AIE-Bench: Benchmarking Agents That Build Agents

Abhishek Mishra [1 2] **Selvam Palanimalai** [1] **Yogendra Manawat** [1] **Samuel Verboomen** [1] **Prannay Hebbar** [1]
Damir Vrabac [1] **Deepak Nathani** [1] **Sumeet Ramesh Motwani** [3] **Kunal Bhatia** [1] **Vignesh Baskaran** [1]

Abstract

We introduce AIE Bench, a benchmark for measuring how well AI agents can build and improve other AI agents. Existing benchmarks evaluate whether an agent can solve tasks. This benchmark aims to measure whether an agent can modify another agent to make it better at those tasks. AIE Bench is built around two roles. A meta-agent proposes modifications, while a target-agent is the system being improved. This setup covers meta-improvement, where one agent improves another, and self-improvement, where an agent improves itself. We instantiate AIE Bench across two task families spanning terminal interaction and tool calling, and we evaluate frontier agentic systems on their ability to drive gains through iterative modification. AIE Bench makes target-agent improvement a measurable and reproducible research target.

1. Introduction

Every manual layer in ML engineering eventually gets automated. Feature engineering was once an art, until deep learning absorbed much of it (LeCun et al., 2015). Hyperparameter tuning, which once required expert intuition, has become a search problem. Prompt engineering is entering the same transition. Methods like GEPA, DSPy, OPRO, and Promptbreeder already treat prompt text as a searchable object rather than a hand-crafted artifact (Agrawal et al., 2026; Khattab et al., 2024; Yang et al., 2024a; Fernando et al., 2024). When a layer starts delivering real gains, the dominant approach shifts from manual design to automated search, and so far each such transition has delivered sustained gains.

Agentic engineering is a natural next candidate for this transition. The scaffolds, tool loops, and workflow structures underlying today's best agent systems are a primary source of performance gains, yet this layer remains almost entirely hand-designed (Yang et al., 2024b; Putta et al., 2024; Wang et al., 2025). The same shift is beginning to reach agent scaffolds, but measuring it requires a benchmark that separates real scaffold improvement from changes in evaluation conditions.

We frame this problem around two roles. A *target-agent* is the agent system being improved. Its prompts, code, and workflow logic are the object of optimization. A *meta-agent* is the system responsible for proposing improvements. Given an unfamiliar task regime, it must diagnose which components of the target-agent matter, propose modifications, and produce a better version without a human redesigning the scaffold for each new setting. How capable current models are at this closed-loop improvement, and how quickly that capability is advancing, are open empirical questions.

Prior work such as DGM, Hyperagents, and Meta-Harness (Zhang et al., 2025a; 2026; Lee et al., 2026) demonstrates that agents can self-improve or meta-improve, but these efforts do not focus on benchmarking the capability itself. To the best of our knowledge, every prominent agent benchmark evaluates a fixed system on a task distribution (Jimenez et al., 2024; Chan et al., 2025). The question we pose is different: *can the system that solves tasks also improve the apparatus it uses to solve them, across qualitatively different task families?*

AIE Bench is specifically designed to answer this question. It evaluates a meta-agent on its ability to improve a target-agent, starting from a fixed baseline. The evaluation framework, datasets, and scoring rules remain fixed throughout. Only the target-agent changes, and each run records the full sequence of modifications and their outcomes.

To illustrate, consider a target-agent that solves terminal tasks by navigating filesystems, editing configurations, and piping outputs between tools. Its baseline version solves 37 of 89 Terminal-Bench tasks. The meta-agent examines the target-agent's code, identifies that its error-recovery logic retries failed commands verbatim instead of adjusting the approach, and patches that behavior. The modified target-agent is re-evaluated on the same 89 tasks under the same scoring

[1]Hexo labs [2]Independent [3]University of Oxford. Correspondence to: Abhishek Mishra <ahm@hexo.ai>.

Accepted to the 1st Workshop on Combining Theory and Benchmarks, CTB@ICML 2026, Seoul, South Korea. Copyright 2026 by the author(s).

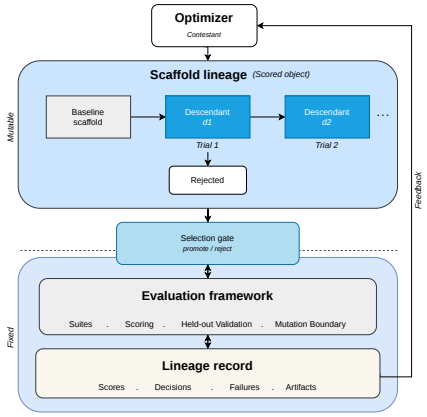

Figure 1. A high-level view of AIE Bench, in which the meta-agent proposes changes to a target-agent. The evaluation framework scores each resulting candidate and records the full sequence of attempts. That record is what the benchmark uses to score the meta-agent.

rules. If it now solves 43, the modification is recorded as a successful improvement and becomes the starting point for the next attempt. The meta-agent repeats this process under a fixed trial budget, and Figure 1 illustrates the resulting evaluation structure. The benchmark scores the meta-agent based on the improvements it produces across this sequence, the target-agent versions are saved, and the evaluation framework, datasets, and scoring rules remain fixed throughout.

Current-generation models can, imperfectly but meaningfully, modify target-agents, reason about tool-chain structure, and determine whether a change has improved behavior (Zhang et al., 2026; Robeyns et al., 2025). Previous-generation models could not do so reliably (Huang et al., 2024). AIE Bench is designed to measure iterative agent improvement as it transitions from an emerging to a practical capability.

The paper makes three contributions to benchmarking target-agent improvement.

- We define AIE Bench, a benchmark for evaluating whether a meta-agent can iteratively improve a target-agent across qualitatively different task regimes.
- We design an evaluation protocol in which the datasets, scoring rules, and infrastructure remain fixed while only the target-agent changes, so that every improvement claim traces back to an actual modification rather than a change in test conditions.
- We instantiate the benchmark across two task families spanning terminal interaction and tool calling, and establish the end-to-end setup for iterative-improvement evaluation.

2. AIE Bench Overview

2.1. What the benchmark measures

Standard agent benchmarks score a fixed system on a task distribution. AIE Bench evaluates whether one agent can improve another. A meta-agent examines a target-agent, proposes a modification, and is scored on whether that modification raises task performance within a fixed trial budget. The benchmark is instantiated across two task families that span complementary agent capabilities, with terminal interaction via Terminal-Bench (Merrill et al., 2026) and structured tool and function calling via BFCL Parity (Patil et al., 2025), as Figure 1 summarizes.

Each run records the starting score, every variant the meta-agent produced, and their outcomes, including failures. The variant-level scores serve as intermediate measurements from which the overall benchmark score for the meta-agent is derived.

2.2. Role definitions

The *evaluation framework* is the fixed infrastructure that executes every experiment. It runs trials, scores target-agents, and records results and remains unchanged across runs.

The *meta-agent* is the system under evaluation, scored on how effectively it improves target-agents. The *baseline target-agent version* is the fixed starting point for a given experiment. The *target-agent* is the system the meta-agent modifies, initialized from that baseline. Each modification produces a *candidate target-agent*, which is launched, scored, and recorded. The *lineage* is the full ordered record of all candidates and the selection decisions that determined their succession. In self-improvement settings, the meta-agent and target-agent may share the same underlying system. The two roles remain distinct in the protocol even when a single model occupies both.

2.3. Individual iterative experiments

An experiment begins from one baseline target-agent version, which serves as the initial active parent. On each trial, the meta-agent receives the current parent and proposes a modification. The evaluation framework validates that the modification falls within the declared boundary, launches the resulting candidate, and scores it.

The candidate's scores and metadata are returned to the meta-agent as feedback. A selection gate then decides whether to reject, record, or promote the candidate. Rejected and recorded candidates stay in the trial ledger but do not become the active parent. Promoted candidates become the parent for the next trial, as Figure 2 shows explicitly. The loop repeats until the trial budget runs out, and the full ordered record is the run's output.

Results from different baselines are comparable only when they share the same mutation boundary, task suites, and scoring rules.

2.4. Mutable and immutable components

The evaluation framework, datasets, scoring logic, and held-out split definitions remain frozen for the duration of an experiment. Only the target-agent is mutable: the meta-agent may modify its code, prompts, and configuration within the declared boundary.

This separation is necessary because without it, apparent gains could arise from modifications to the evaluation infrastructure rather than improvements to the target-agent. By enforcing the mutable/immutable distinction, measured performance is guaranteed to reflect genuine target-agent improvement under fixed evaluation conditions. Section 3.1 formalizes and discusses the enforcement rules.

3. Evaluation Protocol

3.1. Mutation boundary

The mutation boundary formalizes the mutable/immutable split from Section 2. Only the target-agent is mutable, and only within the declared boundary. An experiment may narrow that boundary further through an explicit mutation policy, but nothing may widen it. The evaluation framework, datasets, scoring logic, and prior experiment artifacts are unconditionally out of bounds.

Enforcement occurs before any patch is applied. Any patch that modifies an out-of-boundary path is rejected, and no candidate is produced for that trial. Both the boundary specification and the enforcement decisions are recorded in a machine-readable artifact. Without this discipline, apparent gains could be produced by modifying the benchmark infrastructure rather than improving the target-agent.

3.2. Meta-agent contract

Each run operates under a meta-agent contract that specifies the inputs the meta-agent receives and the outputs it must produce. On each trial, the meta-agent receives the current target-agent state, feedback from previous trials, the mutation boundary, and the remaining budget. It must return a proposed modification with an explicit link to the parent version from which it was derived.

The contract also fixes the per-trial order so that each trial proceeds through propose, validate, evaluate, select, and update. Invalid outputs, such as a missing patch or an out-of-boundary mutation, are classified as contract violations before any performance comparison takes place. Together, these constraints ensure that meta-agent behavior is auditable at the protocol level.

3.3. Optimize and Held-out Splits

Iterative improvement cannot be assessed using the search split alone. AIE Bench therefore distinguishes between two evaluation views: the *optimize* view, which provides the meta-agent with results it may use during search, and the optional *held-out* view, which is never exposed to the agent.

A candidate that fails to improve on the optimize view is rejected. When held-out evaluation is enabled, the protocol can additionally require that promoted candidates avoid regression on the held-out split, or that they demonstrate improvement on it. This design makes overfitting directly observable rather than diagnosed only in post-hoc analysis: the held-out gate actively shapes which candidates pass through the selection gate.

3.4. Budgets

Two budgets govern an iterative experiment and should not be conflated. The *target-agent evaluation budget* controls how many tasks and steps are used when scoring one target-agent version. The *optimizer trial budget* controls how many candidate proposals the meta-agent may attempt over the full experiment.

Every attempted candidate consumes the optimizer trial budget regardless of whether it is promoted, rejected, or invalid. Maintaining this distinction is important because collapsing the two budgets obscures whether gains resulted from more effective search or from increased evaluation effort per candidate.

3.5. Scoring Structure

Scoring in AIE Bench operates at two levels because the benchmark records both candidate performance and the overall search process. Each candidate first produces a scorecard recording its performance on the benchmark suite, and AIE Bench then aggregates those scorecards with promotion decisions, failure counts, and held-out outcomes to produce the meta-agent's overall score.

The primary reported score is the best descendant task accuracy reached within the fixed optimizer trial budget, reported alongside the baseline. The benchmark-level summary also records the best held-out score when held-out evaluation is enabled, promotion count, valid candidate count, and trials consumed by invalid or broken candidates. These quantities constitute the formal basis for comparing meta-agents.

3.6. Failure Classification

Iterative experiments remain interpretable only if each failure is attributed to the component that caused it. AIE Bench

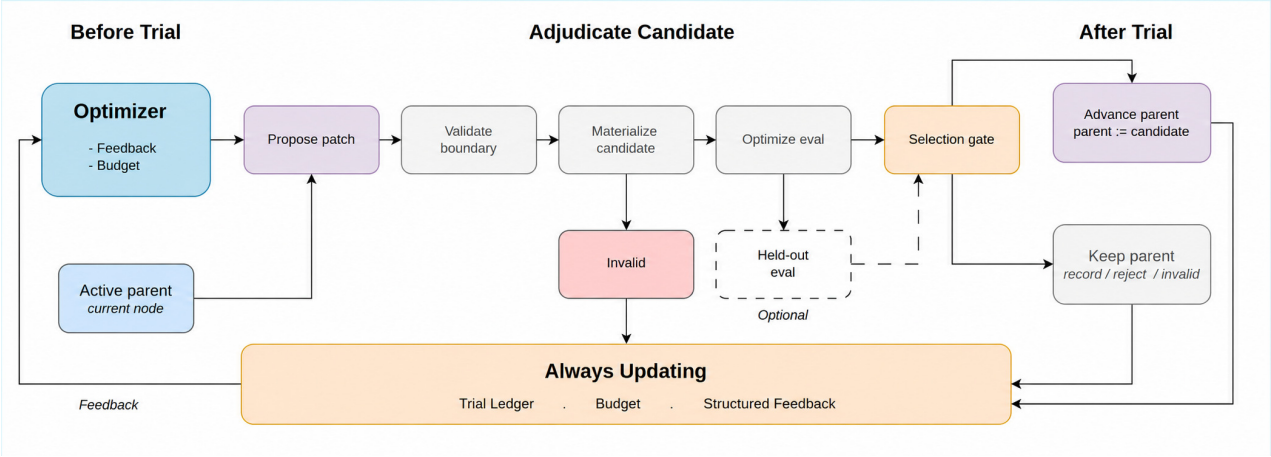

Figure 2. A single trial in the iterative loop, where the meta-agent proposes a candidate from the current active parent. The evaluation framework validates, launches, and scores it. The selection gate then promotes the candidate to active parent or keeps the current one. Every outcome updates the trial ledger and feedback available to the next trial.

therefore separates failures into three classes. A *meta-agent failure* occurs before a valid candidate is produced, for example when the meta-agent violates the contract, emits malformed output, or proposes an out-of-boundary mutation. A *target-agent failure* occurs after a valid candidate is produced but the resulting candidate cannot launch or serve correctly. An *evaluation-framework failure* occurs when infrastructure or scoring breaks outside both the meta-agent and the target-agent.

An out-of-boundary patch is evidence about the meta-agent. A broken candidate is evidence about the target-agent mutation. A host-side infrastructure failure is evidence about the runtime environment. Collapsing these into a single failure category would obscure the distinctions necessary to interpret iterative-improvement results.

3.7. Required artifacts

Claims about iterative improvement require thorough documentation to be verifiable. Each official AIE Bench run therefore produces an artifact bundle sufficient to reconstruct the entire experiment: baseline identity, every candidate and its parent linkage, mutation-boundary policy, task-level results, scorecards, lineage records, and logs for diagnosing failures. When held-out evaluation is enabled, the bundle additionally includes held-out outputs and promotion decisions. The complete file layout is specified in the appendix.

4. Benchmark Instantiation and Experimental Setup

4.1. Reference target-agent family

The reference target-agent family is `mini-swe-agent` (Lieret & Jimenez, 2025). Every official run starts from one pinned git commit of `mini-swe-agent` as the baseline. The meta-agent's modifications are applied within that codebase, and baseline and descendant versions are evaluated through the same interface. The benchmark is agnostic to which parts of the target-agent the meta-agent chooses to modify, as long as the changes stay within the declared boundary. The target-agent inherits the meta-agent's backing model, so a Claude Code lane evaluates `mini-swe-agent` running on the same model that powers the meta-agent. This means baseline performance varies across meta-agent configurations.

4.2. Benchmark domains

We instantiate the benchmark across two benchmark sets chosen to span materially different task regimes. Together they cover long-horizon terminal interaction and structured tool calling. The matrix varies task surface while keeping a shared iterative-search protocol, mutation discipline, and scoring rules.

Benchmark	Version	Tasks	Capability
Terminal-Bench	2.0	89	Terminal-agent execution
BFCL Parity	1.0	123	Function / tool calling

Table 1. Experiment matrix. Each benchmark is evaluated under the same iterative-search protocol, mutation discipline, and scoring contract.

The matrix is intentionally heterogeneous in task surface and homogeneous in protocol. Terminal-Bench tests long-horizon command-loop interaction, while BFCL Parity tests structured tool and function calling. Both pass through the same iterative-search protocol and scoring layer. A meta-agent that improves the target-agent across these two

regimes provides initial evidence beyond a single interaction style.

Iterative execution path. Official AIE Bench runs follow the iterative trial loop described in Section 2. The runner fixes one pinned baseline, and the meta-agent proposes candidates against the current active parent until the trial budget is exhausted. Mutation boundary, selection gate, and scoring rules are fixed for the full run.

Contestants and baselines. The benchmarked contestant is the meta-agent configuration for a run. AIE Bench compares arbitrary meta-agent choices under the same iterative-search contract, including different model combinations. The baseline target-agent version is the starting point, not the contestant. Baseline and descendant scores are task-level measurements that AIE Bench uses to evaluate the meta-agent.

Suites, splits, and run budgets. Official runs use the two benchmark sets in Table 1, each with a separate iterative trial budget. The framework supports optimize-versus-held-out evaluation, where held-out evaluation can either record candidate behavior passively or block promotion when held-out performance regresses.

Runtime and artifact inspection. Every candidate is evaluated through the same interface, ensuring that baseline and descendant scores are directly comparable. Each run yields human-readable summaries alongside the full artifact bundle, making claims about iterative improvement auditable after the fact.

5. Results

We evaluate Claude Code with Claude Sonnet 4.6, Gemini CLI with Gemini 3.1 Pro, and Codex/GPT-5.4-high across both benchmark sets with a maximum optimizer trial budget of 8. All three lanes use the same target-agent family, mutation boundary, and scoring rules. The target-agent, `mini-swe-agent`, uses the same backing model as the corresponding meta-agent, so different lanes produce different baselines. The main matrix uses optimize-only selection, and BFCL controls measure held-out behavior and best-of-8 target-agent variance.

Table 2 summarizes what varies and what stays fixed across the reported runs and controls.

Fixed	Value
Target-agent codebase	`mini-swe-agent` (pinned commit)
Mutation boundary	Target-agent codebase only
Scoring	Accuracy (passed / total)
Held-out evaluation	Disabled in main matrix; enabled in BFCL control
Varied	**Levels**
Meta-agent	Claude Code (Sonnet 4.6), Gemini CLI (3.1 Pro), Codex/GPT-5.4-high
Target-agent model	Same as corresponding meta-agent
Trial budget	8
Benchmark set	Terminal-Bench, BFCL Parity

Table 2. Experimental setup for the reported runs. Each run shares the same target-agent family, mutation boundary, and scoring rules. The benchmark set and meta-agent configuration vary directly, and the target-agent model follows from the meta-agent choice.

5.1. Main results

Table 3 reports the best descendant score achieved by each meta-agent on each benchmark set under a maximum optimizer trial budget of 8.

Meta-agent	Benchmark	Baseline	Best descendant
Claude Code	Terminal-Bench	48.3	52.8
	BFCL Parity	77.2	82.2
Gemini CLI	Terminal-Bench	57.3	61.7
	BFCL Parity	81.8	87.3
Codex/GPT-5.4-high	Terminal-Bench	56.2	62.0
	BFCL Parity	78.1	86.2

Table 3. Best descendant accuracy (%) by meta-agent and benchmark under a maximum optimizer trial budget of 8. Baselines differ because the target-agent inherits the meta-agent's backing model.

All reported meta-agent lanes improve the target-agent under the fixed benchmark contract. Claude Code raises Terminal-Bench accuracy from 48.3% to 52.8% (+4.5pp) and BFCL Parity from 77.2% to 82.2% (+5.0pp). Gemini CLI raises Terminal-Bench from 57.3% to 61.7% (+4.4pp) and BFCL Parity from 81.8% to 87.3% (+5.5pp). Codex/GPT-5.4-high raises Terminal-Bench from 56.2% to 62.0% (+5.8pp) and BFCL Parity from 78.1% to 86.2% (+8.1pp). These results show target-agent improvement under the same benchmark contract across three frontier meta-agent configurations.

Table 4 reports BFCL controls that separate improvement from two common confounds. Held-out evaluation checks whether gains transfer to tasks not exposed during search. Best-of-8 target-agent runs check how much lift comes from rerunning the unmodified target-agent and selecting the best sample.

Control	Outcome
Gemini held-out BFCL	Baseline 51/62 = 0.8226; candidate 52/62 = 0.8387
Gemini BFCL best-of-8	Best single replicate 105/123 = 0.8537; per-task best-of-8 107/123 = 0.8699
Sonnet BFCL best-of-8	Best official scorecard 95/123 = 0.7724; per-task diagnostic 98/123 = 0.7967

Table 4. BFCL controls for held-out behavior and target-agent rerun variance.

Of the invalid trials, the majority are meta-agent failures (out-of-boundary patches or malformed proposals) rather than target-agent failures (candidates that launch but crash). This pattern holds across the reported lanes and indicates that the mutation boundary is the primary constraint, not target-agent fragility.

6. Reliability, Validity, and Failure Analysis

The reported runs demonstrate that AIE Bench measures iterative target-agent improvement under a fixed benchmark contract rather than changes in the evaluation setup. Three properties are central to that result.

First, mutation boundaries were enforced throughout the reported runs. Every patch was validated before application, confirming that the boundary is active rather than vacuous.

Second, baseline pinning held throughout the reported runs. The meta-agent lanes start from the same `mini-swe-agent` family and fixed run configuration. Because the target-agent inherits the meta-agent's model, the lanes have different baselines, but within each lane the baseline measurements were stable, confirming that the evaluation framework produced stable measurements of the unmodified target-agent.

Third, failure attribution separated meta-agent errors from target-agent errors from infrastructure errors. Most invalid trials were meta-agent failures, primarily out-of-boundary patches. A smaller share came from broken target-agent descendants or transient infrastructure issues that the evaluation framework flagged and excluded from scoring.

This matrix covers one target-agent family, one baseline commit, and two benchmark sets spanning distinct task regimes. That scope establishes the benchmark object and the reported operating point. The main matrix covers both benchmark sets, while BFCL provides held-out and best-of-8 controls under the same target-agent family. Further target-agent families, benchmark sets, and cross-model lane comparisons extend the same contract to a broader regime.

7. Related Work

7.1. Static agent benchmarks

Recent agent benchmarks have broadened evaluation beyond single-shot language modeling, but they still score a fixed system on a fixed task distribution. Terminal-Bench measures command-line agents on hard terminal tasks (Merrill et al., 2026). BFCL evaluates structured tool and function calling (Patil et al., 2025). GAIA targets general assistants with reasoning, tool use, and web interaction (Mialon et al., 2024). τ-bench focuses on tool-agent-user interaction under domain policies (Yao et al., 2025). MLE-bench evaluates machine learning engineering ability (Chan et al., 2025), and RefactorBench probes stateful reasoning over multi-file code transformations (Gautam et al., 2025). These benchmarks are valuable because they stress realistic downstream behavior, but the benchmarked object remains a fixed agent or harness. AIE Bench instead evaluates the outer-loop meta-agent through the target-agent lineage it produces.

7.2. Prompt, workflow, and scaffold optimization

The strongest methodological precedents for AIE Bench are systems that turn prompts, workflows, and harness code into optimization targets. Early work such as OPRO and DSPy framed prompt and pipeline optimization as an outer-loop search problem (Yang et al., 2024a; Khattab et al., 2024). More recent systems make that outer loop more explicit. GEPA evolves prompts reflectively (Agrawal et al., 2026). ADAS and AFlow search over agentic workflows (Hu et al., 2025; Zhang et al., 2025b). Meta-Harness performs end-to-end optimization of harness code (Lee et al., 2026). These papers show that prompts, workflows, and harnesses can all be optimized. AIE Bench contributes the complementary benchmark protocol for comparing such outer-loop optimizers under fixed mutation boundaries, budgets, and optional held-out evaluation.

7.3. Self-improving agent systems

Recent work has started to study self-improving agent systems directly. Darwin Gödel Machine studies open-ended evolution of self-improving agents (Zhang et al., 2025a). Hyperagents explores agent improvement at the system level (Zhang et al., 2026). SE-Bench benchmarks self-evolution in a narrower setting centered on knowledge internalization (Yuan et al., 2026). AgentBreeder studies evolutionary self-improvement over multi-agent scaffolds (Rosser & Foerster, 2025). Together these papers make agent improvement an empirical object. AIE Bench differs in its evaluation target: it does not introduce one improvement method, but instead fixes the contract needed to compare arbitrary meta-agents on their ability to improve a target-agent across benchmark regimes.

8. Conclusion

AIE Bench isolates a capability that standard agent benchmarks mostly leave implicit: whether a meta-agent can improve a target-agent under fixed evaluation conditions. By fixing the datasets, scoring rules, and mutation boundary while recording the full target-agent lineage, the benchmark turns iterative target-agent improvement into a measurable and auditable object.

In this setting, multiple meta-agent configurations improve `mini-swe-agent` on Terminal-Bench and BFCL Parity under a maximum optimizer trial budget of 8. This establishes a measured operating point for target-agent improvement with current-generation models. The gains are visible under a constrained search budget, and the full run record is preserved.

AIE Bench provides the benchmark object and protocol needed to compare systems that improve agents under a common contract, and it creates a stable evaluation target for stronger model families, larger-budget search, held-out selection, and broader benchmark coverage.

Impact Statement

AIE Bench is intended to make iterative agent improvement measurable, auditable, and reproducible. The benchmark separates target-agent changes from evaluation-framework changes, records lineage and failure evidence, and supports held-out checks that expose overfitting rather than hiding it. The same capability could contribute to more capable automated agents, so the benchmark emphasizes bounded mutation, fixed scoring rules, and artifact inspection instead of unconstrained self-modification.

References

Agrawal, L. A., Tan, S., Soylu, D., Ziems, N., Khare, R., Opsahl-Ong, K., Singhvi, A., Shandilya, H., Ryan, M. J., Jiang, M., Potts, C., Sen, K., Dimakis, A. G., Stoica, I., Klein, D., Zaharia, M., and Khattab, O. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=RQm2KQTM5r.

Chan, J. S., Chowdhury, N., Jaffe, O., Aung, J., Sherburn, D., Mays, E., Starace, G., Liu, K., Maksin, L., Patwardhan, T., Madry, A., and Weng, L. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=6s5uXNWGIh.

Fernando, C., Banarse, D. S., Michalewski, H., Osindero, S., and Rocktäschel, T. Promptbreeder: Self-referential self-improvement via prompt evolution. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=9ZxnPZGmPU.

Gautam, D., Garg, S., Jang, J., Sundaresan, N., and Zilouchian Moghaddam, R. Refactorbench: Evaluating stateful reasoning in language agents through code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=NiNIthntx7.

Hu, S., Lu, C., and Clune, J. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=t9U3LW7JVX.

Huang, J., Chen, X., Mishra, S., Zheng, H. S., Yu, A. W., Song, X., and Zhou, D. Large language models cannot self-correct reasoning yet. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=IkmD3fKBPQ.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. R. 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.

Khattab, O., Singhvi, A., Maheshwari, P., Zhang, Z., Santhanam, K., Vardhamanan, S., Haq, S., Sharma, A., Joshi, T. T., Moazam, H., Miller, H., Zaharia, M., and Potts, C. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=sY5N0zY5Od.

LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *Nature*, 521(7553):436–444, 2015. doi: 10.1038/nature14539.

Lee, Y., Nair, R., Zhang, Q., Lee, K., Khattab, O., and Finn, C. Meta-harness: End-to-end optimization of model harnesses, 2026. URL https://arxiv.org/abs/2603.28052.

Lieret, K. and Jimenez, C. E. mini-swe-agent. Software, 2025. URL https://github.com/SWE-agent/mini-swe-agent.

Merrill, M. A., Shaw, A. G., Carlini, N., Li, B., Raj, H., Bercovich, I., Shi, L., Shin, J. Y., Walshe, T., Buchanan, E. K., Shen, J., Ye, G., Lin, H., Poulos, J., Wang,

M., Nezhurina, M., Jitsev, J., Lu, D., Mastromichalakis, O. M., Xu, Z., Chen, Z., et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

Mialon, G., Fourrier, C., Wolf, T., LeCun, Y., and Scialom, T. GAIA: A benchmark for general AI assistants. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=fibxvahvs3.

Patil, S. G., Mao, H., Yan, F., Ji, C. C.-J., Suresh, V., Stoica, I., and Gonzalez, J. E. The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 48371–48392. PMLR, 2025. URL https://proceedings.mlr.press/v267/patil25a.html.

Putta, P., Mills, E., Garg, N., Motwani, S., Finn, C., Garg, D., and Rafailov, R. Agent q: Advanced reasoning and learning for autonomous ai agents, 2024. URL https://arxiv.org/abs/2408.07199.

Robeyns, M., Szummer, M., and Aitchison, L. A self-improving coding agent, 2025. URL https://arxiv.org/abs/2504.15228.

Rosser, J. and Foerster, J. N. Agentbreeder: Mitigating the AI safety risks of multi-agent scaffolds via self-improvement. In *Advances in Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=mlU9KqdZUS.

Wang, X., Li, B., Song, Y., Xu, F. F., Tang, X., Zhuge, M., Pan, J., Song, Y., Li, B., Singh, J., Tran, H. H., Li, F., Ma, R., Zheng, M., Qian, B., Shao, Y., Muennighoff, N., Zhang, Y., Hui, B., Lin, J., Brennan, R., Peng, H., Ji, H., and Neubig, G. Openhands: An open platform for ai software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

Yang, C., Wang, X., Lu, Y., Liu, H., Le, Q. V., Zhou, D., and Chen, X. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2024a. URL https://openreview.net/forum?id=Bb4VGOWELI.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K. R., and Press, O. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024b. URL https://openreview.net/forum?id=mXpq6ut8J3.

Yao, S., Shinn, N., Razavi, P., and Narasimhan, K. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=roNSXZpUDN.

Yuan, J., Jin, T., Chen, W., Liu, Z., Liu, Z., and Sun, M. SE-bench: Benchmarking self-evolution with knowledge internalization. *arXiv preprint arXiv:2602.04811*, 2026. URL https://arxiv.org/abs/2602.04811.

Zhang, J., Hu, S., Lu, C., Lange, R., and Clune, J. Darwin gödel machine: Open-ended evolution of self-improving agents, 2025a. URL https://arxiv.org/abs/2505.22954.

Zhang, J., Xiang, J., Yu, Z., Teng, F., Chen, X., Chen, J., Zhuge, M., Cheng, X., Hong, S., Wang, J., Zheng, B., Liu, B., Luo, Y., and Wu, C. AFlow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL https://openreview.net/forum?id=z5uVAKwmjf.

Zhang, J., Zhao, B., Yang, W., Foerster, J., Clune, J., Jiang, M., Devlin, S., and Shavrina, T. Hyperagents, 2026. URL https://arxiv.org/abs/2603.19461.

A. Discussion and Limitations

The reported results establish that AIE Bench detects iterative target-agent improvement under a fixed benchmark contract. The reported meta-agent configurations produce better descendant versions of `mini-swe-agent` on Terminal-Bench and BFCL Parity within a maximum optimizer trial budget of 8. Iterative target-agent improvement is therefore already measurable in a setting that fixes the datasets, scoring rules, and mutation boundary while preserving the full run record.

The results define a compact operating point for current-generation models. The matrix uses a single target-agent family and a small-budget single-lineage search regime. Within that setting, the meta-agent produces measurable improvements, while the dominant source of invalid trials remains bounded-search mistakes such as malformed or out-of-boundary proposals. This is exactly the distinction the benchmark is designed to expose: search failures remain visible without obscuring genuine target-agent outcomes.

The small-budget matrix isolates an iterative-search regime. Larger trial budgets, population-based search, crossover-style evolutionary operators, and stronger held-out selection

policies all fit the same benchmark contract. AIE Bench therefore provides a stable way to compare stronger outer-loop search procedures without changing the benchmark object itself.

Two scope conditions shape this operating point. First, the matrix covers one target-agent family and two benchmark sets, which is sufficient to establish benchmarked iterative improvement across terminal interaction and tool calling. Second, the target-agent inherits the meta-agent's backing model, so lane differences reflect both optimizer quality and target-agent quality. Cross-model runs that decouple the optimizing model from the target-agent model would sharpen attribution without changing the benchmark protocol.

Taken together, the results establish the benchmark object, the protocol, and a first measured operating point for iterative target-agent improvement. The same contract can evaluate larger search budgets, broader benchmark coverage, and stronger model families.

B. Additional Benchmark and Protocol Examples

B.1. Example iterative experiment manifests

The manifest is the canonical description of one iterative run. It fixes the meta-agent lane, the target-agent family, the optimize suite, the budgets, the mutation policy, and the output location. An optimize-only pattern is:

```
{
  "mode": "recursive_search",
  "sut": {
    "sut_config": "../suts/claude_code_haiku_sut.json"
  },
  "sum": {},
  "baseline": {
    "id": "baseline",
    "base_commit": "<pinned-mini-swe-agent-commit>"
  },
  "evaluation": {
    "optimize": {
      "suite_config": "../terminalbench_smoke_5.json",
      "split_id": "optimize"
    },
    "held_out": {
      "required": false,
      "selection_gate": "record_only"
    },
    "trial_budget": 5,
    "budget_step_timeout_s": 1200
  },
  "mutation_policy": {"source": "sum_profile"}
}
```

Held-out-gated iterative search uses the same manifest schema. A minimal held-out-enabled fragment adds a held-out suite and a promotion gate:

```
{
  "mode": "recursive_search",
  "sut": {"sut_config": "sut.json"},
  "sum": {},
  "evaluation": {
    "optimize": {"suite_config": "suite.json"},
    "held_out": {
      "required": true,
      "suite_config": "held-out.json",
      "selection_gate": "must_not_regress"
    },
    "trial_budget": 2
  }
}
```

The first fragment shows the optimize-only pattern, and the second shows the corresponding held-out extension. In both cases, the manifest remains the canonical description of one iterative experiment instance.

B.2. Example bundle layout

Each official iterative run writes a top-level bundle plus one directory per attempted trial. The top-level structure is:

```
experiments/recursive-runs/
  gemini-cli-terminalbench-smoke-1-live-run/
  source_manifest.json
  resolved_manifest.json
  sut_contract.json
  sum_boundary.json
  report.md
  trial_ledger.jsonl
  lineage_summary.json
  promotion_history.json
  final_feedback_packet.json
  execution_trace.txt
  trials/
```

One trial directory contains the optimizer inputs and outputs, the emitted patch, the selection decision, and both structured and raw evaluation artifacts:

```
trials/002/
  optimizer_input.json
  optimizer_instruction.md
  optimizer_output.json
  optimizer_trace.jsonl
  candidate.patch
  feedback_packet.json
  selection_decision.json
  trial_result.json
  artifacts/
    baselines/
    candidates/
    comparisons/
    candidate-patches/
    configs/
  raw/
    baselines/
    candidates/
```

The natural read order starts with `execution_trace.txt`, which provides a stitched narrative of the full run. `report.md` and `trial_ledger.jsonl` then expose the budget, decision sequence, and final outcome. Finally, `trials/<n>/selection_decision.json` and `trials/<n>/candidate.patch` provide the exact evidence for one iterative step.

C. Implementation Details

C.1. PACT and server lifecycle

All evaluation-facing systems in AIE Bench implement the same common interface: a PACT server with `/describe`, `/act`, `/reset`, and `/health` endpoints. The benchmark speaks only to that interface. As a result, benchmark suites do not require benchmark-specific logic for each model or harness lane, and model-facing servers do not need to know which suite is calling them.

A launchable system is defined by a system profile. The profile specifies the server command, working directory, environment variables, host, startup timeout, and the health and describe paths. AIE Bench uses such profiles both for meta-agent-facing lanes such as Gemini CLI and for target-agent-facing scaffold servers such as `mini-swe-agent`. The benchmark resolves a profile, launches the corresponding PACT server, waits for it to become healthy, and only then starts task evaluation or iterative optimization.

The same interface also underlies the Harbor-backed execution path. In that setting, the runtime starts a benchmark-owned shell service inside the container, exposes the PACT actions to the benchmark, and records the resulting interaction traces. This keeps the evaluation contract stable even when the underlying launch path differs across lanes.

C.2. Candidate worktree materialization

Each iterative trial runs inside an isolated worktree materialized from the current active parent. The optimizer session creates a detached git worktree under `trials/<n>/.optimizer-worktree`, rooted at either the baseline commit or the currently promoted parent node. The worktree root is already the scaffold repository itself, so all optimizer paths are scaffold-relative rather than repo-relative.

Before any candidate patch is applied, the benchmark validates the patch against the declared mutation boundary. The validator rejects changes to immutable git metadata, symlinks, and any path outside the allowed mutation roots. Only after this check passes does the benchmark apply the patch and launch the resulting candidate target-agent instance for evaluation.

The optimizer worktree also receives a small amount of benchmark-owned context. The session materializes `.aie_bench_recursive/optimizer_context.md`, a localized `mutation_boundary.json`, and auxiliary artifact references so the meta-agent can reason about the current trial without modifying benchmark-owned state. After evaluation, the benchmark stages the resulting patch, writes the trial artifacts, and removes the temporary worktree.

C.3. Config layering

AIE Bench uses four configuration layers, each with a distinct role.

First, the suite config defines what benchmark slice is evaluated. For example, `configs/terminalbench_2_smoke.json` names the benchmark, dataset version, task count, concurrency, and backend. This layer defines the task regime, not the evaluated system.

Second, the system profile defines how one evaluation-facing system is launched. A profile such as `configs/gemini_cli_pro_preview_profile.json` specifies the PACT launch command, host, timeout, and environment variables for that lane. The corresponding target-agent profile, such as `configs/mini_swe_agent_profile.json`, additionally declares the mutable root and allowed mutation paths for iterative experiments.

Third, the SUT config names the benchmarked meta-agent lane and binds it to both a launch profile and a target-agent family. For example, `configs/suts/gemini_cli_pro_preview_sut.json` declares the optimizer contract, tool menu, timeout, launch profile, and SUM profile used by that lane.

Fourth, the experiment manifest instantiates one concrete iterative run. It fixes the baseline commit, optimize suite, optional held-out suite, trial budget, mutation policy, and output directory. The resolved manifest written into each bundle turns these relative references into a fully explicit run contract.

D. Additional Results

D.1. Per-domain breakdowns

Table 5 restates the benchmark-specific deltas from the main results. Both reported domains improve across all reported lanes, and the BFCL Parity deltas are larger in each case.

Meta-agent	Benchmark	Baseline	Best descendant	Delta
Claude Code	Terminal-Bench	48.3	52.8	+4.5
Claude Code	BFCL Parity	77.2	82.2	+5.0
Gemini CLI	Terminal-Bench	57.3	61.7	+4.4
Gemini CLI	BFCL Parity	81.8	87.3	+5.5
Codex/GPT-5.4-high	Terminal-Bench	56.2	62.0	+5.8
Codex/GPT-5.4-high	BFCL Parity	78.1	86.2	+8.1

Table 5. Per-benchmark deltas derived from the main results. All reported lanes improve on both domains, and the BFCL Parity deltas are larger in each case.

These domain-level deltas show that the reported signal is present in both task regimes rather than being confined to only one interaction style.

D.2. Held-out selection behavior

The main matrix is optimize-only, while held-out selection is part of the benchmark contract and uses the same manifest and artifact pipeline.

When held-out evaluation is enabled, the manifest sets `evaluation.held_out.required = true`, supplies a held-out suite config, and selects a gate such as `must_not_regress`. The benchmark then writes both optimize and held-out fields into the selection decision. Under this regime, optimize-side improvement alone is not enough for promotion if the held-out aggregate regresses.

This design keeps held-out evaluation inside the benchmark contract rather than treating it as a separate evaluation mode. The same iterative loop, artifact structure, and decision record continue to apply when held-out gating is enabled.

E. Failure Case Studies

E.1. Host-side volatility examples

Two concrete examples clarify the distinction between host-side volatility and contestant behavior. One is `chess-best-move`, where Docker build failed during `apt install -y python3-pip` because `ports.ubuntu.com` timed out. Another is `swe-bench-astropy-1`, where `git clone https://github.com/astropy/astropy.git` timed out inside the container build. In both cases, the failure arises from host or network conditions rather than from the behavior of the evaluated system.

The bundle `gemini-cli-terminalbench-smoke-1-auto-trace` shows the same distinction in bundle form. Its trial ledger records `evaluation_framework_failure`, and its execution trace stops with `trial_1_framework_failure`. Host or infrastructure contamination therefore remains visible in the bundle, but it stays separate from benchmarked evidence about the meta-agent or target-agent.

E.2. True task and descendant failures

The live bundle `gemini-cli-terminalbench-smoke-1-live-run` provides the complementary case. In trial 2, both the baseline and the descendant candidate reach normal benchmark execution on the Terminal-Bench task `gpt2-codegolf`. The recorded failure mode is `task_failure`, not `evaluation_framework_failure`. The candidate patch is present, the optimizer rationale is recorded, the selection decision is written, and the task runner returns a task-level failure after the agent emits an invalid action format.

This distinction matters for reading benchmark results. A host-side failure says the environment never gave the contestant a fair shot. A task failure says the benchmarked system was actually executed and failed on the task under the benchmark contract. The live bundle therefore illustrates the kind of evidence that should count toward iterative-search evaluation even when the resulting score improvement is zero.

F. Reproducibility Details

F.1. Environment and installation

A standard Python-and-Docker environment is sufficient to reproduce AIE Bench. The key requirements are Python 3.13 or later, `git`, a working Docker daemon, Docker Compose support, and an initialized `mini-swe-agent` submodule. Benchmark runs also require credentials for the chosen meta-agent lane and, for Terminal-Bench, the pinned dataset cache prepared under the same user account that will execute the run.

A minimal installation path is:

```
python3.13 -m venv .venv
source .venv/bin/activate
python -m pip install --upgrade pip
python -m pip install -e ".[operator,test]"
git submodule update --init --recursive
python -m pip install -e mini-swe-agent
python scripts/setup_bench.py terminalbench
python scripts/setup_bench.py --doctor
python -m pytest tests -q
```

The `setup_bench.py --doctor` preflight checks the shared Harbor-backed prerequisites used by the reported experimentation matrix, including Docker availability and the Terminal-Bench cache.

F.2. Run commands

The benchmark uses one manifest-driven entrypoint for iterative-search runs:

```
python -m aie_bench.recursive_experiment \
  --manifest <manifest.json>
```

The same entrypoint covers both smoke manifests and longer iterative presets. A launcher also wraps the tracked full-run presets directly:

```
python scripts/run_recursive_full.py gemini-t8 \
  terminalbench
python scripts/run_recursive_full.py gemini-t8 bfcl
python scripts/run_recursive_full.py sonnet-t8 \
  terminalbench
```

These commands preserve the benchmark contract inside the manifest itself. The baseline commit, suite config, trial budget, mutation policy, and output directory remain explicit and reconstructable in the resulting bundle.