A Native Rust Re-implementation of the SIA Self-Improving-Agent Framework: Architecture, Fidelity, and Extensions

Micah Stubbs[*]
Superradiant
micah@superradiant.ai

Will Stark
Superradiant
will@superradiant.ai

June 6, 2026

Abstract

SIA (Self-Improving AI with Harness & Weight Updates) is a framework in which a Meta-Agent writes and improves a Target-Agent, the Target-Agent executes a task, and a Feedback-Agent analyzes the resulting trajectory to propose the next improvement [1]. This paper is a system and experience report on `sia_rust`, a native Rust re-implementation of the SIA framework plus a set of safety, observability, and scheduling extensions.

We describe three bodies of work. First, a **faithful port** of the deterministic SIA core (orchestrator, context/trajectory management, prompt system, run-directory format, and web visualizer) that reproduces the reference Python implementation's prompt, context, and execution-log outputs **byte-for-byte**, verified by a differential-parity harness over an ASCII + CJK + emoji + control-character matrix. Second, a **native multi-provider LLM agent-runner layer** built on `rig-core` and injectable HTTP transports, comprising three runners (a Claude `/v1/messages` tool-use loop, an OpenHands-style OpenAI-compatible loop, and a PydanticAI-style loop), with trajectory-capture middleware, a provider/profile mapping layer, retry/resilience decorators, and a structured-output parity harness — the entire LLM layer is feature-gated and driven through injectable transports so its full tool-use loops are tested offline with scripted responses. Third, a set of **extensions** scoped honestly as engineering primitives and seams: a pure-`std` capability allow-list and a formal threat model for self-modifying agents; a native `Verifier` trait with adversarial-robustness hooks; per-generation telemetry; an Axum-based "SIA Studio" dashboard; a heuristic adaptive harness-vs-weight-update scheduler; and a native weight-update *abstraction* with a tested pure-CPU LoRA *reference* implementation.

We evaluate the port via differential parity, deterministic microbenchmarks (the Rust core runs roughly **5.8× faster** by geometric mean across nine operations, up to $\sim$25× on prompt building and $\sim$18× on execution-log loading, on the machine in `benchmarks/REPORT.md`), and offline test coverage including mocked tool-loop tests and golden-master trajectory round-trips. We are explicit about what is **not** evaluated and what remains future work: a full meta-RL scheduler (only a heuristic exists; UCB/contextual-bandit stages are not implemented), GPU LoRA / real weight-update training (only a CPU reference plus an abstraction; Candle is not integrated), OS-level sandbox enforcement (only an advisory capability allow-list plus a roadmap; landlock/seccomp/WASI are not wired in), and any live end-to-end self-improvement accuracy study.

Keywords self-improving agents · Rust · LLM agents · reproducibility · sandboxing

[*]This is a system-and-design / experience report on the `sia_rust` codebase, not a results paper: its empirical content is limited to (a) differential byte-parity tests against the reference Python implementation, (b) deterministic microbenchmarks, and (c) offline unit/integration tests. End-to-end self-improvement accuracy studies are explicitly future work. All quantitative claims are drawn from artifacts in the repository and cited to their source.

1 Introduction

Self-improving agent systems repeatedly read, write, and execute machine-generated code on the host. Three properties matter for such systems and are in tension with the rapid-prototyping languages typically used to build them:

- **Speed** of the deterministic scaffolding (prompt assembly, context construction, trajectory/log I/O), which is on the critical path of every generation but does no model inference itself.
- **Safety**: because the agent's output is itself executable code (and the tool arguments are chosen by an LLM that can be steered by task data or prompt injection), the execution sandbox is a first-class security concern rather than an afterthought.
- **Observability**: the trajectory logs, scores, and context that drive the next generation must be captured faithfully, or the self-improvement signal degrades.

Rust is an attractive substrate on these axes: it provides predictable, allocation-controlled performance without a GC; a strong type system and explicit error handling that make the orchestration logic auditable; and a path toward capability-based and OS-level isolation. This paper reports on `sia_rust`, which ports the SIA framework to Rust and extends it along these three axes, while preserving the original task contract so existing Python tasks run unchanged.

We frame this as a **system and experience report**, not a results paper. Our empirical claims are confined to fidelity (byte-parity), core-operation speed (microbenchmarks), and test coverage. We do not claim accuracy improvements from self-improvement, and we mark every not-yet-evaluated component as future work.

1.1 Contributions

Each contribution is tied to the module(s) that implement it.

1. **A faithful, byte-parity port of the SIA deterministic core** — orchestrator, context manager, prompt system, results, providers/profiles, run-directory format, and CLI (`src/orchestrator.rs`, `src/run.rs`, `src/context_manager.rs`, `src/prompts.rs`, `src/results.rs`, `src/providers.rs`, `src/profiles.rs`, `src/cli.rs`), with a CPython-faithful JSON serializer (`src/pyjson.rs`) and a differential-parity helper (`src/bin/sia_parity.rs`, `scripts/parity_check.py`).
2. **A native multi-provider LLM agent-runner layer** (`src/llm/`, behind the non-default `llm` cargo feature): an `AgentRunner` trait and trajectory types (`src/llm/mod.rs`, `src/llm/trajectory.rs`); three native runners — Claude `/v1/messages` tool-use loop (`src/llm/claude_runner.rs`), OpenHands-style OpenAI-compatible loop (`src/llm/openhands_runner.rs`), and PydanticAI-style loop (`src/llm/pydantic_ai_runner.rs`); injectable Anthropic and OpenAI-compatible transports (`src/llm/anthropic_api.rs`, `src/llm/openai_api.rs`); a `rig-core` wrapper runner (`src/llm/rig_runner.rs`); and shared sandboxed tool executors (`src/llm/tools.rs`).
3. **Trajectory-capture middleware** (`src/llm/trajectory_middleware.rs`) that records structured events, token usage, and wall-clock timing and renders into the exact `agent_execution.json` shape the orchestrator and web visualizer already consume.
4. **A provider/profile → transport mapping layer** (`src/llm/provider_mapping.rs`) that resolves a provider's `client_kind` (anthropic / openai / google), API-key env var, and base-URL default into a constructed transport.
5. **Retry / resilience middleware** (`src/llm/retry.rs`): a deterministic exponential-backoff policy and transport decorators with an optional secondary-transport fallback, implemented over the same transport traits so the behavior is drop-in.
6. **A structured-output extraction + parity harness** (`src/llm/structured.rs`) that mirrors the Python GPQA reference's answer-parsing semantics and wraps `rig-core`'s `Extractor` for the live path.
7. **A capability sandbox + formal threat model** (`src/sandbox.rs`, `SECURITY.md`): a pure-`std`, deny-by-default capability allow-list (the single auditable in-process enforcement point), plus a documented escalation/mitigation analysis and an OS-enforcement roadmap.
8. **A native verifier layer** (`src/verifier.rs`): a `Verifier` trait with reusable offline verifiers using the same $[0, 1]$ scoring semantics as the Python `evaluate.py` `accuracy` field, plus adversarial-variant generation and a stability check that makes the Goodhart/brittleness risk testable.
9. **Per-generation telemetry** (`src/llm/telemetry.rs`): token/call/timing accounting derived from already-captured run metrics, written as a `telemetry.json` artifact (no fabricated dollar costs).

10. **"SIA Studio" web visualizer** (`src/web/server.rs`, `src/web/runs.rs`): an Axum dashboard that serves and renders runs from the `runs/` directory.
11. **A heuristic adaptive harness-vs-weight-update scheduler** (`src/scheduler.rs`): a transparent, deterministic decision rule over an "improvement-per-compute" metric — explicitly a first step toward, not an implementation of, the SIA paper's meta-RL future-work item.
12. **A native weight-update abstraction + CPU reference LoRA** (`src/weights.rs`): a `WeightUpdater` trait, training-example extraction, an update-trigger seam, and a tested pure-`f64` CPU reference LoRA updater — explicitly *not* a GPU trainer and *not* integrated with a real model.
13. **A standalone eval harness** (`evals/`): a GPQA-style `dspy-rs` Signature / Module / accuracy Evaluator with an offline deterministic mock adapter and a real-provider path.

Contributions 7–13 are deliberately scoped as primitives, seams, and offline references. Section 3 states precisely what each does and does not yet do.

2 Background

2.1 The SIA framework

SIA [1] structures self-improvement as a loop over three roles. A **Meta-Agent** writes or edits the *harness* — the Target-Agent's code/scaffold. The **Target-Agent** runs the task and produces a trajectory and a score. A **Feedback-Agent** analyzes the trajectory and proposes the next improvement. The paper distinguishes two levers for improvement between generations: **harness updates** (editing the agent's prompt/scaffold, the lever the base loop uses) and **weight updates** (training model weights from high-reward trajectories). It calls out meta-RL over the harness-vs-weight decision policy, and verifier quality, as key future-work and limitation areas — points this work engages with directly but does not claim to resolve.

2.2 rig-core

`rig-core` is a Rust LLM/agent library [2]. In `sia_rust` it supplies the Anthropic provider behind `RigAgentRunner` and the `Extractor` used by the structured-output live path, and its exact version is pinned (`=0.22.0`) to stay compatible with the `dspy-rs`-based eval crate. The three native runners do not delegate their tool-use loops to `rig`'s agent abstraction; they drive thin, injectable HTTP transports directly so the loops are fully testable offline (Section 4.2).

2.3 Trajectory-based improvement

The fidelity of the captured trajectory is what makes Feedback-Agent decisions meaningful. SIA persists trajectories as Anthropic-style `agent_execution.json` (an array of `{role, content}` messages with `text` / `tool_use` / `tool_result` content blocks) and, for the OpenHands path, as per-event JSON files under `openhands_trajectory/`. `sia_rust` reproduces these shapes so the same web visualizer and context-loading code consume native and ported outputs without an adapter.

3 System Design & Architecture

3.1 Overview

The orchestrator is the trusted policy decision point. It resolves the task, loads profiles/providers, sets up the run directory and (Python) virtual environment, builds prompts, dispatches the meta/feedback agents, runs the Target-Agent as a Python subprocess via the `evaluate.py` contract, evaluates the result, and tracks context and feedback for the next generation. The meta/feedback agents are driven by the native Rust LLM layer when built with `-features llm`; the Target-Agent is *always* a real Python subprocess, because it is generated code that must run exactly as the paper intends and often depends on the Python ML/task ecosystem.

The architecture layering is summarized in Figure 1.

This phased design — native meta/feedback runners, a Python bridge only for target execution — yields safety and performance gains while preserving the task contract existing custom tasks rely on.

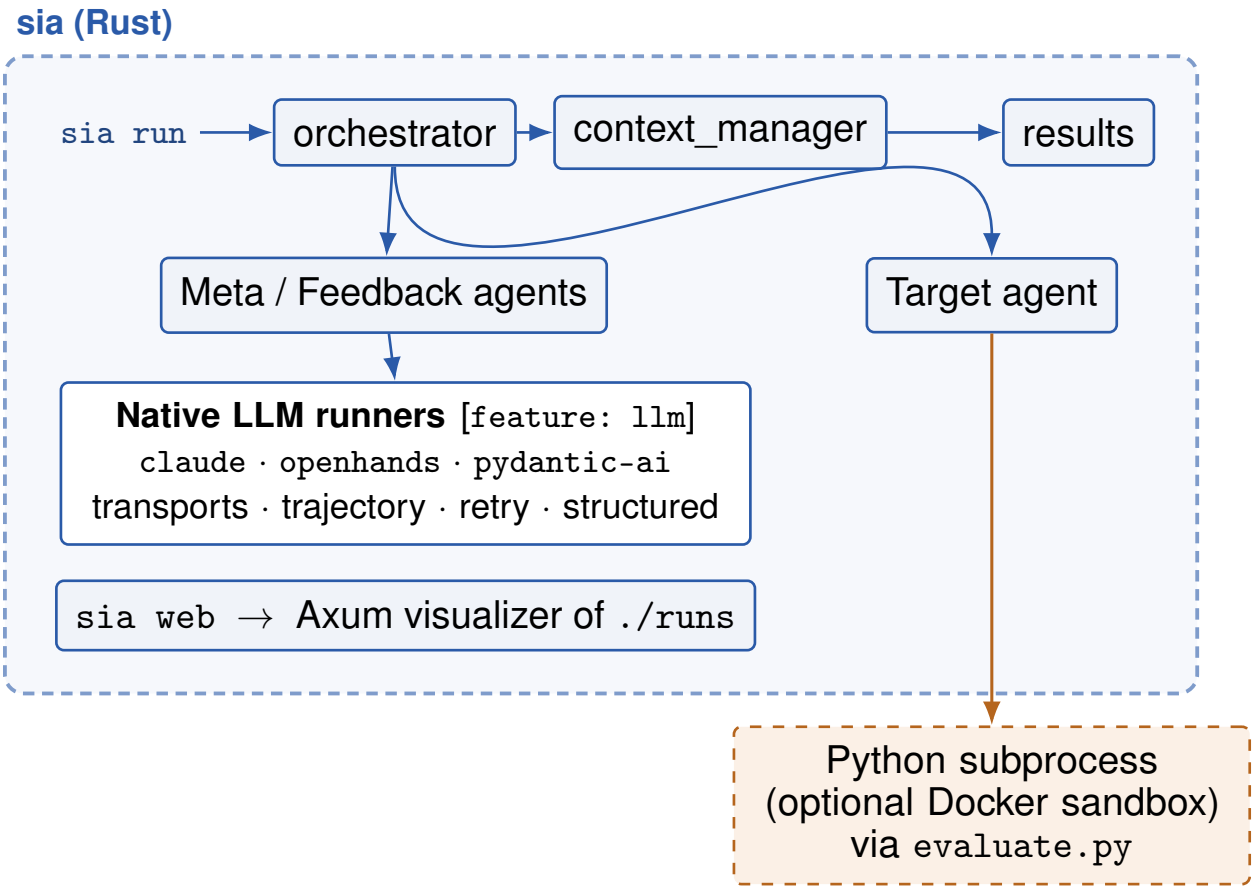

Figure 1: The `sia_rust` architecture. Native Rust meta/feedback runners drive the LLM agents inside the trusted boundary; the Target-Agent is always executed as a separate Python subprocess (optionally Docker-sandboxed) via the `evaluate.py` contract.

3.2 The native LLM layer

The LLM layer (`src/llm/`, feature `llm`) centers on a small synchronous `AgentRunner` trait (`src/llm/mod.rs`) returning a final text plus a captured `AgentTrajectory`. Three native runners implement the meta/feedback agents:

- **claude** (`src/llm/claude_runner.rs`): a direct `/v1/messages` tool-use loop. A thin Messages-API client is used rather than `rig`'s agent abstraction so the loop has an injectable transport (`MessagesTransport`), exact `tool_use` / `tool_result` block plumbing, and `stop_reason` control flow — all testable offline with scripted responses.
- **openhands** (`src/llm/openhands_runner.rs`): an OpenAI-compatible chat-completions loop exposing `terminal` and `file_editor` tools, persisting events in the OpenHands `openhands_trajectory/.../events/event-*.json` shape that the web visualizer renders. It is the native replacement for the Python OpenHands (litellm) integration; model-spec routing to an OpenAI-compatible `base_url` is handled by `HttpChatTransport` plus `resolve_model` prefixing.
- **pydantic-ai** (`src/llm/pydantic_ai_runner.rs`): a `write_file` / `read_file` / `bash` loop with a request limit matching Python's `UsageLimits(request_limit=max_turns)`. It reuses the in-tree OpenAI-compatible transport, shared tool executors, and trajectory middleware rather than taking a new agent dependency.

All three reuse the shared sandboxed `Bash` / `Read` / `Write` / `Edit` / `Glob` executors in `src/llm/tools.rs`.

3.3 Trajectory middleware

`TrajectoryMiddleware` (`src/llm/trajectory_middleware.rs`) is a reusable observer that records structured `TrajectoryEvents` (user prompt, assistant text turns, tool calls, tool results, errors) and tracks token usage and wall-clock timing. It mirrors each event into an `AgentTrajectory` via that type's public `push_*` builders, so it reuses the trajectory storage rather than reinventing it, and its output is in the exact `agent_execution.json` shape the orchestrator reads back with no adapter. API-reported usage captured here is the single source of truth for telemetry (Section 3.7).

3.4 Provider mapping

`provider_mapping.rs` is a single source of truth from a `Provider` config to a constructed transport. It resolves the `client_kind` (anthropic / openai / google), the API-key env var, and a per-kind base-URL default (anthropic → `api.anthropic.com`, openai → `api.openai.com/v1`, google → the OpenAI-compatible `generativelanguage.googleapis.com/...` endpoint driven through the chat transport), and emits helpful user-facing errors. This removes hand-rolled transport construction from the individual runners.

3.5 Retry / resilience

`retry.rs` provides a deterministic exponential-backoff `RetryPolicy` (jitter is applied only in the sleeping path so the schedule itself is exactly unit-testable), a transient-error classifier, a reusable retry loop with an `on_attempt_failed` hook for trajectory capture, and `RetryMessagesTransport` / `RetryChatTransport` decorators that implement the same transport traits as what they wrap (so retry is drop-in) and support an optional secondary-transport fallback. Backoff is synchronous (`std::thread::sleep`) to match the blocking transports' call sites.

3.6 Structured-output parity

`structured.rs` provides offline structured-output extraction plus a wrapper over `rig-core`'s `Extractor` for the live path. It mirrors the Python GPQA reference's `parse_answer_letter` semantics: `.strip().upper()` normalization of the structured `answer` field, a fall-back to raw-text uppercasing on JSON-parse failure, a final letter that is the value if it is exactly one of A–D else the first A–D letter scanning left-to-right, and a non-raising empty-string fallback when no A–D letter is present.

3.7 Telemetry

`telemetry.rs` turns the run metrics the runners already capture (token usage, tool calls, timing) into per-generation `GenerationTelemetry` plus a cumulative summary, written as a `telemetry.json` artifact next to `agent_execution.json`. It performs no re-accounting — the API-reported usage recorded by the trajectory middleware is the single source of truth — and deliberately records **no** dollar-cost field because per-provider pricing is unknown.

3.8 SIA Studio

The web layer (`src/web/server.rs`, `src/web/runs.rs`) is an Axum app — the "SIA Studio" dashboard — that serves the runs visualizer over the `runs/` directory, with routes for listing runs, run detail, evaluation details, artifacts, and trajectories. It can run in the foreground (`sia web`) or on a daemon thread so the orchestrator can expose a live dashboard during `sia run`. The bundled single-page UI is embedded at build time via `include_dir`.

3.9 Capability sandbox + threat model

`SECURITY.md` is a formal threat model for self-modifying agents: it enumerates assets (host filesystem, environment secrets, network egress, host compute, run integrity), three trust boundaries (trusted orchestrator; semi-trusted LLM-driven native runner tools; untrusted Python target subprocess), escalation paths (filesystem escape, network exfiltration, resource exhaustion, prompt-injection-driven tool abuse), current mitigations, and a three-stage OS-enforcement roadmap.

`src/sandbox.rs` implements stage 1: a pure-`std`, dependency-free `Capabilities` allow-list compiled on the **default** build. It is deny-by-default (read/write only within a declared `fs_root`, `Bash` gated, network denied, a per-file size cap) with `permissive` and `read_only` presets, and exposes `check_read` / `check_write` / `check_bash`

/ `check_within_root` / `check_size`. It is the single auditable enforcement point the landlock/WASI roadmap builds on. Crucially, it is **advisory and in-process**: it raises the bar against injection and accidental escape but does not constrain a process that ignores it (Section 6.1).

3.10 Verifier layer

`src/verifier.rs` defines a `Verifier` trait and reusable, fully-offline verifiers that score a single `(submission, reference)` pair without spawning Python, using the same $[0, 1]$ semantics as the Python `evaluate.py` `accuracy` field so the two are directly comparable. The Python `evaluate.py` remains the authoritative system-of-record for whole-dataset aggregation; the native path is scoped to unit-level checks. To make the SIA paper's verifier-quality (Goodhart) concern testable, the module ships `adversarial_variants` (semantically-equivalent perturbations: whitespace, case, trailing punctuation, wrapping prose) and `is_stable` (asserts a verifier returns the same outcome across all variants); every verifier is panic-free on malformed input, returning a failing outcome rather than erroring.

3.11 Heuristic adaptive scheduler

`src/scheduler.rs` targets the SIA paper's harness-vs-weight decision-policy future-work item with a deliberately simple, deterministic **heuristic**. For each lever it defines an "improvement efficiency" metric — mean positive score improvement per unit of compute over consecutive same-lever generations — and a transparent decision tree (`decide_next`) that reuses the weight module's plateau detector and an efficiency comparison to recommend the next `UpdateKind`. It is explicitly **not** meta-RL (no learned policy, no environment, no gradient over the decision) and ships as a standalone, tested module **not yet wired into the orchestrator**; the intended hookup (record per-generation, decide before the next generation, surface an efficiency summary in SIA Studio) is documented as a seam.

3.12 Native weight-update abstraction + CPU reference

`src/weights.rs` provides the honest first step toward weight updates: a `WeightUpdater` abstraction, `TrainingExample` extraction from high-reward trajectories, a `should_trigger_weight_update` trigger seam, and `LoraReferenceUpdater` — a tested, pure-`f64`, fully-offline CPU reference that runs LoRA gradient mechanics end-to-end (`W_eff = (B·A)·(alpha/rank)`, gradients through both factors, decreasing loss on a learnable signal). It is explicitly a **reference**, not a GPU trainer: it embeds examples into a small fixed-dimensional feature vector and does reward-weighted MSE gradient descent; it does **not** load a real model, run on a GPU, or update a real LLM's parameters. The intended real backend is Candle behind a future non-default cargo feature; it is not depended on here because `candle-core` is not in this build's offline cargo cache. The Python bridge is **not** a training path — this module spawns no subprocess and touches no Python.

4 Implementation & Engineering

4.1 Feature-gated `llm` build

The entire native LLM layer is gated behind the non-default `llm` cargo feature (`Cargo.toml`: `llm = ["dep:rig-core", "dep:reqwest", "dep:glob", "dep:schemars"]`) so the default build and published crate stay dependency-light. Without the feature, the meta/feedback runners return a clear "build with `-features llm`" message and everything else (orchestration, web UI, parity) still works. CI exercises **both** the default build and `-features llm`.

4.2 Injectable-transport offline testability

Every native tool-use loop is driven through an injectable transport (`MessagesTransport` / `ChatTransport`). In-test mock transports return scripted response sequences, so the full loops — including tool execution, tool-result feedback, and `stop_reason` control flow — are tested offline with no network and no API keys. Real-provider calls are `#[ignore]`d and gated on the relevant API key. The end-to-end test (`tests/end_to_end_llm.rs`) proves the native loops integrate with the artifacts the orchestrator/context pipeline consume and that dispatch routes to the native runners rather than the feature-off boundary error.

4.3 Byte-parity methodology

`src/pyjson.rs` reproduces CPython's `json.dumps(..., ensure_ascii=True)` exactly, including the `\uXXXX` escaping needed for CJK content (e.g. LawBench). The `sia-parity` binary (`src/bin/sia_parity.rs`) emits the

Table 1: Core-operation microbenchmarks (ns/op) on the recorded machine. Speedups are order-of-magnitude indicators, not portable constants (see text).

Operation	Python ns/op	Rust ns/op	Speedup
`build_meta_prompt`	13,616	551.4	24.7×
`build_feedback_prompt`	1,357	519.9	2.6×
`context_manager_run`	1,260,413	1,041,368	1.2×
`build_feedback_context_single`	200,225	14,333	14.0×
`build_feedback_context_multi`	283,415	27,682	10.2×
`load_agent_execution_single`	125,182	6,997	17.9×
`load_agent_execution_multi`	199,860	25,149	7.9×
`web_list_runs`	3,485,486	1,956,338	1.8×
`web_get_run`	684,079	253,380	2.7×

Rust output for a given operation from a JSON request on stdin; `scripts/parity_check.py` runs the reference Python implementation and the Rust helper on the same inputs — `json.dumps` with indent, meta/feedback prompts, feedback context, and execution-log loading — over an ASCII + CJK + emoji + control-character matrix and asserts byte-identical output. This parity gate runs in CI on every push.

4.4 Embedding and dual-build CI

Package-data that Python loaded via `importlib.resources` (bundled provider/profile JSON and the web `index.html`) is embedded at build time via `include_dir`. CI (`.github/workflows/rust.yml`) runs `cargo fmt -check`, `cargo clippy -D warnings`, and `cargo test` for both the default and `-features llm` builds, plus the differential-parity gate and the standalone `evals/` crate.

5 Evaluation

We report only what the repository actually measures. We do **not** report any live self-improvement accuracy study (Section 5.4).

5.1 Differential parity (fidelity)

`scripts/parity_check.py` asserts byte-identical output between the reference Python implementation and the Rust port across the operations in Section 4.3, over an ASCII + CJK + emoji + control-character matrix (e.g. the Chinese-charge LawBench-style fixture 故意伤害罪, the `ok ✓ [emoji] done` emoji fixture, and a `\t\n\r` control-char fixture). The byte-for-byte criterion — rather than semantic equality — is what lets the same web visualizer and context-loading code consume native and ported outputs interchangeably. The port was developed red→green with every Python test and all seven golden-master files mirrored in Rust.

5.2 Microbenchmarks (core-operation speed)

Deterministic core operations are measured in both languages with byte-identical fixtures (Rust via Criterion `cargo bench -bench core`; Python via `benchmarks/bench_python.py`); `benchmarks/run_comparison.py` regenerates `benchmarks/REPORT.md`. LLM calls are neutralized on both sides so the benchmark isolates the deterministic core. On the machine recorded in `benchmarks/REPORT.md` (4-core Xeon @ 2.80 GHz, Linux, CPython 3.11.15, rustc 1.94.1), the reported figures are given in Table 1.

The **geometric-mean speedup across the nine operations is** $\sim 5.8\times$, with prompt building up to $\sim 25\times$ and execution-log loading up to $\sim 18\times$. As `benchmarks/REPORT.md` itself cautions, the ns/op figures mix CPU work with filesystem I/O for the fs-backed ops and are sensitive to machine load and OS file-cache state, so these speedups should be read as order-of-magnitude indicators on this machine, not precise or portable constants. They characterize the deterministic scaffolding only and say nothing about end-to-end run time, which is dominated by model inference latency.

5.3 Test coverage

The native loops are covered by offline mocked tool-loop tests (Section 4.2). The trajectory types support golden-master round-trips: middleware output renders into the `agent_execution.json` shape and is

read back by `load_agent_execution` without an adapter. Where the Python tests patch `subprocess.run` / `subprocess.Popen`, the Rust port exposes injectable seams (`run_evaluation_with`, `run_target_agent_with`, `run_generation_with`) so the branching logic is unit-tested without a real interpreter. The standalone `evals/` crate runs a GPQA-style `dspy-rs` pipeline fully offline via a deterministic mock adapter, asserting a perfect mock scores 100% and an always-"A" mock scores the expected lower value (20% on the bundled five-question fixture); it also offers a real-provider path. These are infrastructure/correctness tests, **not** a self-improvement accuracy study.

5.4 What is NOT evaluated

There is **no** live end-to-end self-improvement study here: we do not report that the loop raises task accuracy across generations, nor any scheduler ablation, nor any real weight-update training run. The scheduler and weight modules are tested in isolation and are not yet wired into the orchestrator; the weight updater is a CPU reference, not a real-model trainer. A runnable example task exists, but it is a demonstration, not a benchmark study. Producing such studies is future work (Section 6.2).

5.5 Reproducibility statement

The repository defines a single authoritative reproducibility standard in `docs/REPRODUCIBILITY.md`: the exact per-run and per-generation artifact set (real filenames + JSON schemas), the mandatory metadata to report (git commit, profiles, model slugs, provider/`client_kind`, task, generation count), and a third-party verification path that separates *verifying the implementation* (offline, no keys: `cargo test`, the `scripts/parity_check.py` parity gate, and `sia web`) from *reproducing a live run* (keys required). It is explicit that, because LLM sampling is involved and **no random seed is set or recorded**, live runs are not bit-reproducible: reproducibility here means the same artifact set/schema plus the byte-parity-tested deterministic core (Section 5.4 notwithstanding), not identical model text. Consistent with the above, a live end-to-end run remains **unvalidated** (issue #93).

6 Discussion, Limitations & Future Work

6.1 Limitations (today)

- **Bash / sandbox gaps.** The native `Bash` tool runs arbitrary `sh -c` commands with only a wall-clock timeout — no command allow-list, no network restriction, no memory/CPU cap at that layer — and is **not yet wired** to the capability layer's `check_bash`. The lexical path sandbox blocks `..` and absolute paths but **does not resolve symlinks** (TOCTOU/symlink traversal remain possible). The capability layer (`src/sandbox.rs`) is **advisory and in-process**, not kernel-enforced. The default `-sandbox none` offers no confinement for the target subprocess; `-sandbox docker` (network-none, read-only dataset, cpu/mem caps) is the recommended mode for untrusted tasks. The Claude SDK runner path uses `bypassPermissions` to enable automation.
- **Heuristic scheduler.** The harness-vs-weight scheduler is a deterministic heuristic, not a learned policy, and is not yet wired into the orchestrator. UCB / contextual-bandit / meta-RL stages are **not implemented**.
- **Reference-only weight updates.** `src/weights.rs` is an abstraction plus a CPU `f64` reference LoRA that does not touch a real model or GPU. Candle is not integrated (it is not in the offline cargo cache).
- **No live improvement benchmark.** As stated in Section 5.4.
- **Benchmark portability.** The speedups are single-machine and mix CPU with I/O.

6.2 Future work

- **Full meta-RL scheduler.** Replace the heuristic `decide_next` (keeping its shape) with a learned harness-vs-weight policy; the intermediate UCB and contextual-bandit stages named in the project roadmap are the natural staging path and remain unimplemented.
- **Candle GPU LoRA.** A `CandleLoraUpdater` implementing `WeightUpdater` behind a non-default `weights-gpu/candle` feature [3], reusing the existing trait, `TrainingExample` extraction, and trigger unchanged, to perform real weight-update training from high-reward trajectories.
- **OS-level sandboxing.** Stage 2 (a Linux `landlock` filesystem ruleset scoped to `fs_root` plus a `seccomp` syscall filter, behind a `landlock-sandbox` feature) and stage 3 (running untrusted generated agents as WASI preview2 components under `wasmtime` [4] with only preopened directories and no ambient authority) — each *enforcing* the capability policy at a lower level so a tool-layer bug cannot ignore it. A code sketch exists in `src/sandbox.rs`; neither stage is wired in (the `landlock` crate is not in the offline cache).

- **End-to-end studies.** Once the scheduler and a real weight backend are wired into the orchestrator, run live multi-generation self-improvement studies (accuracy-vs-generation curves, scheduler ablations, cost/telemetry analysis) on the bundled tasks. None of this is claimed here.

7 Related Work

The closest prior work is the original **SIA** framework [1], which defines the meta/target/feedback loop, the harness-vs-weight-update distinction, and the future-work and verifier-quality concerns this work engages with; `sia_rust` is a native Rust re-implementation plus the extensions above. The native agent layer is built on **rig-core** [2], and the eval crate on **dspy-rs / DSRs** [5], the Rust port of DSPy. The roadmap toward stronger isolation references **Candle** [3] for native-Rust ML, and **landlock**, **seccomp**, and **WASI/wasmtime** [4] for OS- and VM-level enforcement. We keep this section modest and cite tools by their repositories rather than attributing them to papers we have not verified.

8 Conclusion

`sia_rust` is a native Rust re-implementation of the SIA self-improving-agent framework whose deterministic core reproduces the reference Python implementation's prompt, context, and execution-log output **byte-for-byte**, layered with a native multi-provider LLM agent-runner stack (three runners, injectable transports, trajectory middleware, provider mapping, retry, structured-output parity) and a set of honestly-scoped safety, observability, and scheduling extensions (capability sandbox + threat model, native verifier, telemetry, SIA Studio, a heuristic scheduler, and a CPU reference weight updater). Our evaluation is fidelity (byte-parity), core-operation microbenchmarks ($\sim$5.8$\times$ geomean on the recorded machine), and offline test coverage — and we are explicit that full meta-RL scheduling, GPU LoRA training, OS-level sandbox enforcement, and live end-to-end self-improvement studies are future work, not present results. We offer this as an experience report on building a faster, safer, more observable substrate for self-modifying agents, and as a stable set of seams others can build those future results on.

Note on citations. The SIA arXiv identifier is taken from this repository's README badge and links. Tool references point to project repositories rather than to papers, to avoid fabricating bibliographic details. The Hebbar et al. [1] author list is intentionally minimal (`and others`) and should be completed against the arXiv record before any external submission.

References

[1] Sahil Hebbar et al. Sia: Self-improving ai with harness and weight updates. 2026. arXiv:2605.27276.

[2] 0xPlaygrounds and contributors. rig: A rust library for building llm-powered applications. https://github.com/0xPlaygrounds/rig. rig-core crate.

[3] Hugging Face and contributors. Candle: Minimalist ml framework for rust. https://github.com/huggingface/candle.

[4] Bytecode Alliance and contributors. Wasmtime: A fast and secure runtime for webassembly. https://github.com/bytecodealliance/wasmtime.

[5] krypticmouse and contributors. Dsrs (dspy-rs): A rust port of dspy. https://github.com/krypticmouse/DSRs.