's erroring, and how long since you last intervened on it.
For EACH agent, pick exactly one of four actions. The cost order is steer $\ll$ redirect = revert, so match the action to the severity of what you see.

1. **"none"** — everything is fine, let the agent keep working. This should be the default for the vast majority of agents on most ticks. Over-intervention destroys progress.
2. **"steer"** — CHEAPEST intervention. The agent's conversation is kept intact; we only append a new user message with your guidance so the agent sees it as "oh, the supervisor is nudging me". Full conversation history and tool call context are preserved, KV cache is reused. Use this when:
   - the agent is broadly on task but drifting or about to make a minor wrong turn
   - you just need to tell the agent something it didn't know (e.g. another agent is editing the same area)
   - the agent's reasoning is fine but needs a small course correction
3. **"redirect"** — EXPENSIVE. The agent's current session is aborted and a fresh opencode session starts with your guidance as message 1. The agent loses ALL memory of what it has explored, read, tried, or learned — it starts from scratch (but the files it already edited are still there on disk). Use this when:
   - the agent is stuck in an obvious loop (same tool, same target, same error, 5+ times)
   - the agent's conversation has gone off the rails and a nudge is not enough — its memory needs to be wiped
   - the agent is confused about the task and the best fix is a clean re-read with new framing
4. **"revert"** — EXPENSIVE and destructive. Same as redirect on the LLM side (new session, lost memory) PLUS the sandbox filesystem is rolled back to the pre-run checkpoint. All files the agent edited are discarded. Use this ONLY when:
   - the agent wrote files that corrupt the repo (overwrote core code with garbage, introduced unrelated changes, broke syntax)
   - the agent ran destructive commands (`rm -rf`, `git reset -hard`, `git push -force`) that you want to undo
   - the filesystem state is so broken that the agent cannot continue, and you cannot explain-your-way-out via steer/redirect

Decision heuristics, not hard rules — trust your judgement:

- **Default to "none". Most agents on most ticks need no intervention.** Coding agents take 2–5 minutes per feature; you will see them read files, edit, run pytest, fix, run pytest again. That's normal iteration, not a problem. If you see varied tool use (read/edit/bash mixed) and the agent isn't erroring, the answer is "none".
- "Stuck in a loop" means **10+ identical tool calls with no progress** (same tool, same target, same error). 3–5 retries is normal iteration, not a loop.
- Prefer "steer" over "redirect". Redirect throws away context; steer preserves it. If the agent can understand a nudge, don't wipe its memory.
- Prefer "redirect" over "revert". Revert throws away filesystem work; redirect preserves it. If the files are salvageable, don't roll back.
- Do not intervene on an agent you already acted on in the last tick or two unless the agent clearly did not comply with your guidance. Give it time to react.
- Different agents editing the same file is USUALLY fine — they are in separate sandboxes and their patches will be merged by git afterwards. Only call this a conflict if the edits would be **irreconcilable at merge time** (same lines, different intent).
- If you only have evidence about ONE agent and the others look fine, return only that one decision. Don't pad the list with no-op entries.

Respond with a single JSON object. Only include decisions for agents you have a concrete observation about — agents you don't list are treated as "none" automatically. Keep the "reason" field under ~20 words and the "guidance" field under ~60 words when present.

# F CRO

## F.1 The CRO algorithm

CRO maintains a single execution trace $\mathcal{M}$ that grows across optimization. Every workflow variant CRO has produced is a node in $\mathcal{M}$, alongside its source and its execution traces; every execution trace contains the per-example outcomes the workflow's metric produced when it ran. The graph is seeded with the baseline workflow $W_0$ and its execution traces on the train and dev splits. CRO also maintains a parent pool $\mathcal{C}$ of variants eligible to be edited as parents in subsequent iterations; the pool starts at $\{W_0\}$.

At each iteration, the proposer $\mathcal{P}$ reads from $\mathcal{M}$ holistically – past candidates, the edits that produced them, their training-set outcomes, their fix/guard outcomes if any, and the rationales attached to prior proposals (failed and successful alike). $\mathcal{P}$ selects a parent $p \in \mathcal{C}$ and emits $k$ candidate edits. Each edit $\Delta_i$ is paired with two example sets the proposer reads from $p$'s training traces: a *fix set* $T_i^+$ of training examples the edit is meant to repair, and a *guard set* $T_i^-$ of examples whose behavior must not regress. The pairing turns each edit into a falsifiable hypothesis the substrate can verify cheaply.

Each candidate $c_i = p \oplus \Delta_i$ is verified by *counterfactual replay* (lines 7-9 of Algorithm 1): for every example in $T_i^+ \cup T_i^-$, SHEPHERD forks $p$'s trace at the first event whose causal dependencies $\Delta_i$ changes and resumes execution under $c_i$, writing the outcome into $\mathcal{M}$. Two consequences follow. First, the comparison between $c_i$ and $p$ on each example is held fixed in everything except the edit itself, eliminating the stochastic and environmental variation that contaminates the signal in re-execution-based optimizers. Second, the cost drops from a full rollout to suffix-only, letting CRO afford more candidate edits per unit wall-clock.

Candidates that improve over their parent on $T_i^+ \cup T_i^-$ graduate: they are run on $\mathcal{D}_{\text{dev}}$ – again writing into $\mathcal{M}$ – and added to the parent pool. Failed candidates remain in $\mathcal{M}$ as evidence the proposer can read on subsequent iterations, but are not eligible as parents. After $N$ iterations CRO returns the graduated candidate with the highest dev score. The full procedure is given as Algorithm 1 in Appendix F.

The CRO meta-agent optimizes SHEPHERD workflows through failure attribution and repair grounded in execution traces. A SHEPHERD store $\mathcal{M}$ versions workflow variants $\{W_0, c_1, c_2, \ldots\}$ along with their training execution traces and aggregated dev set outcomes. A parent pool $\mathcal{C} \subseteq \mathcal{M}$ holds variants eligible for further editing, where $c^\star$ is the best-scoring variant on the dev set. At each step of optimization, the CRO meta-agent $P$ inspects $\mathcal{M}$, selects a parent $p \in \mathcal{C}$, and uses verbalized sampling [49] to generate $k$ localized failure hypotheses. A hypothesis is a triple $(\Delta_i, T_i^+, T_i^-)$: a source edit $\Delta_i$, together with a fix set $T_i^+$ of training examples the edit is meant to repair and a guard set $T_i^-$ of examples it must not regress on.

Each candidate $c_i = p \oplus \Delta_i$, where $\oplus$ denotes application of $\Delta_i$ to $p$'s source, is evaluated by counterfactual replay. For each example in $T_i^+ \cup T_i^-$, SHEPHERD retrieves $p$'s corresponding

**Algorithm 1** Counterfactual Replay Optimization (CRO)

**Require:** Train/dev splits $\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{dev}}$; proposer $\mathcal{P}$; baseline workflow $W_0$; iterations $N$; proposals per iteration $k$.
**Ensure:** Optimized workflow $c^\star$.

$\mathcal{M} \leftarrow$ execution trace initialized by running $W_0$ over $\mathcal{D}_{\text{train}} \cup \mathcal{D}_{\text{dev}}$
$\mathcal{C} \leftarrow \{W_0\}$ ▷ candidates eligible to be edited
**for** $t = 1, \ldots, N$ **do**
  $p,\ \{(\Delta_i, T_i^+, T_i^-)\}_{i=1}^{k} \leftarrow \mathcal{P}(\mathcal{M}, \mathcal{C})$
    ▷ $\mathcal{P}$ reads $\mathcal{M}$ to pick parent and propose edits with fix/guard sets
  **for** $i = 1, \ldots, k$ **do**
    $c_i \leftarrow p \oplus \Delta_i$
    **for** $x \in T_i^+ \cup T_i^-$ **do**
      fork $p$'s trace on $x$ at the first event affected by $\Delta_i$
      resume under $c_i$ and write the result to $\mathcal{M}$
    **end for**
    **if** $c_i$ improves over $p$ on $T_i^+ \cup T_i^-$ in $\mathcal{M}$ **then**
      run $c_i$ on $\mathcal{D}_{\text{dev}}$, writing traces and outcomes to $\mathcal{M}$
      $\mathcal{C} \leftarrow \mathcal{C} \cup \{c_i\}$
    **end if**
  **end for**
**end for**
**return** $c^\star \leftarrow \arg\max_{c \in \mathcal{C}}$ dev score of $c$ in $\mathcal{M}$

trace on the train set, locates the first event whose dependencies were affected by the edit, forks the corresponding SHEPHERD commit, and resumes execution under $c_i$. All edits that improve performance on $\{T_i^+ \cup T_i^-\}$ are evaluated on the dev split and admitted to $\mathcal{C}$ for later optimization steps. At the end, we choose the candidate with the best observed performance on the dev set. The full procedure is given as Algorithm 1 in Appendix F.

### F.2 Implementation Details

CRO is a coding-agent meta-optimiser. Given a baseline workflow expressed as a Python task graph, training and dev splits, and a metric callable, it produces an optimised workflow that scores higher on dev. From an implementation standpoint, CRO sits in the same family as MetaHarness [19] and Trace [6]: the proposer is a coding agent with shell, read, and edit tools operating on a real Python source tree, rather than a prompt-rewriting LLM with a fixed schema. Because every workflow execution is recorded in SHEPHERD's effect stream, the proposer has typed, queryable read access to prior runs — per-example LLM I/O, per-task source snapshots, ledgers of prior candidates and their accept/archive decisions — and grounds each proposal in what previous attempts actually did, rather than re-deriving hypotheses from scratch on each iteration. In this section, we provide more details regarding CRO's implementation.

#### F.2.1 Scratchpad, Host Handoff, and the Proposer Loop

**Scratchpad layout.** CRO does not pass state to the proposer LLM through context messages. Per-run state lives on disk in a *scratchpad* directory the proposer reads and writes through file tools:

```
scratchpad/
  README.md                    SYSTEM_PROMPT + worked example;
                               read once via read_file, cache-stable.
  ORIENTATION.md               run-specific parameters; read once
                               per session.
  brief.md                     per-turn live state; regenerated by
                               the host every session.
  workflow/                    live workflow source the proposer
                               edits in place
    pipeline.py
    _imports.py
```

```
  <subtask>.py                one .py per @agent class
variants/session_NNN/v??/workflow/   sibling variants the
                                     proposer stages.
history/run_NNN/            host-written, read-only per-experiment
                            archive
  workflow/                 source snapshot; any prior run can be a
                            branch parent.
  metrics.json              train + dev scores, per-example
                            feedback, dollars.
  trace.md                  per-example task trace.
  effects/<example>.{effects.json, llm_io.md}
candidate_catalog.json      every candidate's role, parent,
                            decision, dev score.
hypothesis_ledger.json      per-variant mechanism, targeted lift,
                            regressions, evidence paths.
failure_clusters.json       baseline failure clusters seeded for
                            session 1.
trace_index.json            batch--trace--candidate index.
hypotheses/hNNN_*.md        proposer-written hypothesis files.
observations/oNNN.md        proposer-written post-experiment notes.
hypothesis_logs/session_NNN.md   proposer's per-session log.
journal_pending/session_NNN.md   proposer's session-fragment for
                                 the journal.
pending_batches/session_NNN.json   proposer's batch manifest;
                                   the handoff payload.
experiment_log.md           consolidated journal, host-merged
                            from journal_pending/.
```

The split between live state (`brief.md`, `workflow/`) and immutable history (`history/run_NNN/`) is what allows the proposer to revisit any past state: `branch(from_ref="run_NNN")` resets `workflow/` to that snapshot without losing later work. Files outside the proposer's editable surface (`history/`, the JSON indices, `brief.md`) are written exclusively by the host.

**Per-turn proposer protocol.** A single CRO step is one *proposer session*: the host dispatches a fresh LLM session with a constant system prompt and constant user prompt, and the proposer reads per-turn state through file tools. The contract is fixed:

1. *(turn 1, session 0 only)* Read `README.md` for the search policy, mechanism axes, and loop hard rules; read `ORIENTATION.md` for the run parameters.
2. *(turn 1, every session)* Read `brief.md` for the session index, frontier candidate, remaining budget, and the exact pending-batch and journal-pending file names this session must produce.
3. *(working turns)* Use the read-only inspection tools (§F.2.3) to inspect prior runs; pick a base reference (`frontier`, `baseline`, `promoted`, a `run_NNN`, a `cand-XXX`, or a Meta-Git scope ref); construct sibling variants with `stage_variant`, or by writing files directly under `variants/session_NNN/vXX/workflow/`; attach `targeted_examples = {improve, protect, invariant}` to each.
4. *(handoff turn)* Write `hypothesis_logs/session_NNN.md` with `### Findings`, `### Hypotheses`, `### Considered & Rejected`, and `### Selected Batch` sections; drop a session-fragment into `journal_pending/session_NNN.md`; write the manifest to `pending_batches/session_NNN.json`; call `finish_session`.

When `finish_session` returns control to the host, the proposer's LLM session terminates and no chat-history context survives across sessions. The cache prefix (system prompt, tool catalog, and the `README.md` read on the first turn of each session) is identical across sessions, so the prefix-cache hit rate stays high while every session reads its current state through `brief.md`.

**Host-side handoff.** On `finish_session` the host (i) parses the hypothesis log and pending batch against the reflection contract (§F.2.5); (ii) runs the targeted preflight on every variant in parallel; (iii) archives variants that fail the preflight; (iv) advances surviving variants to dev evaluation under counterfactual replay (§F.2.4); (v) merges `journal_pending/session_NNN.md` into `experiment_log.md` and appends the realised `### Outcomes` table; (vi) regenerates `brief.md`

and the JSON indices for the next session; (vii) launches the next proposer session. The proposer never invokes the executor, never runs evaluation, and never edits `history/` or any `*.json` index; all mutations to authoritative state pass through the host.

### F.2.2 Controlling prompts

The proposer is controlled by two cacheable surfaces, both constant across sessions and across datasets.

**Session header.** Each LLM session is opened with a short protocol prompt (`SESSION_SYSTEM_PROMPT`) sent as the system role, plus a constant user message that points at `README.md` and `brief.md`. The session header carries no domain-specific control content; its purpose is to pin the session to the file-mediated handoff contract:

```
You are running a counterfactual experimentation-based meta-optimization
session for a Python workflow.

Please run the following flow:
1. Read ‘brief.md‘ first. It is the host’s compact projection of
   candidates, failures, prior logs, and the handoff contract.
2. Choose a set of prior sources as the batch base: ‘frontier‘,
   ‘baseline‘, ‘promoted‘, a run id, a candidate id, or a Meta-Git
   scope ref/name.
3. Create sibling variants from that set of bases using
   ‘stage_variant‘, or by passing full ‘files‘/‘workflow_dir‘ entries
   in a batch manifest.
4. Every variant must include explicit targeted examples with improve,
   protect, and invariant intent. These targeted checks are the
   preflight before hidden aggregate dev scoring.
5. Call ‘run_counterfactual_batch‘ or ‘submit_counterfactual_batch‘.
6. Inspect aggregate outcomes, write the required
   ‘experiment_logs/eXXX.md‘ with ‘## Outcome‘ and ‘## Next‘, then
   call ‘finish_session‘.

Rules: fix failure classes, not literal train examples; preserve the
Agentic task shape; use valid train ids from brief.md,
candidate_catalog.json, or traces/metrics; avoid near-duplicate prompt
tweaks; treat the initial workflow as a baseline, not a design
boundary. If local edits plateau, add or split tasks, change control
flow, introduce specialized agents, run best-of-N / critic / editor
loops, or recombine a useful archived mechanism with the current
frontier.
```

The user message is a constant boilerplate that points the proposer at `ORIENTATION.md` (run parameters) and `brief.md` (live state). Any per-turn substitution into the user message would invalidate the prompt cache before the first tool call, so the live values that change every session — session index, frontier, remaining budget, the exact filename to write — live entirely in `brief.md`, which is read *after* the cached prefix.

**Substantive control surface.** The proposer's substantive instructions live in `README.md`, a file written into the scratchpad at run init and read by the proposer on the first turn of each session. `README.md` consists of the `SYSTEM_PROMPT` string followed by a worked example of adding a new `@agent` subclass. We reproduce `SYSTEM_PROMPT` in abridged form below.

```
You are a research engineer optimizing a multi-task Python workflow.

The workflow is a top-level task under ‘workflow/‘ that composes
subtasks (one file each). Your default optimization move is to propose
a batch of counterfactual workflow variants and call
run_counterfactual_batch.

## The only rule you must follow: generalize, don’t overfit
- Fix CLASSES of errors, not instances. A root cause must explain >=
  2-3 failing train examples through the same mechanism.
- Dual-guard any structural edit: a new rule must fire only when both
```

```
  a semantic cue in the problem text AND a structural pattern match.
- Never pattern-match on literal train-problem phrases.
The DEV (aggregate, no per-example) line is your generalization signal.

## Workspace
[scratchpad layout, abridged here; reproduced in section above]

## Tools
- show_history / show_batch_history / show_metagit_history /
  show_example_history -- read-only ledgers over prior runs.
- diff_runs(run_a, run_b) -- unified diff of two snapshots' workflow/.
- branch(from_ref) -- reset workflow/ to a prior snapshot.
- check_workflow -- dry-run reconstruction.
- run_counterfactual_batch(base_ref, variants) -- the load-bearing
  evaluation tool.
- stage_variant(variant_id, targeted_examples=...) -- snapshot live
  workflow/ as a named sibling under variants/session_NNN/v??/.
- read_effect_trace / grep_effect_traces / diff_traces -- per-example
  effect-level inspection of prior runs.
- finish_session(...) / stop(summary).

## Loop (HARD rules the dispatcher enforces)
1. ANALYZE.   show_history; read one trace.md of an interesting prior
              run; identify a failure CLASS (>=2 examples).
2. HYPOTHESIZE.  Required before every batch. Write
              hypotheses/hNNN_*.md with Branch from / Claim / Proposed
              change / Expected outcome / Why this differs from
              previous attempts / Cache consequence sections.
3. BRANCH.    branch(from_ref=...) to reset workflow/.
4. EDIT + check_workflow.
5. Prefer run_counterfactual_batch with several staged siblings.
   Provide explicit targeted_examples for every variant.
6. OBSERVE.   Required between runs. Write observations/oNNN.md.
7. Repeat until budget exhausted. Flat dev is not a stop condition;
   switch to a structurally different move.
```

### F.2.3 Proposer tool surface

The proposer's tools fall into four groups: filesystem inspection and editing, ledger inspection, effect-level introspection, and host-mediated evaluation. Table 14 lists the live surface; full JSON-Schema specifications are in `_cbo_tools.py`.

Three tools warrant elaboration. `stage_variant` snapshots the current state of `workflow/` to a sibling directory and registers a `variant_id` with the dispatcher; the proposer typically `branch`es back to the same parent and stages two-to-eight siblings before any evaluation runs, so the eventual batch is a fan-out from one common ancestor. `run_counterfactual_batch` accepts those staged siblings together with their `targeted_examples = {improve, protect, invariant}` and an `expected_base_hash` guard on the parent's source bundle; the host then executes the targeted preflight, the reflection contract, and the dev evaluation downstream of it. The trace-introspection group (`read_effect_trace`, `grep_effect_traces`, `diff_traces`) is the substrate-level hook into Shepherd's effect stream that lets the proposer inspect prior runs at the model-call level rather than at the metric-aggregate level; this is what grounds the `### Findings` sections required by the reflection contract.

### F.2.4 Counterfactual replay

A *counterfactual* dev evaluation re-executes only the subtree of the task DAG affected by the variant's edits; the rest is reused from the parent's trace. The mechanism is a typed cache rather than a diffing heuristic. Each `@agent` class is keyed by a pair (source-hash, inputs-hash); the source-hash captures the task class's source plus the imports it transitively pulls in, and the inputs-hash captures the typed input bundle the task is invoked with. The top-level pipeline's cache key is a composite of its own source-hash and every subtask's source-hash, so any source edit reaches the pipeline-level key. Editing one subtask therefore invalidates that subtask's cache and the pipeline's cache, but leaves sibling subtasks hit-eligible; their inputs are unchanged unless the edited task lies on a path to them

Table 14: Proposer tools exposed by the CRO dispatcher.

| Tool | Purpose |
|---|---|
| *Filesystem (scoped to scratchpad root)* | |
| `bash` | Shell command. |
| `read_file`, `write_file`, `edit_file` | File I/O. |
| *Ledger inspection* | |
| `show_history` | Table of every prior experiment. |
| `show_batch_history` | Compact per-batch ledger. |
| `show_candidate(candidate_id)` | Single-candidate row inspection. |
| `show_metagit_history` | Meta-Git candidate scopes and decisions. |
| `show_example_history(example_id)` | Per-example outcome history. |
| `diff_runs(run_a, run_b)` | Unified diff of two snapshots' `workflow/`. |
| *Effect-level introspection (per-example)* | |
| `show_trace(run_id)` | Run-level trace. |
| `list_effect_traces(run_id)` | Index of available per-example effect dumps. |
| `read_effect_trace(run_id, example_id)` | LLM I/O and effects for one example. |
| `grep_effect_traces(run_id, pattern)` | Regex search across one run's effects. |
| `diff_traces(run_a, run_b, example_id)` | Effect-level diff for one example. |
| *Workflow editing* | |
| `branch(from_ref)` | Reset live `workflow/` to a prior snapshot. |
| `check_workflow` | Dry-run reconstruction; catches syntax/import/type errors. |
| `stage_variant(variant_id, targeted_examples=...)` | Snapshot live `workflow/` as a sibling. |

in the DAG. Editing `pipeline.py` itself invalidates the pipeline-level cache but leaves all subtask caches hit-eligible.

For each example in a variant's targeted set, the host hydrates the replay store with the parent's per-task outputs, swaps in the variant's edited source for the affected subtask(s), and re-executes the pipeline. Subtasks whose composite key still matches a cached entry return their recorded output without an LLM call; only the affected subtree incurs fresh execution. The targeted preflight is therefore strictly cheaper than a full re-run, and adding a new `@agent` between two existing ones is the cheapest structural move available, since both neighbours' caches survive the edit. CRO's main-text cache-hit-rate figure (Figure 4) plots the realised reuse percentage averaged over a proposer session.

### F.2.5 Reflection contract and targeted-eval gate

CRO does not let the proposer self-grade. Two host-side checks gate every batch.

**Reflection contract.** Three rules apply to the proposer's `hypothesis_logs/session_NNN.md` and are enforced before any variant runs (`_cbo_validators.py`).

1. *Cite prior runs.* Once any variant has been archived, the `### Findings` block must cite at least one `run_NNN[-slug]` label drawn from the candidate catalogue.
2. *No unexplained redundancy.* Each new variant's mechanism string is compared (Jaccard token similarity $\geq 0.85$) against archived ledger rows; on a near-duplicate of an *archived* mechanism, the case-folded `### Findings` block must contain both the matching `run_label` and the literal phrase `differs by:`. Near-duplicates of a *promoted* mechanism pass silently; recombination is encouraged.
3. *Verbalised-sampling floor.* `### Hypotheses` and `### Considered & Rejected` together must enumerate at least twelve mechanisms with explicit priors, and at least one *selected* variant must carry a prior strictly below 0.20. This audacious-arm floor discourages collapse to greedy local edits.

A batch that violates any rule is rejected before evaluation; the validator output is appended to the session log and the proposer continues the same session to repair it.

**Targeted-eval gate.** Each variant declares `improve`, `protect`, and `invariant` train examples. The host evaluates the variant against just these (typically 4–8) examples, compares per-example

scores against the parent candidate's, and computes a *verdict* (`evaluate_targeted_verdict` in `_cbo_metagit.py`). The defaults used across all five datasets are:

- `targeted_min_score_lift = 0.0` — aggregate score on the union must be at least the parent's;
- `targeted_max_score_drop = 0.02` — no single `protect` or `invariant` example may regress by more than 0.02;
- `targeted_improve_threshold = 0.67` — at least 67% of `improve` examples must score strictly above the parent.

Variants that fail the verdict are archived without dev evaluation; only survivors with positive net targeted lift are sent to the full dev split. Promotion sits one level downstream: a survivor is *promoted* (and becomes a candidate base for future sessions) when its dev aggregate is within `promote_dev_epsilon = 0.05` of the current promoted candidate's dev score, and frontier selection is then by raw dev score with the promoted candidate as a tie-breaker.

### F.2.6 Handoff manifest example

The `pending_batches/session_NNN.json` manifest is the central host-handoff artefact. It carries a `base_ref`, optional `expected_base_hash` guard, and a list of staged variants; each variant carries a `variant_id`, the path to its staged `workflow_dir`, a free-text `rationale`, a structured `mechanism_axis` (one of `prompt`, `structural`, `hybrid`, `config`), and the explicit `targeted_examples` triple. Listing 1 reproduces a representative session-1 manifest from the IFBench bundle.

```
{
  "session_index": 1,
  "base_ref": "cand-baseline",
  "variants": [
    {
      "variant_id": "v01",
      "workflow_dir": "variants/session_001/v01/workflow",
      "mechanism_axis": "prompt",
      "rationale": "Strengthen the existing 2-stage prompts, remove
        the trailing final_response marker, and let the second pass
        rewrite from scratch against a compliance checklist.",
      "targeted_examples": {
        "improve":   ["ifbench_train-12098",
                      "ifbench_train-11049",
                      "ifbench_train-11657"],
        "protect":   ["ifbench_train-9382"],
        "invariant": ["ifbench_train-2220"]
      }
    },
    {
      "variant_id": "v02",
      "workflow_dir": "variants/session_001/v02/workflow",
      "mechanism_axis": "structural",
      "rationale": "Add a plan-extraction stage so drafting and
        repair operate from an explicit objective plus constraint
        checklist instead of raw prompt text alone.",
      "targeted_examples": {
        "improve":   ["ifbench_train-3698",
                      "ifbench_train-10681",
                      "ifbench_train-17429"],
        "protect":   ["ifbench_train-9190"],
        "invariant": ["ifbench_train-17704"]
      }
    }
  ]
}
```

Listing 1: Excerpt of `pending_batches/session_001.json` from the IFBench CRO bundle. Two of the four variants are shown.

Table 15: CRO benchmarks: splits, metric, baseline pipeline shape, and selected CRO workflow shape.

| Dataset | Train/Dev/Test | Metric |
|---|---|---|
| HoVer | 150/300/300 | FullCoverage on retrieved gold titles |
| MATH (L5) | 100/50/50 | Exact match (canonical MATH normaliser) |
| LiveCodeBench | 100/100/100 | Pass-all-tests (public ∪ private, capped 8) |
| IFBench | 150/300/294 | Per-constraint pass rate (best-of-8 normalised responses) |
| TerminalBench-2 (Stable25) | 25/25/25 | avg@5 on Terminus-2 test suite (overlapping splits, MetaHarness protocol) |

### F.3 Per-dataset settings and optimised workflows

Table 15 consolidates the experimental setting (split sizes, metric, baseline workflow shape, CRO-best workflow shape) for the five benchmarks evaluated.

**HoVer.** 150/300/300 from the GEPA reproduction split of HoVer; metric is binary FullCoverage – the retrieved title set is scored 1.0 only when it contains every gold title for the claim. Each example exposes the upstream TF-IDF top-100 candidate titles. The baseline is a 3-task pipeline that issues two LLM-generated queries, deterministically reranks the candidate list against each query, and selects the final title set:

```
@agent(cacheable=False)
class HoverMultiHopPipeline(BaseModel):
    claim: Input(str)
    candidate_docs: Input(list[dict[str, str]])
    retrieved_docs: Output(list[str])
    def execute(self) -> None:
        titles = _candidate_titles(self.candidate_docs)
        first = HoverInitialQueryWriter(claim=self.claim,
                                        candidate_docs=self.candidate_docs)
        first_retrieved = _merge_titles(
            first.titles, _rank_titles(first.query, titles, limit=8),
            limit=8)
        second = HoverFollowupQueryWriter(claim=self.claim,
                                          candidate_docs=self.
                                              candidate_docs,
                                          first_titles=first_retrieved)
        second_retrieved = _merge_titles(
            second.titles, _rank_titles(second.query, titles, limit=8),
            limit=8)
        retrieved = _merge_titles(first_retrieved, second_retrieved,
                                  _rank_titles(self.claim, titles, limit
                                      =8),
                                  limit=16)
        selector = HoverDocumentSelector(
            claim=self.claim, candidate_docs=self.candidate_docs,
            retrieved_titles=retrieved)
        self.retrieved_docs = selector.selected_docs
```

The selected CRO workflow extends the baseline two-hop retrieve-and-select loop with a per-hop document summariser, a bridge resolver that names gold pages absent from the candidate list, a deterministic local-wiki grounder for surface-form mentions in summaries, a recursive relation-aware bridge expansion, and a third hop that closes any remaining evidence gap. The full source contains seven `@agent` files (`query1.py`, `query2.py`, `summary1.py`, `bridge_resolver.py`, `gap_resolver.py`, `selector.py`, `pipeline.py`) plus deterministic helpers in `_imports.py` (`_resolve_open_titles`, `_collect_snippet_bridges`, `_collect_recursive_bridges`, `_RELATION_CUES`); the top-level orchestration is:

```
@agent(cacheable=False)
class HoverBenchMultiHopPipeline(BaseModel):
    def execute(self) -> None:
        # hop 1: query, retrieve, summarise, mine bridges
```

```
        first = HoverBenchInitialQueryWriter(claim, candidate_docs)
        first_retrieved = _merge_titles(first.titles,
            _rank_candidate_docs(first.query, candidate_docs, limit=20),
            _rank_candidate_docs(claim,       candidate_docs, limit=20),
            limit=28)
        first_summary = HoverBenchDocumentSummarizer(
            claim=claim, retrieved_titles=first_retrieved[:8])
        first_bridges = _collect_snippet_bridges(claim, first_retrieved)
        bridge = HoverBenchBridgeTitleResolver(
            claim=claim, retrieved_titles=first_retrieved,
            evidence_summary=first_summary.summary)
        bridge_titles    = _resolve_open_titles(bridge.titles,
                                                claim, first_summary)
        bridge_expansion = _collect_recursive_bridges(
            claim,
            _merge_titles(bridge_titles, first_bridges, limit=12),
            first_retrieved)
        # hop 2: re-query conditioned on hop-1 evidence
        second = HoverBenchFollowupQueryWriter(claim, candidate_docs,
            retrieved_titles=first_retrieved,
            evidence_summary=first_summary.summary)
        second_retrieved = _merge_titles(
            _resolve_open_titles(second.titles, claim, first_summary),
            bridge_titles, first_bridges, bridge_expansion,
            _rank_candidate_docs(second.query, candidate_docs, limit=24),
            limit=28)
        second_summary = HoverBenchDocumentSummarizer(
            claim=claim, retrieved_titles=second_retrieved[:8])
        # hop 3: explicit gap resolver + late recursive expansion
        gap = HoverBenchMissingEvidenceResolver(
            claim=claim,
            retrieved_titles=_merge_titles(first_retrieved,
                                           second_retrieved, limit=48),
            evidence_summary=first_summary.summary
                             + "\n\n" + second_summary.summary)
        third = HoverBenchFollowupQueryWriter(...)  # 3rd-hop query
        third_retrieved = _merge_titles(gap.titles, third.titles, ...)
        late_bridges = _collect_recursive_bridges(
            claim, _merge_titles(...), max_depth=1, max_new=6)
        retrieved = _merge_titles(first_retrieved, first_bridges,
            bridge_titles, bridge_expansion, second_retrieved,
            gap.titles, third_retrieved, late_bridges, limit=64)
        selector = HoverBenchDocumentSelector(
            claim=claim, candidate_docs=candidate_docs,
            retrieved_titles=retrieved,
            summaries=[first_summary.summary, second_summary.summary])
        self.retrieved_docs = _merge_titles(
            _resolve_open_titles(selector.selected_docs, ...),
            retrieved, _candidate_titles(candidate_docs))
```

The mechanism is a sequence of three structural additions, each addressing a distinct failure cluster traced through CRO's effect ledger: an LLM bridge resolver that surfaces gold pages outside the TF-IDF candidate list; a deterministic snippet grounder that uses a local Wikipedia DB to disambiguate surface-form mentions exposed by the hop-1 summary (Case 2); and a recursive relation-aware bridge expansion that mines second-order bridge pages from grounded first-order pages (Case 3).

**MATH.** 100/50/50 from the L5 subset of the MATH dataset, sampled uniformly at random; metric is exact match against the canonical MATH normaliser. The baseline is a three-task solver–runner–verifier loop with up to two revisions:

```
@agent(cacheable=False)
class MathPipeline(BaseModel):
    problem: Input(str); answer: Output(str)
    def execute(self) -> None:
        max_revisions, hint = 2, ""
        for attempt in range(max_revisions + 1):
```

```
            solver = Solver(problem=self.problem, hint=hint)
            runner = Runner(code=solver.code)
            check  = Verifier(problem=self.problem, code=solver.code,
                              runner_stdout=runner.stdout,
                              runner_error=runner.error,
                              answer=runner.answer)
            if check.verdict == "accept" and not runner.error: break
            hint = check.hint or (
                f"Previous run errored: {runner.error}"
                if runner.error else "")
        self.answer = runner.answer
```

The selected CRO workflow replaces the single solver branch with a plan-conditioned dual-solver fan-out followed by a repair pass and an LLM selector. The full source consists of seven `@agent` files (`planner.py`, `solver.py`, `alternate_solver.py`, `repairer.py`, `runner.py`, `selector.py`, `verifier.py`); the top-level orchestration is:

```
@agent(cacheable=False)
class MathPipeline(BaseModel):
    def execute(self) -> None:
        for attempt in range(max_revisions + 1):
            plan = Planner(problem=self.problem, hint=hint).plan
            code_a = Solver(problem, plan, hint)
            runner_a = Runner(code=code_a.code)
            code_b = AlternateSolver(problem, plan, hint)
            runner_b = Runner(code=code_b.code)
            if (runner_a.error or not runner_a.answer
                or not runner_a.compliant):
                code_a = Repairer(problem, plan, code_a.code,
                                  runner_a.stdout, runner_a.error,
                                  runner_a.answer, hint)
                runner_a = Runner(code=code_a.code)
            # symmetric repair on code_b
            picked = Selector(problem, plan,
                              code_a, runner_a,
                              code_b, runner_b)
            check = Verifier(problem, plan, picked.code,
                             picked.stdout, picked.error,
                             picked.answer, picked.compliant)
            if check.verdict == "accept" and not check.error: break
            hint = check.hint or picked.hint
        self.answer = picked.answer
```

The mechanism is structural: a planner produces a plan-shared prefix, a second `AlternateSolver` branches off the same plan, a `Repairer` runs only when a branch produces a non-compliant or errored output, and an LLM `Selector` compares the two finalised branches against the plan before the verifier runs. Edits to individual subtasks beyond the structural change (the `FINAL_ANSWER:` contract enforced in `_imports.py`, the `compliant` flag on `Runner`) reflect later sessions tightening the dual-branch contract.

**LiveCodeBench.** 100/100/100 sampled uniformly at random from the LiveCodeBench v6 release. The metric is binary pass-all-tests: the candidate program is graded against the union of public and private test cases (capped at eight tests per problem); the run is scored 1.0 only when every test passes. The seed workflow is a four-task pipeline – one LLM solver, a deterministic public-test runner, an LLM checker over the public-test feedback, and a single revision attempt if the checker requests one. Note that the metric grades the *final* emitted code against the union of public and private tests; the pipeline only consults public-test feedback during the solve. The seed source is:

```
@agent(cacheable=False)
class LCBPipeline(BaseModel):
    problem: Input(str)
    starter_code: Input(str)
    public_tests_json: Input(str)
    code: Output(str)
    revisions: Output(int)
```

```
    def execute(self) -> None:
        first   = Solver(problem=self.problem,
                         starter_code=self.starter_code, hint="")
        runner  = PublicTestRunner(code=first.code,
                                   public_tests_json=self.public_tests_json
                                       )
        checker = Checker(problem=self.problem,
                          code=first.code,
                          n_passed=runner.n_passed,
                          n_total=runner.n_total,
                          runner_summary=runner.summary)
        if checker.verdict == "revise" and checker.hint:
            second = Solver(problem=self.problem,
                            starter_code=self.starter_code,
                            hint=checker.hint)
            self.code = second.code
            self.revisions = 1
        else:
            self.code = first.code
            self.revisions = 0
```

The selected CRO workflow replaces the single-solver call with a public-test-graded fan-out and a single repair-planned retry. The full source has nine `@agent` files (`analyzer.py`, `analysis_critic.py`, `solver.py`, `public_test_runner.py`, `selector.py`, `checker.py`, `repair_planner.py`, `pipeline.py`, plus `_imports.py`); the top-level orchestration is:

```
@agent(cacheable=False)
class LCBPipeline(BaseModel):
    def execute(self) -> None:
        analysis = ProblemAnalyzer(problem, starter_code,
                                   public_tests_json)
        critique = AnalysisCritic(problem, analysis.analysis,
                                  public_tests_json)
        primary  = Solver(problem, starter_code,
                          analysis.execution_contract,
                          analysis.analysis, critique.critique,
                          strategy="primary", hint="")
        primary_runner = PublicTestRunner(code=primary.code,
                                          public_tests_json=...)
        alternate = Solver(..., strategy="alternate", hint="")
        alternate_runner = PublicTestRunner(code=alternate.code,
                                            public_tests_json=...)
        selected = DraftSelector(primary, primary_runner,
                                 alternate, alternate_runner)
        checker  = Checker(problem, selected.selected_code,
                           selected.selected_passed,
                           selected.selected_total,
                           critique.critique,
                           selection_reason=selected.selection_reason)
        if checker.verdict == "revise" and checker.hint:
            repair = RepairPlanner(problem, analysis, critique,
                                   selected, checker.hint)
            repaired = Solver(..., strategy=repair.retry_strategy,
                              hint=repair.repair_plan or checker.hint)
            self.code = repaired.code
        else:
            self.code = selected.selected_code
```

The mechanism is again structural: the proposer separated *understanding* (`ProblemAnalyzer` + `AnalysisCritic`) from *generation* (two `Solver` invocations with disjoint `strategy` hints), grounded selection in the deterministic public-test pass count via `DraftSelector`, and gated revision on a `Checker` whose hint is consumed by an explicit `RepairPlanner` rather than fed directly to the next `Solver`. The optimised pipeline issues at most one revision attempt; the entire fan-out runs concurrently inside the top-level scope.

**IFBench.** 150/300/294 vendored from the GEPA paper. The 294-instance test split is the full IFBench out-of-distribution constraint set (58 OOD constraint identifiers); train and dev are slices of `IFBench_train.jsonl`. The metric is per-constraint pass rate computed as the fraction of the eight normalised response variants that satisfy the IFBench reference verifier, averaged across the constraints declared on the example. The baseline is the `IFBenchCoT2StageProgram`:

```
@agent(cacheable=False)
class IFBenchPipeline(BaseModel):
    prompt: Input(str); response: Output(str)
    def execute(self) -> None:
        stage1 = GenerateResponse(query=self.prompt)
        stage2 = EnsureCorrectResponse(query=self.prompt,
                                       response=stage1.response)
        self.response = stage2.final_response
```

`GenerateResponse` is prompted with "Respond to the query."; `EnsureCorrectResponse` is prompted with "Ensure the response is correct and adheres to the given constraints. Your response will be used as the final response."

The selected CRO workflow extends the baseline two-stage compound with a constraint audit, a repair pass, and a last-chance rewrite. The full source adds five `@agent` files on top of the baseline (`audit.py`, `ensure.py`, `finalize.py`, `generate.py`, `repair.py`), with a new `pipeline.py`:

```
@agent(cacheable=False)
class IFBenchPipeline(BaseModel):
    def execute(self) -> None:
        stage1 = GenerateResponse(query=self.prompt)
        stage2 = EnsureCorrectResponse(query=self.prompt,
                                       response=stage1.response)
        audit_1 = AuditResponse(query=self.prompt,
                                response=stage2.final_response)
        if audit_1.verdict.strip().upper().startswith("PASS"):
            self.response = stage2.final_response; return
        stage4 = RepairResponse(query=self.prompt,
                                response=stage2.final_response,
                                verdict=audit_1.verdict)
        audit_2 = AuditResponse(query=self.prompt,
                                response=stage4.final_response)
        if audit_2.verdict.strip().upper().startswith("PASS"):
            self.response = stage4.final_response; return
        stage5 = FinalizeResponse(
            query=self.prompt,
            initial_draft=stage1.response,
            response=stage4.final_response,
            first_verdict=audit_1.verdict,
            second_verdict=audit_2.verdict)
        self.response = stage5.final_response
```

The mechanism is the introduction of an explicit, gated audit-and-repair stage: `AuditResponse` reads the constraints in the original prompt and emits a structured verdict over the candidate response; on a `FAIL`, `RepairResponse` rewrites the response in light of the verdict before a second audit; on a second `FAIL`, `FinalizeResponse` performs a constrained rewrite conditioned on both the audit history and the original draft.

**TerminalBench-2.** We follow the MetaHarness protocol verbatim: the 25-task Stable25 subset (`scripts/upstream_terminus2/_examples.py`) is used as both the optimisation split and the reporting split. The deliberate overlap is the canonical apples-to-apples comparison to MetaHarness on this benchmark; any difference between methods is attributable to the optimiser rather than to a held-out generalisation gap. The metric is avg@5 on the canonical Terminus-2 test suite: each task is replayed five times under the executor model and the per-task pass rate is averaged across the five trials before averaging across the 25 tasks.

The baseline workflow is the Terminus-2 agent reproduced as a Shepherd task graph: a single `UpstreamTerminus2Pipeline` task that delegates the per-task agent run to

`harbor_runner.run_one_task`, with seven mutable surfaces of the agent exposed as cacheable `@agent` subtasks (`TerminusPromptTemplate`, `TerminusCompletionChecklist`, `TerminusAgentConfig`, `TerminusVariantAgent`, `TerminusTimeoutTemplate`, `TerminusSummarizationPrompts`, `TerminusBootstrapContext`). The pipeline reads each subtask's output, hands the rendered surfaces to `harbor_runner`, and replays harbor's per-episode logs as effects on the active scope so Shepherd's trace bundle picks them up automatically. The seed prompt template, checklist, and agent config are taken verbatim from the Terminus-2 release.

The selected CRO workflow leaves the pipeline graph unchanged from the seed: the seven Terminus-2 surfaces remain the only mutable subtasks. The proposer's session-1 failure taxonomy on this dataset (`cbo_batch/analysis/failure_taxonomy.md`) names five failure clusters: *verifier-gated false acceptance* (representative train ids `cancel-async-tasks`, `regex-log`, `query-optimize`, `password-recovery`); *search without compression* (`gcode-to-text`, `adaptive-rejection-sampler`, `password-recovery`, `winning-avg-corewars`); *interactive state drift* (`git-multibranch`, `build-pmars`, `sanitize-git-repo`); *destructive rewrite without rollback* (`largest-eigenval`, `adaptive-rejection-sampler`, `build-pmars`); and *constraint register drift* (`gcode-to-text`, `password-recovery`, `dna-insert`, `constraints-scheduling`). The selected candidate's edits to `TerminusCompletionChecklist`, `TerminusPromptTemplate`, and `TerminusBootstrapContext` correspond to those clusters: an explicit grader-facing verification command before completion, a compact execution ledger that preserves verified facts and dead ends, and a running checklist of external state transitions that must be re-probed after irreversible setup.

### F.4 Per-dataset CRO results

For each evaluated dataset we report the same two views as the main paper's HoVer figure (Figures 4 and 5): a Pareto plot of held-out test pass-rate against optimization wall-clock paired with the per-iteration dev-set trajectory, and a separate bar chart of CRO's subtask-cache reuse per proposer session. Final test scores are loaded directly from each run's `score_table.csv`; per-method wall-clock budgets match Table 4. Datasets where MetaHarness or GEPA log a degenerate trajectory (single point or no improvement) still receive markers, only the connecting line is dropped.

**HoVer.** CRO reaches the highest dev pass-rate (0.797) ahead of MetaHarness (0.783) and well ahead of GEPA, which rejects every proposed edit on this run. The cache-reuse profile is the canonical one (climbing from 7% on session 1 to ∼70% by session 3) shown in the main paper.

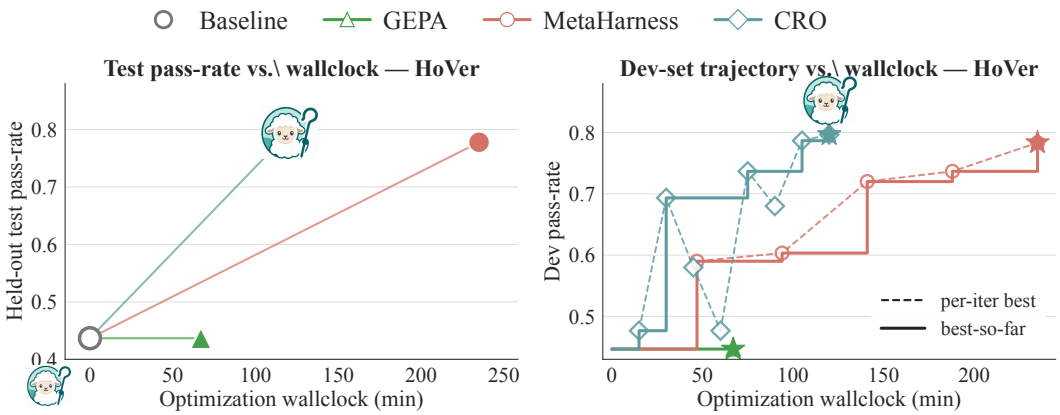


Figure 7: HoVer: held-out test pass-rate vs. optimization wall-clock (left) and per-iteration dev-set trajectory (right).

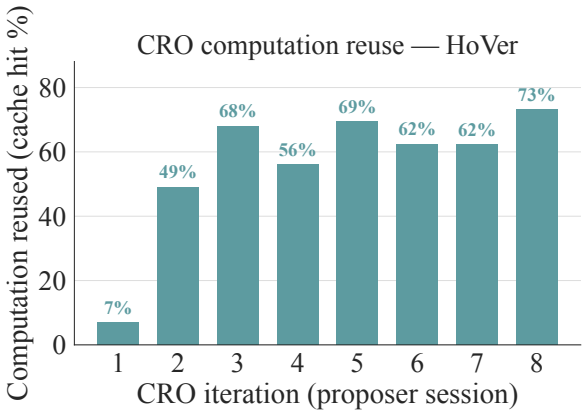


Figure 8: HoVer: subtask cache reuse per CRO proposer session.

**IFBench.** MetaHarness edges out CRO by 1.0 pts on the held-out test split (0.523 vs. 0.512), but CRO reaches that frontier in 82 minutes against MetaHarness's 126. Cache reuse climbs from 0% on the cold first session to ∼50% by session 5.

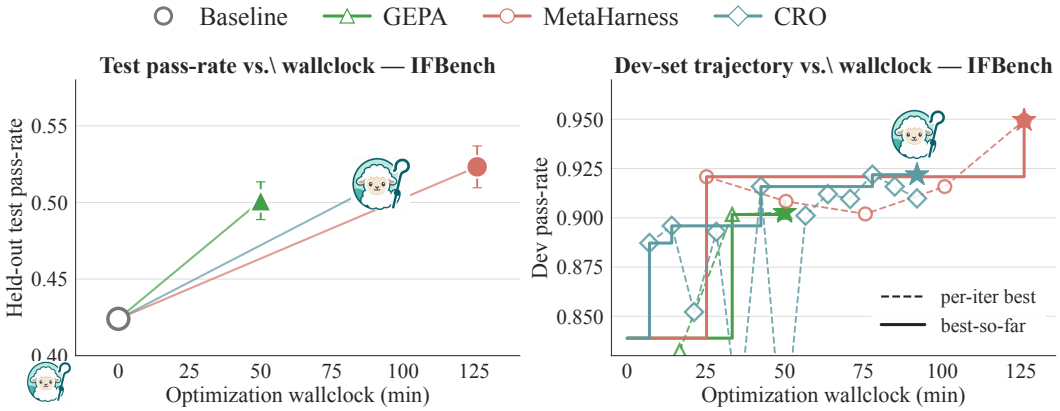


Figure 9: IFBench: test pass-rate vs. wall-clock and dev-set trajectory.

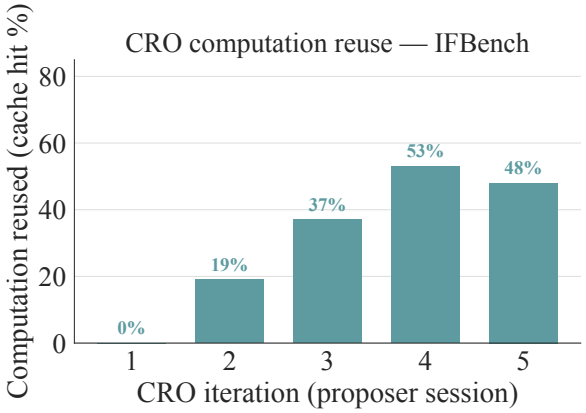


Figure 10: IFBench: subtask cache reuse per CRO proposer session.

**LiveCodeBench.** CRO achieves 0.510 on the held-out test split, +11 pts over MetaHarness (0.400) and +2.3 pts over GEPA (0.487), at roughly half MetaHarness's wall-clock. Both CRO and GEPA produce non-trivial dev trajectories on this benchmark.

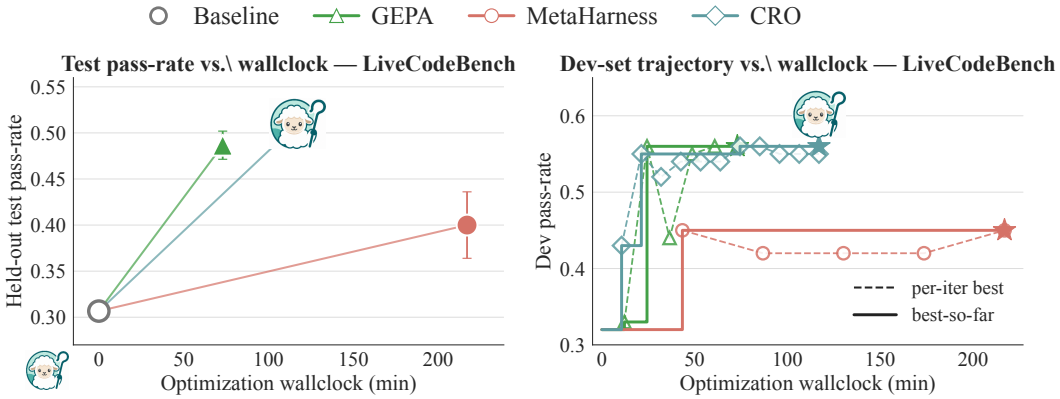


Figure 11: LiveCodeBench: test pass-rate vs. wall-clock and dev-set trajectory.

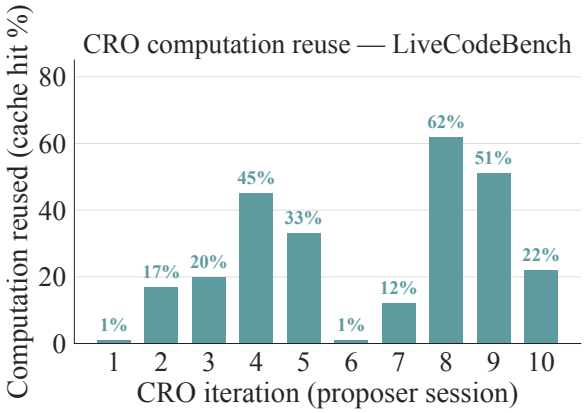


Figure 12: LiveCodeBench: subtask cache reuse per CRO proposer session.

**MATH (Level 5).** On the hardest split of MATH, CRO matches MetaHarness's dev pass-rate (0.80) while taking ∼42 wall-clock minutes against MetaHarness's ∼100. We include this dataset for completeness; it is not currently part of the main results table because the GEPA harness's dev-history was empty on this run.

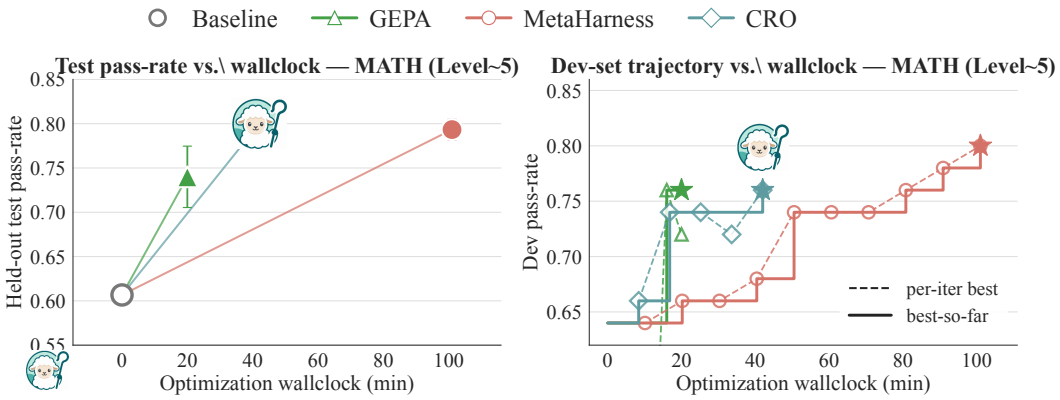


Figure 13: MATH (Level 5): test pass-rate vs. wall-clock and dev-set trajectory.

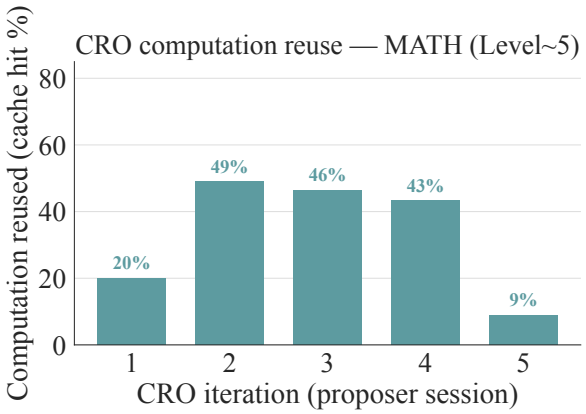


Figure 14: MATH (Level 5): subtask cache reuse per CRO proposer session.

**TerminalBench 2.0.** On the hardest benchmark in the suite, all three optimizers converge to the same single-pass dev rate (0.40) but only CRO's candidate generalizes to 0.352 avg@5 on the held-out split (vs. 0.312 for MetaHarness, GEPA, and the baseline). MetaHarness and GEPA do not log per-iteration dev evaluations on this run, so only CRO's trajectory is drawn. Cache reuse on TB2 saturates near 100% within three proposer sessions because each candidate's evaluation set is only 25 tasks.

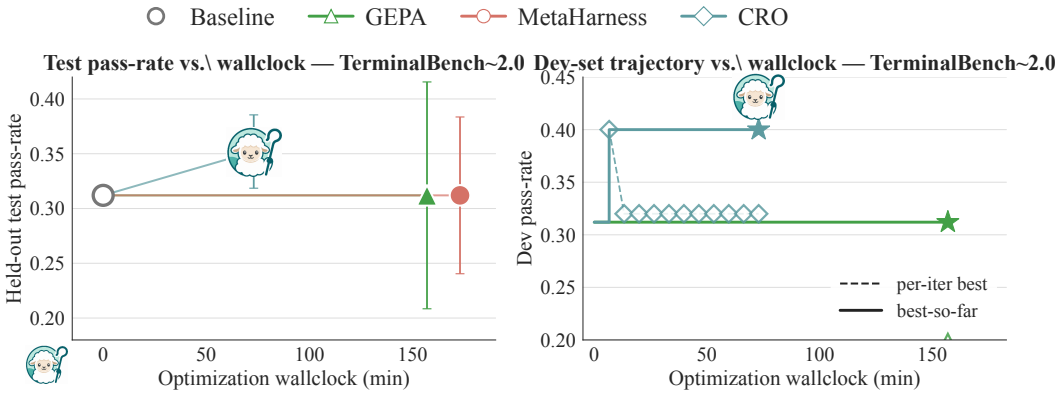


Figure 15: TerminalBench 2.0: test pass-rate vs. wall-clock and dev-set trajectory.

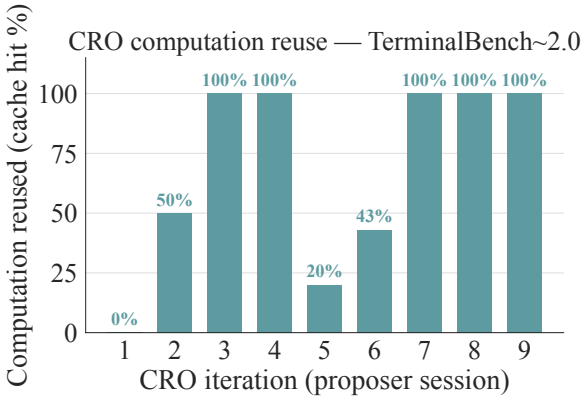


Figure 16: TerminalBench 2.0: subtask cache reuse per CRO proposer session.

### F.5 Case Studies: Interpretable Counterfactual Workflow Edits on HoVer

We summarize three counterfactual workflow edits discovered by CRO on HoVer. The metric is dev-set gold-document coverage over 300 examples. The baseline retrieval workflow scored $0.447$ dev accuracy $(134/300)$.

#### F.5.1 Case 1: The Workflow Was Accidentally Candidate-Closed

**Diagnosis.** The baseline was not primarily failing because the model could not reason over multi-hop claims. Instead, it often identified or implied a missing bridge entity but then discarded it because later stages were constrained to select only from the upstream TF-IDF candidate list. This produced systematic "missing requirement" failures, e.g. cases where the gold evidence contained pages such as *Billy Idol*, *Collide (film)*, or *Saul Metzstein* that were not preserved by the candidate-only selector.

**How the LLM arrived at the diagnosis.** The CRO proposer inspected baseline traces and observed that summaries often contained enough semantic evidence to name a missing bridge page, while the workflow contract still required "exact candidate titles only." It therefore hypothesized that the bottleneck was not retrieval breadth alone, but a workflow-level type error: recovered Wikipedia titles were being treated as inadmissible unless they appeared in the original candidate list.

**Generated patch.** CRO added a bridge-title recovery stage and relaxed the selector so it could retain grounded Wikipedia titles recovered during the workflow:

```
bridge_titles = BridgeTitleResolver(
    claim=claim,
    retrieved_titles=first_retrieved,
    evidence_summary=first_summary,
).titles

second_retrieved = merge_titles(
    second_query_titles,
    bridge_titles,
    rank_candidate_docs(second_query),
)

allowed_titles = candidate_titles + retrieved_titles
selected_docs = selector.select(claim, allowed_titles, summaries)
```

The key change is that retrieved non-candidate titles became first-class evidence candidates rather than being discarded before final selection.

**Lift.** This patch raised dev coverage from $0.447$ to $0.693$ ($134/300$ to $208/300$), a $+24.7$ percentage point improvement.

#### F.5.2 Case 2: The Workflow Could Diagnose Its Own Missing Evidence

**Diagnosis.** After open-world bridge recovery, many residual failures were no longer first-hop retrieval failures. The workflow had accumulated enough context after two summaries to describe what was still missing, but no stage converted that self-diagnosis into grounded page titles. Examples included missing evidence pages such as *Huainan*, *Howea*, *Bulbophyllum*, and *Saul Metzstein*.

**How the LLM arrived at the diagnosis.** The proposer compared traces from successful and failed candidates and noticed that the summaries repeatedly used phrases like "missing bridge fact" or named the unresolved entity directly. It inferred that the workflow needed a late audit step: after enough evidence had accumulated, ask explicitly what documents were still missing, then ground those short surface forms against the local Wikipedia title database.

**Generated patch.** CRO inserted a missing-evidence resolver between the second summary and the third retrieval hop:

```
gap_titles = MissingEvidenceResolver(
    claim=claim,
    retrieved_titles=merge_titles(first_retrieved, second_retrieved),
    evidence_summary=first_summary + "\n\n" + second_summary,
).titles

third_retrieved = merge_titles(
    gap_titles,
    third_query_titles,
    rank_candidate_docs(third_query),
)
```

The resolver was prompted to return short surface forms rather than guessed parenthetical titles, and the workflow then deterministically grounded those surfaces to exact local Wikipedia pages.

**Lift.** Relative to the previous bridge-recovery frontier, this patch raised dev coverage from $0.737$ to $0.787$ ($221/300$ to $236/300$), a $+5.0\%$ improvement. Relative to the original baseline, the resulting workflow was $+34.0\%$ higher.

#### F.5.3 Case 3: Recovered Bridge Pages Needed to Become New Evidence Sources

**Diagnosis.** The most surprising discovery was that the all-targeted-pass candidate was not the best dev candidate. A candidate that solved all targeted training examples scored $0.787$ on dev, but CRO found a more general mechanism: recovered bridge pages should themselves be re-read as evidence sources. In one persistent failure, the workflow recovered *Volkswagen CrossBlue* but still failed to recover the second-order comparison page *Honda Pilot*.

**How the LLM arrived at the diagnosis.** The proposer inspected the trace and saw that the workflow treated recovered bridge pages as endpoints. It also observed that earlier recursive bridge attempts failed when they replaced the late missing-evidence resolver. The successful hypothesis was therefore compositional: keep the late resolver, but add recursive reading over relation-bearing sentences from newly recovered pages.

**Generated patch.** CRO added relation-cue filtering and recursive bridge expansion over recovered titles:

```
relation_cues = [
    "competed", "introduced", "based on", "inspired by",
    "starred", "founded", "won", "record"
```

```
]

def relation_sentences(claim, title, snippet):
    return [
        sent for sent in split_sentences(snippet)
        if has_relation_cue(sent, relation_cues)
        or token_overlap(sent, claim + " " + title) >= 4
    ]

bridge_expansion = collect_recursive_bridges(
    claim=claim,
    seed_titles=merge_titles(bridge_titles, snippet_bridges),
    known_titles=first_retrieved,
    max_depth=2,
)

retrieved = merge_titles(
    first_retrieved,
    bridge_titles,
    bridge_expansion,
    second_retrieved,
    gap_titles,
    third_retrieved,
)
```

This changed the workflow from "find a bridge page" to "find a bridge page and inspect it for the next bridge."

**Lift.** This patch raised dev coverage from $0.787$ to $0.797$ ($236/300$ to $239/300$), a further $+1.0\%$ improvement and $+35.0\%$ over baseline. Notably, its targeted-train score was lower than the previous candidate ($0.833$ vs. $1.000$), but its aggregate dev score was higher. CRO selected a more general workflow mechanism rather than the edit that best fit the targeted training slice.

### F.6 Meta-Agent Guided Tree-RL: full training configuration

Training is performed on Modal-managed $8\times$H100 nodes using SkyRL's GRPO recipe with FSDP2, gradient checkpointing, and `torch.compile` on the policy. Each training step uses a batch of 16 prompts with 8 samples per prompt for 128 rollouts per step. Each rollout caps at 8 turns, 1024 maximum generated tokens per turn, and 16,384 maximum input length; trajectories that would exceed these caps are filtered via SkyRL's overlong-filtering mode. The optimizer is Adam with learning rate $5\times10^{-7}$, weight decay 0.01, and gradient clipping at max-norm 0.1, with a 20-step linear warm-up followed by constant schedule for ten epochs over the training set (1,120 total steps). KL loss is disabled; advantages are GRPO-normalized and standardized per group. Inference uses vLLM with four engines at tensor-parallel size 2, weight synchronization via NCCL, and `gpu_memory_utilization=0.80`. Checkpoints and validation evaluations are taken every 10 training steps. The canonical launch script is `experiments/a4-reversible-rl/modal_smoke_train.py` in the released codebase.

### F.7 Meta-Agent Guided Tree-RL: meta-agent qualitative examples

The tree-search rollouts in Section 5.3 hand the fork choice to a stronger model (`claude-opus-4-7`). Given the inner agent's full transcript and final reward, it returns a turn $t^* \in [1, T-2]$ to fork at and the bash command it would run there. We re-render the command through the policy's renderer, so the tokens match what the policy itself would emit, and inject it as one of the $K$ branches at $t^*$.

This subsection walks through three trajectory shapes that show the meta-agent's behaviour: an early single-cause failure, an ambiguous failure with multiple plausible branches, and a long trajectory with a deeply-nested mistake. The trajectories are short enough to read end-to-end and were hand-constructed so a careful reader has a strong prior over the "right" branch turn. We check whether the meta-agent's pick matches that prior, whether its emitted command actually addresses the failure,

**Algorithm 2** Meta-Agent Guided Tree-RL

**Require:** Policy $\pi_\theta$; meta-agent $P$; task pool $\mathcal{D}$; group size $G$; branch factor $K$; iterations $T$

**for** $t \leftarrow 1$ to $T$ **do**
  Sample prompt batch $\{q_b\}_{b=1}^{B} \sim \mathcal{D}$.
  **(A) Root Rollouts**:
  **for** each $q_b$ in parallel, each $g \in \{1, \ldots, G\}$ in parallel **do**
    Initialize a fresh sandbox $\sigma_{b,g}$.
    $\tau_{b,g}^{\text{root}} = (o_0, a_1, o_1, \ldots, a_{T_{b,g}}, o_{T_{b,g}}) \sim \pi_\theta(\cdot \mid q_b, \sigma_{b,g})$
    $R_{b,g}^{\text{root}} \leftarrow \textsc{Grade}(\tau_{b,g}^{\text{root}})$
  **end for**
  **(B) Meta-Agent Branching**:
  **for** each root $\tau_{b,g}^{\text{root}}$ in parallel **do**
    $t^* \leftarrow P(\tau_{b,g}^{\text{root}})$ ▷ meta-agent picks fork step $t^*$
    $\sigma^* \leftarrow \textsc{Revert}(\tau_{b,g}^{\text{root}}, t^*)$ ▷ env state rolled back to right before step $t^*$
    $\tau_{b,g}^{\text{pre}} \leftarrow (o_0, a_1, o_1, \ldots, o_{t^*-1})$ ▷ shared trajectory prefix for all $K$ branches
    **for** $k \in \{0, \ldots, K-1\}$ in parallel **do**
      $\sigma_k \leftarrow \textsc{Fork}(\sigma^*)$ ▷ isolated env per branch
      $\tau_{b,g}^{(k)} \sim \pi_\theta(\cdot \mid q_b, \sigma_k, \tau_{b,g}^{\text{pre}})$ ▷ $\pi_\theta$ rollout from $t^*$ in $\sigma_k$
      $R_{b,g}^{(k)} \leftarrow \textsc{Grade}(\tau_{b,g}^{(k)})$
    **end for**
  **end for**
  **(C) Credit Assignment**:
  *Inter-root advantage* for prefix actions $j < t^*$ (shared by all branches of root $g$):
    $A_{b,g}^{\text{inter}} \leftarrow R_{b,g}^{\text{root}} - \frac{1}{G}\sum_{g'=1}^{G} R_{b,g'}^{\text{root}}$ ▷ $G$-root group baseline
  *Intra-tree advantage* for suffix actions $j \geq t^*$ on tree member $k \in \{\text{root}, 0, \ldots, K-1\}$:
    $A_{b,g}^{\text{intra},(k)} \leftarrow R_{b,g}^{(k)} - \frac{1}{K+1}\left(R_{b,g}^{\text{root}} + \sum_{k'=0}^{K-1} R_{b,g}^{(k')}\right)$ ▷ $(K+1)$-sibling baseline; $R_{b,g}^{(\text{root})} \equiv R_{b,g}^{\text{root}}$
  Update $\pi_\theta$ via clipped GRPO with $\{A_j\}_j$ pooled across all $q_b$ in the batch.
**end for**
**return** trained policy $\pi_\theta$

and whether it stays well-behaved across trajectory lengths. (Three further cases in the released probe set, omitted for space, cover clean-success no-ops, late-turn range clamping, and a redundant tool-replacement variant of Case 1; results match the patterns shown here.)

**Setup.** For each case, we render the inner-agent trajectory (system prompt, user task, and the alternating assistant tool calls and tool results) as a numbered turn list and send it to the meta-agent along with the final reward. The meta-agent returns a JSON object `{branch_turn, tool_call, reason}`, where `tool_call.name = "bash"` and `tool_call.arguments.command` is the bash payload.

**Headline.** The meta-agent's choices share two patterns uniform sampling cannot reproduce. First, it prefers the earliest turn at which a downstream wrong choice was made, even when that turn was not itself a syntactically obvious failure (Case 1: package-name typo; Case 3: target-file choice). Second, when the trajectory shows symptom-then-cause dynamics (Case 2: a defunct process surfaces at turn 5, traceable to a launch issue at turn 3), it follows the causal chain rather than picking the symptom turn. Demanding an executable bash command, not just a turn index, forces the meta-agent past hand-wavy diagnoses (Case 2: a specific `nohup` invocation, not "use proper launch flags"), and the command round-trips through the same renderer the policy uses, so the injected branch's tokens are indistinguishable from the policy's own emission at parse time.

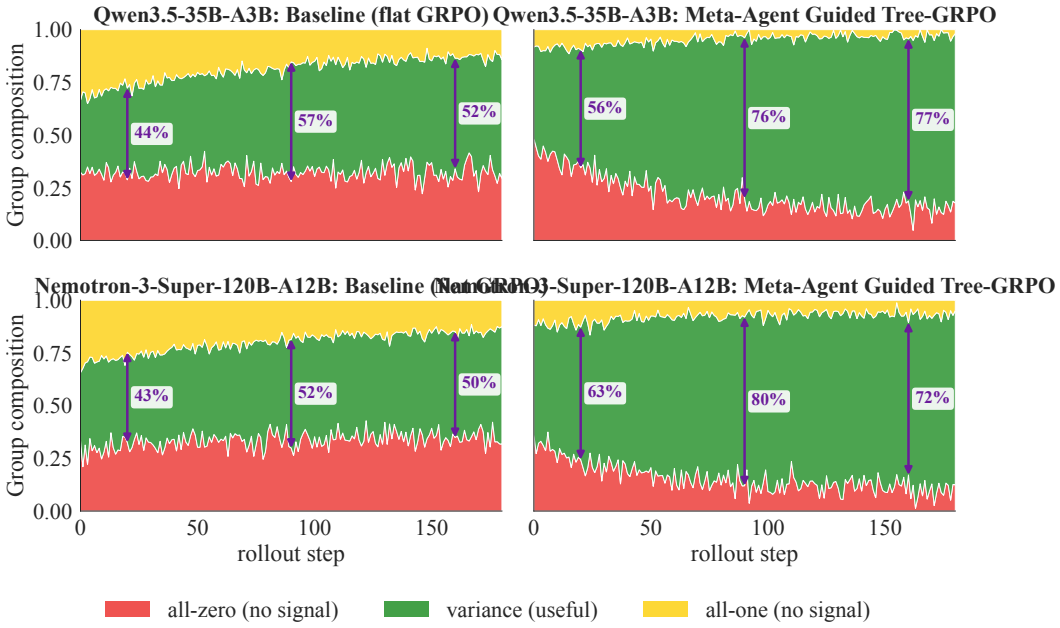


Figure 17: GRPO group composition over training (rows: base model; columns: setting). Tree-GRPO keeps the *informative* (variance, green) fraction higher than Flat GRPO throughout, producing more gradient signal at matched compute. Flat GRPO's all-one share grows with training as easy tasks saturate, eating the variance band. Purple double-headed arrows annotate the variance share at steps 20/90/160.

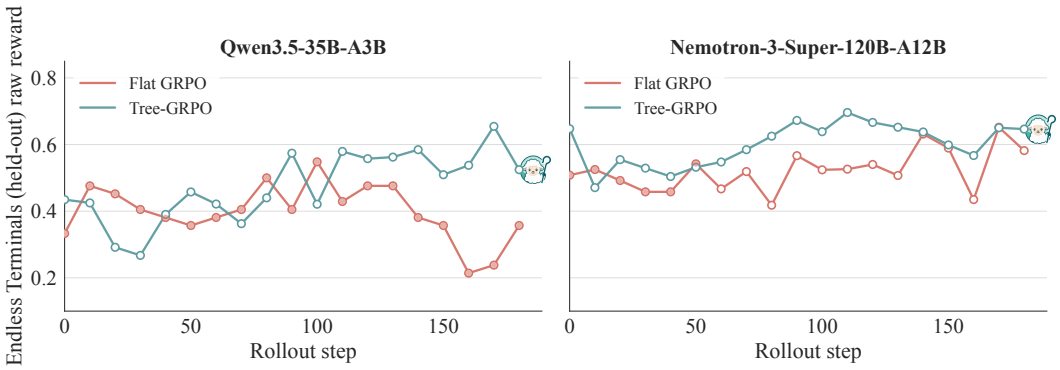


Figure 18: Held-out Endless Terminals evaluation, sampled every 10 training steps (raw, unsmoothed). Open circles plot the actual measured `val/endless_mean_reward`; solid line is a noise-preserving fit. Tree-GRPO climbs to and stabilises at a higher held-out reward than Flat GRPO on both base models. Final-policy Terminal-Bench 2.0 transfer is reported separately in Table 5.

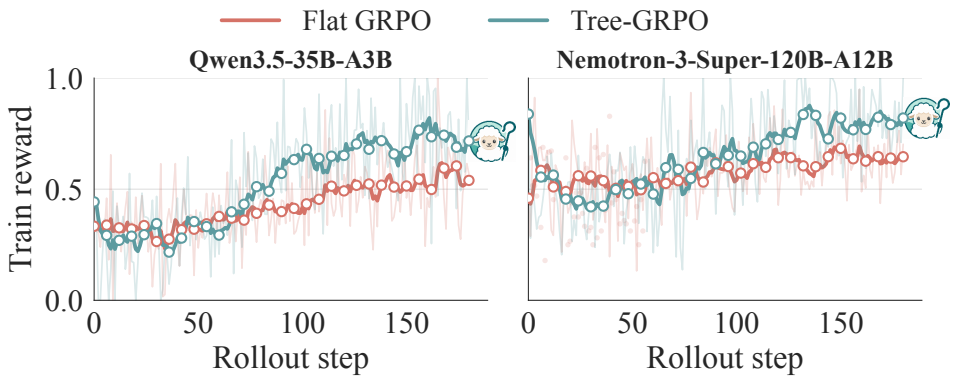


Figure 19: Train raw reward (mean over $G$=8 roots) for both base models, panels are Qwen3.5-35B-A3B (left) and Nemotron-3-Super-120B-A12B (right). Tree-GRPO ($K$=4, teal) reaches higher reward than Flat GRPO (red) at every rollout step. Faint dots are observed steps from the flat-baseline run; smooth lines are denoised trajectories.

**Case 1: Early mistake (T=4, reward=0.00)**

**Task:** Install the `requests` package and verify it imports.

**Trajectory:**

- **Turn 1:** `pip install request` *(typo: missing the s)* → "Successfully installed request-2025.10.0"
- **Turn 2:** `python -c "import requests"` → `ModuleNotFoundError: No module named requests`
- **Turn 3:** `python3 -c "import requests"` → same error.
- **Turn 4:** prints `sys.path`, gives up.

**Meta-agent decision (obvious turn: 1)**

**branch_turn: 1** ✓ (matches obvious)

**meta-agent action:** `pip install requests && python -c "import requests; print(requests.__version__)"`

**reason:** "Turn 1 installed the wrong package 'request' instead of 'requests'; fixing the typo here resolves the entire failure."

Figure 20: Early-mistake case. The wrong package name on turn 1 dooms the rest of the trajectory. The meta-agent picks turn 1 and writes a one-liner that installs the right package and verifies the import in the same command.

**Case 2: Ambiguous failure (T=5, reward=0.30)**

**Task:** Serve the Flask app at `/app/app.py` on port 5000.

**Trajectory:**

- **Turn 1:** read the app source.
- **Turn 2:** `pip install flask` (no python-version check).
- **Turn 3:** `python app.py &` (background launch, no `nohup`).
- **Turn 4:** `curl localhost:5000/` → `Connection refused`.
- **Turn 5:** `ps aux` reveals the process is `<defunct>`.

**Meta-agent decision (multiple plausible turns)**

**branch_turn:** 3

**meta-agent action:** `cd /app && nohup python app.py -host=0.0.0.0 > /tmp/flask.log 2>&1 &`

**reason:** "The app.py has no __main__ block, so running it directly exits; need to bind to 0.0.0.0 with `nohup` to keep it alive."

Figure 21: Ambiguous case. At least three turns offer plausible branches (skip-the-version-check, pip-vs-pip3, the background launch). The meta-agent traces the symptom at turn 5 (`<defunct>`) back to the launch on turn 3 and writes a concrete replacement: `nohup` with a log redirect and an explicit host binding. Demanding a runnable command pushes the meta-agent past hand-wavy diagnoses to a specific hypothesis we can run.

**Case 3: Long trajectory, deep mid-mistake (T=9, reward=0.20)**

**Task:** Fix the failing test in `/app/test_utils.py`.

**Trajectory:**

- **Turns 1–3:** explore, read `utils.py` (case mismatch in `normalize`), run pytest, see the failing assertion.
- **Turn 4:** edits *the test file* with a fragile `sed` substitution *(wrong file—fix belongs in `utils.py`)*.
- **Turns 5–7:** doom-spiral—broken sed quotes, more sed patches, file syntax errors.
- **Turns 8–9:** attempts `git checkout` to revert, but repo is not git-tracked.

**Meta-agent decision (obvious turn: 4)**

**branch_turn: 4** ✓ (matches obvious)

**meta-agent action** (multi-line heredoc):

```
cat > /app/utils.py <<'EOF'
def normalize(s):
    return s.strip()
EOF
```

**reason:** "Fix the implementation rather than the test, since the task is to fix the failing test by correcting the underlying code."

Figure 22: Long-trajectory case. A 9-turn rollout with a wrong-file edit at turn 4 cascades into 5 turns of recovery thrash. The meta-agent picks turn 4 and writes a multi-line heredoc that rewrites `utils.py` (the file the agent should have edited) cleanly. Uniform sampling has no way to prefer turn 4 over the five recovery turns; only the meta-agent's pick concentrates the branch budget where it matters.