TOWARD GENERALIST AUTONOMOUS RESEARCH VIA HYPOTHESIS-TREE REFINEMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Autonomous research requires agents to make sustained progress over long horizons by continuously refining hypotheses through experimental feedback. Although recent large language model agents can edit code, invoke tools, and execute experiments for extended periods, longer execution alone does not make research cumulative. Without an explicit research state, experiments remain isolated attempts, preventing evidence from being accumulated and reused to guide future exploration. We formulate this setting as *Autonomous Optimization* (AO), in which an agent iteratively improves an artifact under a fixed objective without step-level human supervision, and introduce `Arbor`, a general framework for AO. At the core of `Arbor` is *Hypothesis Tree Refinement* (HTR), an iterative algorithm that maintains a persistent hypothesis tree as the research state and continuously refines it as isolated experiments return evidence. By grounding each hypothesis in its artifact and evidence, and propagating findings to guide subsequent experiments, HTR connects local outcomes to cumulative hypothesis refinement. The tree serves as the search frontier, long-term memory, and audit trail. Across six real research tasks spanning model training, harness engineering, and data synthesis, `Arbor` improves the admission metric on every task under the same time budget, yielding roughly twice the gain of Codex and Claude Code. On MLE-Bench Lite it reaches a 77.3% gold rate, 24 points above the strongest prior system in our comparison. After optimization on BrowseComp, the frozen search harness transfers to unseen HLE and DeepSearchQA tasks, improving both without further optimization. Ablations and research traces support the value of connecting hypothesis management with evidence propagation for cumulative autonomous research.

1 INTRODUCTION

Scientific research is a central form of long-horizon human intelligence (Chen et al., 2026a; Tie et al., 2025). Sustaining progress across uncertain hypotheses, costly experiments, and failed attempts requires an evolving understanding of what should be tried next (Hu et al., 2025). Recent LLM agents can edit code, call tools, retrieve information, and run experiments for extended periods (Yao et al., 2023; Schick et al., 2023; Qin et al., 2024), while Codex (OpenAI, 2025), Claude Code (Anthropic, 2025), and OpenHands (Wang et al., 2025) make sustained progress in real codebases. Yet longer execution alone does not guarantee research progress: the open challenge is maintaining a research state that turns local attempts into cumulative hypothesis refinement and artifact improvement.

We formalize this problem as *Autonomous Optimization* (AO), in which an agent improves an initial artifact under a research objective through experimental feedback without step-level human supervision. Research feedback is delayed, experiments can be expensive, and failed attempts often contain information that should guide later search. As the horizon grows, effective AO requires a persistent research state that records what has been tried, what evidence was obtained, and how each result changes the space of future hypotheses.

Recent systems address parts of this problem through reusable skills and evolving playbooks (Wang et al., 2023; Zhang et al., 2025b), code search (Jiang et al., 2025), and agentic trees of research experiments (Yamada et al., 2025). For AO, the central question is how accumulated experience changes what should be tested next. A failed implementation may expose a limitation of one variant without invalidating the broader direction; its evidence should therefore inform related hypotheses

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

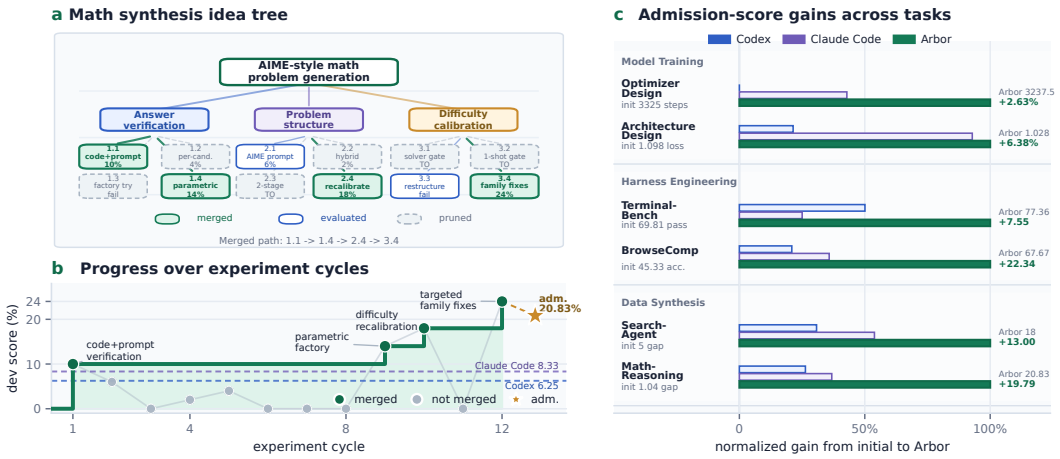

Figure 1: `Arbor` at a glance. **(a)** Hypothesis tree and **(b)** development score from one Math-Reasoning Data Synthesis run. **(c)** Normalized admission gains across all tasks.

even when its code is discarded. This calls for a research state that connects hypotheses at different levels of abstraction and carries experimental findings into subsequent frontier decisions.

We introduce **Arbor**, a general framework and open-source research system for AO. `Arbor` separates autonomous research into a long-lived coordinator and short-lived executors. The coordinator owns the global research state and decides how the search frontier should evolve, while each executor tests one hypothesis in an isolated worktree and returns structured evidence. `Arbor` makes this two-level process cumulative through Hypothesis Tree Refinement (HTR): a persistent tree in which each node binds a hypothesis, the artifact version that realizes it, the experimental evidence it produces, and the distilled insight that should shape later decisions. When executor results return, `Arbor` writes evidence back to the executed nodes, abstracts local findings upward, and uses the updated tree to decide which directions to expand, prune, or merge, promoting a candidate to the current best only when it improves an admission evaluation. The tree therefore acts as the operational research state: it is simultaneously the search frontier, the memory of past attempts, and the audit trail for artifact improvement. Its hierarchy distinguishes refinement of an implementation from revision of a research direction, while upward insight propagation makes local findings available to later branches. Persistent summaries preserve this context across executor lifetimes, connecting experience reuse directly to experiment selection. Figure 1 shows this process in practice: a hypothesis tree grown on one task, the development score it drives over the run, and the resulting admission gains across all tasks.

To evaluate `Arbor`, we construct six AO tasks from real research settings across model training (Jordan & contributors, 2025; Karpathy, 2026), harness engineering (Merrill et al., 2026; Wei et al., 2025), and data synthesis. Each task specifies an initial artifact, a natural-language objective, a task-native metric, and development/admission evaluation for local experiments and artifact selection. `Arbor` improves the admission metric on all six tasks under a shared 48-hour wall-clock cap, yielding roughly twice the gain of Codex and Claude Code, and on BrowseComp raising accuracy from 45.33 to 67.67 on 300 held-out questions. On MLE-Bench Lite (Chan et al., 2024), `Arbor` reaches a 77.27% gold rate with GPT-5.5, 24 points above the strongest prior system in our comparison. To evaluate generalization beyond the optimization feedback, we freeze the BrowseComp-optimized harness and evaluate it on HLE (Phan et al., 2025) and DeepSearchQA (Gupta et al., 2026), improving both without further optimization. Ablations and hypothesis traces further connect these outcomes to HTR's organization of experiments and reusable evidence.

Our contributions are summarized as follows:

- We formulate *Autonomous Optimization* (AO), where an agent iteratively improves an artifact under a fixed objective without step-level supervision, and construct six AO tasks across three categories of real research (model training, harness engineering, and data synthesis).

- We introduce **Arbor**, a general open-source AO framework built on *Hypothesis Tree Refinement* (HTR), which pairs a persistent coordinator with isolated executors so that hypotheses, artifact versions, evidence, and distilled insights accumulate into an auditable research state.
- We demonstrate strong optimization results across six AO tasks and MLE-Bench Lite, transfer of a frozen search harness to unseen tasks, and the contribution of hypothesis management and insight propagation through ablations and trace analysis.

2 Related Work

Autonomous Research Agents. LLM-based research automation first appeared as end-to-end pipelines connecting ideation, implementation, and analysis, such as The AI Scientist (Lu et al., 2024) and Agent Laboratory (Schmidgall et al., 2025), and later made the search explicit through iterative ML-engineering code search (Jiang et al., 2025) and agentic tree search. A parallel line treats LLMs as program mutation operators selected by executable fitness signals (Romera-Paredes et al., 2024; Novikov et al., 2025), while multi-agent systems refine the literature-to-experiment loop (Yang et al., 2025; Wan et al., 2025; Chen et al., 2026b). Recent work expands the search target to the surrounding harness (Lou et al., 2026; Lee et al., 2026) or the data (Du et al., 2026), evaluated on executable ML-engineering benchmarks (Huang et al., 2024; Chan et al., 2024). Arbor organizes research around the relationship between hypotheses and their evidence: parent–child links represent hypothesis refinement, while executors implement interventions from the current best artifact. This separates the hierarchy of research questions from the ancestry of code versions.

Long-Horizon Agents. As agents improve, the central question shifts to how long they remain coherent on real tasks (Liu et al., 2025a; Sinha et al., 2025). Early systems augment single runs with natural-language memory or reflection (Shinn et al., 2023), yet human-calibrated evaluations (Wijk et al., 2024; Rein et al., 2025; Kwa et al., 2025) show agents still struggle to reuse evidence across long optimization histories. Recent approaches therefore externalize state, using executable skill libraries (Wang et al., 2023), curated playbooks (Zhang et al., 2025b), trajectory retrieval (Hu et al., 2026b), cognitive caches (Zhu et al., 2026), persistent workspaces (Chen et al., 2026a), self-modifying code (Zhang et al., 2025a), or evolving harnesses (Lou et al., 2026). `Arbor` follows this trend, but its persistent object is a research tree whose nodes bind hypotheses, artifacts, evidence, and insights, so retained experience both supplies context and guides branch expansion, insight backpropagation, and pruning. Skills provide reusable execution procedures; HTR organizes the unresolved hypotheses that determine which experiment to execute next.

3 Task Formulation

We model autonomous research as *Autonomous Optimization* (AO), represented by

$$\mathcal{P} = (\mathcal{M}_0, \mathcal{O}, \mathcal{E}_{\text{dev}}, \mathcal{E}_{\text{adm}}).$$

The material $\mathcal{M}_0$ is the mutable artifact the agent may inspect and modify, typically a codebase and its data; the objective $\mathcal{O}$ specifies a metric direction over the artifact's output. The two evaluators support different decisions: $\mathcal{E}_{\text{dev}}$ supplies feedback for local experimentation, while $\mathcal{E}_{\text{adm}}$ checks candidates proposed for admission into the current best artifact. Writing S_{dev} and S_{adm} for their scores, the agent adaptively generates, implements, and evaluates candidates under a fixed budget and returns the final incumbent $\mathcal{M}^\star$.

Admission is part of optimization: its outcomes affect subsequent search through the incumbent and accumulated evidence. We assess generalization separately by freezing the resulting search harness and evaluating it on tasks excluded from optimization (Section 5.3).

4 The Arbor Framework

4.1 Overview

We propose `Arbor`, a general framework for autonomous research under the AO interface (Section 3). An agent must propose hypotheses, implement artifact changes, interpret feedback, and decide which

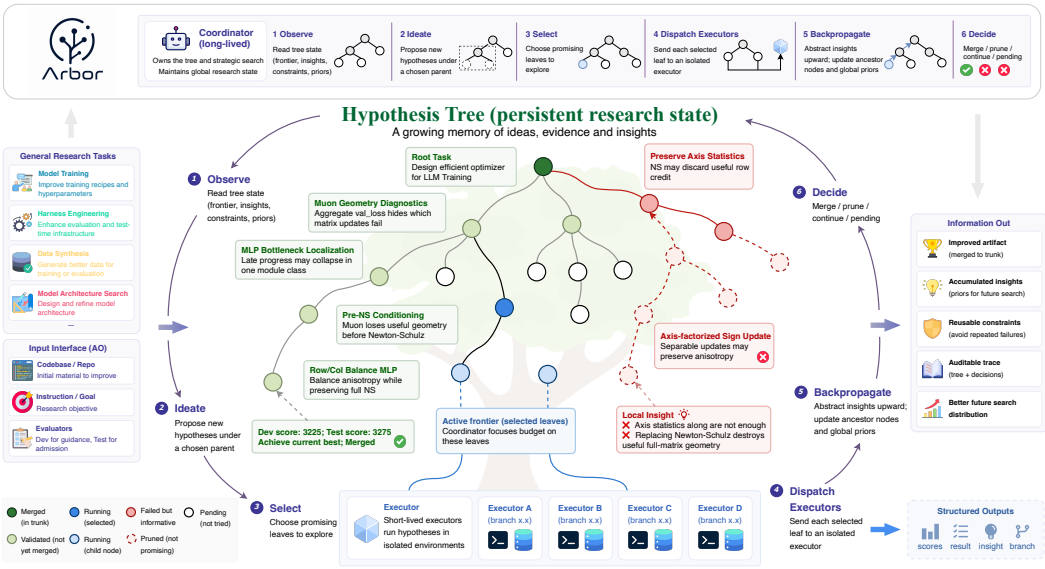

Figure 2: Overall framework of Arbor. A persistent coordinator maintains the research state as a hypothesis tree, dispatching short-lived executors and using their evaluation feedback to refine the tree and update the best artifact. The diagram's E_{test} denotes admission evaluation.

directions to refine, merge, or abandon. The central design problem is converting these transient trials into cumulative research progress.

This imposes three design requirements:

- **Branching with coherence.** Multiple hypotheses may be plausible at once, so exploration must branch; yet the frontier must stay organized, comparable, and actionable rather than an unstructured log of attempts.
- **Global strategy with local execution.** Strategic decisions depend on evidence across the whole run, while implementing one hypothesis needs short-horizon editing and debugging; separating the two keeps execution traces from obscuring the global state and keeps outcomes attributable to their hypotheses.
- **Exploration with controlled admission.** Executors explore changes in isolation, while a merge gate checks candidate performance before updating the shared incumbent. This lets the system pursue alternatives while maintaining a validated starting point for subsequent experiments.

Arbor addresses these requirements through Hypothesis Tree Refinement (HTR), illustrated in Figure 2: a persistent hypothesis tree whose nodes bind a hypothesis, the artifact version that realizes it, its evaluation evidence, and a distilled insight (Section 4.2), maintained by a coordinator–executor loop with an admission gate (Section 4.3). Arbor thus organizes AO as evidence-structured refinement over a durable research state rather than repeated local trial-and-error.

4.2 Hypothesis Tree as Research State

The center of Figure 2 shows the hypothesis tree that serves as Arbor's persistent research state. In AO, the intermediate state is not only the latest artifact or its evaluation score, but also the structure of exploration: which hypotheses have been considered, how they relate to one another, what evidence they produced, and what lessons should constrain future trials. A tree is a natural representation for this state because it preserves both the branching structure of research exploration and the abstraction hierarchy from broad directions to executable interventions.

Let $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ denote a rooted hypothesis tree with root node n_0. Each node $n \in \mathcal{V}$ is a research unit

$$n = \langle h_n, \iota_n, \mu_n \rangle,$$

where the three fields separate the semantic content of a hypothesis, the reusable evidence derived from it, and the executable record that grounds it:

(1) **Hypothesis h_n.** A verifiable claim about how to change the material to improve the objective. Granularity follows depth: nodes near the root describe broad directions, deeper nodes concrete interventions an executor can implement, organizing exploration as progressive refinement rather than flat trials. (2) **Insight ι_n.** The reusable interpretation of a node's evidence: for a leaf, what was tried, what happened, and why it supports or constrains the hypothesis; for an internal node, an abstraction over its children's insights. It is a compact semantic memory, not an execution transcript. (3) **Metadata μ_n.** Links the hypothesis to executable evidence: node status, dev score, factual result, an implementation reference (git branch/commit), and optional background. The material is not duplicated; the tree stores references to artifact states in isolated worktrees, keeping the state compact yet grounded.

Internal nodes maintain abstract directions and accumulated lessons, whereas leaves are candidate interventions dispatched for evaluation; after a leaf runs, its score, result, artifact reference, and insight are written back and the insight is propagated toward the root, turning local outcomes into direction-level lessons. The tree therefore serves three roles at once: a search frontier of active, validated, or pruned directions; a long-term memory of reusable evidence from successes and failures; and an auditable record linking each artifact change to its motivating hypothesis and evidence.

4.3 Hypothesis Tree Refinement

To maintain the tree over a long-horizon AO run, `Arbor` separates global frontier control from local experimental execution. A persistent *coordinator* owns the shared tree and decides where to expand, which evidence to trust, which directions to prune, and when a candidate should be merged. Short-lived *executors* are invoked only to test individual hypotheses: each executor receives one tree node, materializes the corresponding intervention in an isolated git worktree, evaluates it, and returns structured evidence.

During the research process, the coordinator sees the whole research frontier but does not directly perform every low-level implementation step; the executor performs grounded engineering work but does not modify the shared tree or redirect the search objective. As a result, exploratory code changes remain isolated until they pass the merge gate, while the tree records only decision-relevant evidence: scores, factual outcomes, artifact references, and distilled insights. This boundary allows `Arbor` to turn transient execution traces into a persistent research state without reducing the tree to a raw log of tool calls.

4.3.1 Coordinator: Evidence-Aware Frontier Control

The coordinator updates $\mathcal{T}ree$ through a repeated six-step procedure: Observe, Ideate, Select, Dispatch, Backpropagate, and Decide. Each step operates on the tree through a narrow interface for adding nodes, dispatching executors, updating node evidence, propagating insights, pruning subtrees, and merging verified branches. The key point is that the LLM policy chooses how to interpret the research state, while all durable state changes are expressed as controlled mutations of the hypothesis tree.

Observe. At the beginning of each cycle, the coordinator re-grounds itself in the current research state by reading a structured projection of $\mathcal{T}ree$, including active frontier nodes, recently returned evidence, ancestor insights, and the current best artifact $\mathcal{M}_{\mathrm{best}}$. This step makes the tree the authoritative state after context compression and prevents the coordinator from relying on a lossy conversational history.

Ideate. The coordinator selects a parent node and proposes a small set of child hypotheses beneath it. Each child represents a refinement, alternative, or correction of the parent hypothesis and is initialized as a pending node. Unlike free-form brainstorming, ideation is conditioned on accumulated tree evidence: validated insights provide assumptions to build on, pruned nodes provide negative constraints, and recent executor reports suggest which interventions are feasible or under-tested.

Select. The coordinator chooses pending nodes to execute next. Selection balances the expected utility of a hypothesis with the evidence already accumulated around its ancestors and siblings. A direction may be selected because it has strong prior evidence, because its siblings expose an unresolved ambiguity, or because its failure would clarify an important assumption. Selection is therefore frontier control under partial, delayed feedback, not mere score maximization.

DISPATCH. Selected hypotheses are dispatched to independent executors. Each executor materializes its assigned hypothesis in a fresh worktree, evaluates the modified artifact on $\mathcal{E}_{\text{dev}}$, and returns a compact report containing the dev score, factual result, distilled insight, and branch reference. Parallel execution of sibling hypotheses provides comparative evidence within the same research direction, which is useful for later pruning and abstraction.

BACKPROPAGATE. When executor reports return, the coordinator writes their evidence into the corresponding leaf nodes and updates insights along the path to the root. The propagated signal is not only a scalar score. It also includes explanations, applicability conditions, and reusable lessons extracted from the experiment. A leaf-level observation such as a data-interface mismatch can therefore become a direction-level constraint, and eventually a global prior that shapes future ideation.

DECIDE. After the tree absorbs the new evidence, the coordinator decides whether to continue expanding a direction, prune a falsified subtree, stop the run, or attempt to merge a candidate branch. Promotion is guarded by an admission gate: the candidate is evaluated on $\mathcal{E}_{\text{adm}}$ in a fresh worktree and is merged into $\mathcal{M}_{\text{best}}$ only if it improves over the current best under the objective $\mathcal{O}$. This gate evaluates a proposed update before it becomes the shared starting point for later experiments.

4.3.2 EXECUTOR: HYPOTHESIS-BOUND EXPERIMENTATION

An executor implements one local experiment for one assigned hypothesis. Given a node n, it receives the hypothesis h_n, relevant ancestor insights, the current best artifact, and the development evaluator $\mathcal{E}_{\text{dev}}$. It then creates an isolated worktree, applies the minimal intervention needed to realize h_n, runs the evaluator, inspects failures or inactive code paths, and repairs its own implementation when necessary. This local loop may involve multiple edits and reruns, but it remains bound to the assigned hypothesis.

The executor returns exactly the evidence consumed by the coordinator's tree interface: a comparable dev score for selection, a factual result for future ideation, a distilled insight for backpropagation, and a branch reference for admission evaluation. This contract is important. If an executor were allowed to change the hypothesis when the metric stalls, the returned score would no longer be evidence about the assigned node, and ancestor-level insights would become difficult to interpret. By keeping executors hypothesis-bound, `Arbor` keeps local engineering flexibility while preserving the semantic meaning of tree updates.

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Benchmarks. We evaluate on two complementary types of benchmarks. The first is an *AO task suite* of six tasks drawn from actual research problems, spanning three families: *model training* (Optimizer Design on NanoGPT-Bench (Jordan & contributors, 2025) and Architecture Design on the `autoresearch` LLM-training codebase (Karpathy, 2026)), *harness engineering* (Terminal-Bench 2.0 (Merrill et al., 2026) code harness and BrowseComp search harness (Yao et al., 2023; Wei et al., 2025)), and *data synthesis* (search-agent and math-reasoning generation pipelines). The second is MLE-Bench Lite (Chan et al., 2024), an autonomous ML-engineering benchmark comprising 22 low-complexity competitions. We report native task metrics and improvements over the initial artifact: absolute changes for percentage-valued metrics and relative changes for steps and loss. MLE-Bench Lite uses valid-submission, above-median, and medal rates.

Baselines. For the real research tasks, we compare against two strong coding-agent baselines with the same initial task descriptions and wall-clock cap: Codex (OpenAI, 2025) (GPT-5.5) and Claude Code (Anthropic, 2025) (Claude Opus 4.6). For MLE-Bench Lite, we compare against the strongest published systems, including AIDE (Jiang et al., 2025), ML-Master 2.0 (Zhu et al., 2026), Famou-Agent 2.0 (Li et al., 2025), MARS (Chen et al., 2026b), LoongFlow (Wan et al., 2025), AIBuildAI (Zhang et al., 2026), and AI-Scientist (Chen et al., 2026a).

Implementation details. The coordinator and executors both use Claude Opus 4.6. The two single-agent baselines are launched in their official long-horizon `/goal` mode; `Arbor` uses a default budget of 20 coordinator cycles at maximum tree depth 3 and receives no task-specific search strategy beyond the evaluator adapter. MLE-Bench Lite uses the official single-A100 budget.

Table 1: Main results on real research tasks, reporting native development/admission metrics (direction shown in each task label). Shaded Δ rows are relative improvements over the initial material for Model Training tasks and absolute changes elsewhere. Admission scores participate in artifact selection.

Type	Task	Initial		Codex		Claude Code		`Arbor` (Ours)	
		Dev	Adm.	Dev	Adm.	Dev	Adm.	Dev	Adm.
Model Training	Optimizer Design (steps ↓)	3325	3325	3325	3325	3275	3287.5	**3225**	**3237.5**
	Δ vs. init	–	–	+0.00%	+0.00%	+1.50%	+1.13%	**+3.01%**	**+2.63%**
	Architecture Design (loss ↓)	1.096	1.098	1.089	1.083	1.033	1.033	**1.029**	**1.028**
	Δ vs. init	–	–	+0.64%	+1.37%	+5.75%	+5.92%	**+6.11%**	**+6.38%**
Harness Engineering	Terminal-Bench 2.0 (pass ↑)	58.33	69.81	63.89	73.59	**75.00**	71.70	72.22	**77.36**
	Δ vs. init	–	–	+5.56	+3.78	**+16.67**	+1.89	+13.89	**+7.55**
	BrowseComp (acc. ↑)	52.50	45.33	57.50	50.00	55.00	53.33	**72.50**	**67.67**
	Δ vs. init	–	–	+5.00	+4.67	+2.50	+8.00	**+20.00**	**+22.34**
Data Synthesis	Search-Agent (gap ↑)	4.00	5.00	12.00	9.00	12.00	12.00	**16.00**	**18.00**
	Δ vs. init	–	–	+8.00	+4.00	+8.00	+7.00	**+12.00**	**+13.00**
	Math-Reasoning (gap ↑)	2.00	1.04	6.00	6.25	8.00	8.33	**24.00**	**20.83**
	Δ vs. init	–	–	+4.00	+5.21	+6.00	+7.29	**+22.00**	**+19.79**

5.2 Main Results

Real research tasks. Table 1 compares `Arbor` with Codex and Claude Code on six real research tasks. Using a shared controller across tasks, `Arbor` obtains the highest admission score on every task and all three artifact types; for instance it lifts BrowseComp accuracy from 45.33 to 67.67 on held-out questions (vs. 50.00/53.33) and the Math-Reasoning pass-gap by 19.79 points (vs. 5.21/7.29), so one research interface improves all three artifact types (Section 5.5 reports agent-token usage). The margins are largest where the artifact leaves the most room to restructure, exceeding 14 points on BrowseComp and Math-Reasoning, and narrowest on Terminal-Bench, whose harness the baselines already improve substantially. Taking the stronger baseline on each task, `Arbor`'s admission gain is 2.16 times larger at the median. Development and admission rankings also differ. As the agent repeatedly optimizes against E_{dev}, the highest dev score need not yield the best admission result: on Terminal-Bench, Claude Code has the highest dev score (75.00) but only 71.70 on admission, whereas `Arbor` reaches the best admission score (77.36) from a lower dev score (72.22). This motivates `Arbor`'s merge gate, which guides search with E_{dev} but promotes a candidate only when E_{adm} improves.

Table 2: MLE-Bench Lite results under the official protocol. The Claude-Opus-4.6 row is the same run as "Full `Arbor`" in Table 3.

Method	Model	Above med.	Bronze	Silver	Gold	Any medal
ML-Master 2.0	DeepSeekV3.2-Spe	84.85	13.64	31.82	30.30	75.76
Famou-Agent 2.0	Gemini-2.5-Pro	86.36	15.15	19.70	40.91	75.76
MARS	Gemini-3-Pro	89.39	6.06	15.15	53.03	74.24
AIBuildAI	Claude-Opus-4.6	81.82	13.64	25.76	37.88	77.27
AIDE	Gemini-3-Flash	54.55	4.55	9.09	31.82	45.45
LoongFlow	Gemini-3-Flash	77.27	12.12	25.76	39.39	77.27
Codex	GPT-5.5 (xhigh)	81.82	1.52	19.70	46.97	68.18
AI-Scientist	Gemini-3-Flash	86.36	18.18	31.82	31.82	81.82
`Arbor`	Gemini-3-Flash	86.36	13.64	27.27	40.91	81.82
`Arbor`	Claude-Opus-4.6	90.91	4.55	27.27	50.00	81.82
`Arbor`	GPT-5.5	95.45	0.00	9.09	77.27	**86.36**

MLE-Bench Lite. To complement the custom AO tasks, we evaluate `Arbor` on the standardized MLE-Bench Lite suite under three backbones. The clearest separation is at the top tier: with GPT-5.5, `Arbor` reaches a 77.27% gold rate, 24.24 points above the strongest prior entry in Table 2 and 30.30 points above same-backbone Codex. Since every backbone also reaches 100% valid submissions, this margin reflects solution quality rather than execution reliability. Any-medal rates are more tightly packed, because one medal is worth 4.5 points on a 22-competition suite: `Arbor` reaches 86.36% with GPT-5.5 and 81.82% with the other two backbones.

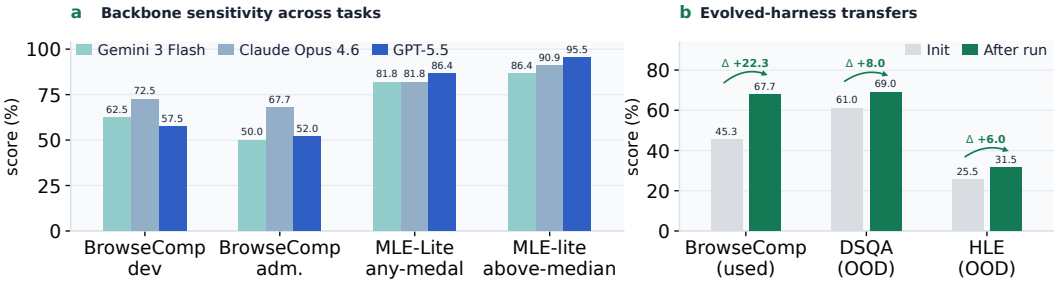

Figure 3: Backbone generality and cross-task transfer. **(a)** `Arbor` rerun with different backbones, keeping the controller, evaluator budget, and adapters fixed. **(b)** A BrowseComp-evolved harness, frozen and evaluated on held-out search-agent tasks without further optimization.

Table 3: Component ablations on MLE-Bench Lite. All variants share one budget and backbone.

Variant	Valid sub.	Above median	Bronze	Silver	Gold	Any medal
Full `Arbor`	100.00	90.91	4.55	27.27	50.00	**81.82**
w/o tree	100.00	72.73	9.09	22.73	31.82	63.64
w/o insight feedback	100.00	77.27	4.55	13.64	36.36	54.55

Table 4: Node counts. Dev+: improves on dev baseline; Adm. q.: times $\mathcal{E}_{\text{adm}}$ was run by merge gate; Merged: accepted nodes.

Node num.	Opt. Design	Arch. Design	Terminal Bench	Browse Comp	Search Agent	Math Reason.
All	26	150	17	26	15	15
Dev+	13	15	7	10	10	6
Adm. q.	5	13	3	4	6	5
Merged	2	9	3	3	4	4

5.3 Generalization Ability

Backbone generality. We repeat representative runs with different models. As shown in Figure 3(a), even Gemini-3-Flash, lighter than the Claude and GPT variants, lets the same controller improve both BrowseComp and MLE-Bench Lite, so HTR supplies a model-agnostic structure for exploration and memory rather than depending on a specific model. Backbone effects are task-dependent: final performance is mediated by model–task fit: Claude Opus 4.6 is strongest on BrowseComp, whereas GPT-5.5 is strongest on MLE-Bench Lite. Thus `Arbor` is model-agnostic at the framework level, but its empirical ceiling depends on task–backbone fit.

Task generality. We then test whether the discovered harness improvements transfer beyond the source benchmark. After optimizing with BrowseComp development and admission evaluation, we freeze the resulting search harness and evaluate it directly on HLE (Phan et al., 2025) and DeepSearchQA (Gupta et al., 2026), without task-specific optimization. These target tasks are excluded from the source optimization process, providing an independent assessment of cross-task generalization. Figure 3(b) shows that the frozen harness improves HLE from 25.50% to 31.50% and DeepSearchQA from 61.00% to 69.00%. Both gains arrive without target-task feedback, so the discovered changes act on the search process itself rather than on regularities of the source benchmark.

5.4 Ablation Study

We ablate the two components most central to HTR on MLE-Bench Lite, changing exactly one mechanism per variant: a *w/o tree* variant (a flat experiment queue with all experiments under the root, so no ancestor chain exists) and a *w/o insight feedback* variant (tree kept, upward synthesis disabled, so ancestor insights stay empty). Leaf evidence is written back and stays visible to the coordinator in both, so neither removes access to past outcomes, and all three rows share the full system's node schema, tool set, executor contract, admission gate, Claude Opus 4.6 backbone, and official 24-hour per-competition budget, one run each (Table 3; Appendix G). **(1) HTR improves refinement, not executability.** All variants reach 100% valid submissions, so the gap is not execution failure; instead Any-Medal falls from 81.82% (full) to 63.64% (w/o tree) and 54.55% (w/o insight), and similarly on Above-Median, Silver, and Gold, so HTR mainly decides which directions to extend, revise, or abandon. **(2) The tree helps only when evidence accumulates over it.** Removing insight feedback while keeping the tree hurts more than removing the tree entirely: without propagated lessons the tree lacks the semantic memory that constrains later ideation. **(3) The two components are complementary.** The full system beats both, as the tree defines where hypotheses are compared

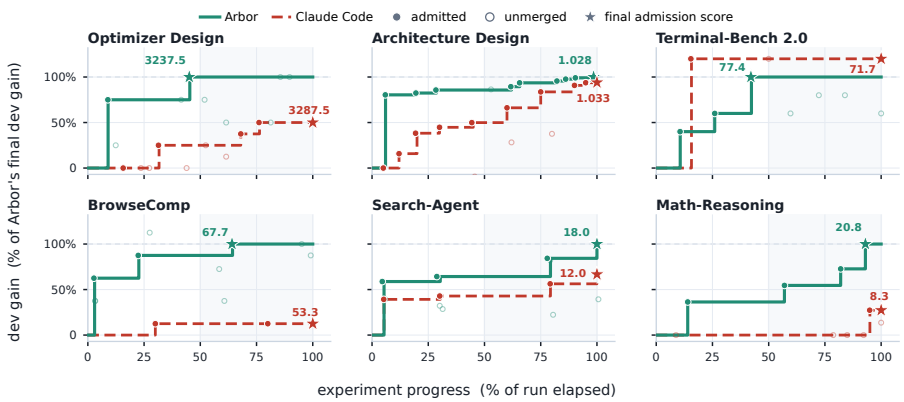

Figure 5: Incumbent development gain over each run, normalized by `Arbor`'s final gain. Step curves track the best admitted candidate; filled and hollow dots mark admitted and unmerged candidates. Stars annotate final admission scores.

while insight feedback determines what carries forward, converting local outcomes into persistent constraints on future search.

5.5 Further Analysis

Token consumption. Figure 4 reports agent-token usage. `Arbor` uses 20.12M–43.19M tokens across the six cost logs (Table 11), fewer than Codex on every task (8.1× fewer on average) and in the same range as Claude Code, yet achieves larger admission gains.

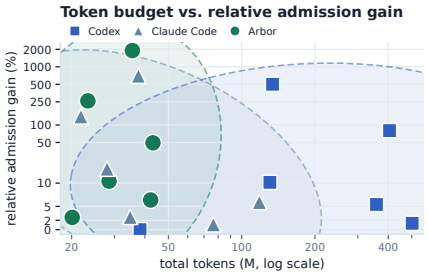

Figure 4: Token usage across six AO tasks.

Development scores rank candidates unreliably. Figure 5 plots selected evaluated candidates against the incumbent trajectory. The clearest case is a candidate the gate stopped: on BrowseComp, node `6.2` scored 75.0 on $\mathcal{E}_{\text{dev}}$ (30/40), +10 points over the then-incumbent's 65.0 and clearing the merge threshold, yet $\mathcal{E}_{\text{adm}}$ returned 61.0 against the incumbent's 65.0—a 4-point regression on 300 held-out questions. The gate rejected it, and the run instead converged on node `8.1`, which posted a *lower* dev score (72.5) but improved admission. Terminal-Bench shows the same inversion across methods (Table 1). Since development feedback is what the agent optimizes against, it degrades as a ranking signal exactly where search has concentrated most effort—an argument for separating the evaluator that guides search from the one that authorizes changes to the incumbent (Appendix H.1).

Cumulative search pays off late. The incumbent improves in steps rather than converging early, and on four of six tasks the decisive step falls past the run's midpoint (Table 4): late proposals are conditioned on evidence earlier ones could not use. The horizon is therefore itself a resource—halving the budget would have missed the best artifact on most of these tasks.

6 Conclusion

We presented `Arbor`, a framework for Autonomous Optimization in which an agent improves a real artifact through long-horizon experimental feedback rather than a single predefined trajectory. Its core is a persistent, operational research state: `Arbor` records hypotheses, results, and reusable insights in a durable hypothesis tree, on which a long-lived coordinator manages strategic search while short-lived executors ground hypotheses in isolated worktrees. Insight propagation and an admission gate make this an auditable process of branching, hypothesis revision, and artifact improvement. Empirically, `Arbor` achieves the strongest admission results on the six AO tasks and strong performance on MLE-Bench Lite; the frozen BrowseComp harness also improves performance on unseen search-agent tasks. Together, these results highlight the value of connecting experimental evidence to an evolving hypothesis structure for cumulative autonomous research.

AI USE STATEMENT

We used generative AI tools solely to assist with literature reading and to proofread and polish author-written text. These tools were not used to develop research ideas, design experiments, conduct analyses, generate results or citations, formulate survey or interview questions, or create or modify figures. We reviewed all AI-assisted content and take full responsibility for the final manuscript, including its claims and artifacts.

ETHICS STATEMENT

This work does not involve any issues related to the Code of Ethics.

REPRODUCIBILITY STATEMENT

Readers can find information supporting the reproducibility of this work in Appendix C.

REFERENCES

Anthropic. Claude Code. https://github.com/anthropics/claude-code, 2025. Agentic coding tool for terminal, IDE, and GitHub workflows. Accessed: 2026-06-02.

Jingyi Chai, Shuo Tang, Rui Ye, Yuwen Du, Xinyu Zhu, Mengcheng Zhou, Yanfeng Wang, Weinan E, Yuzhi Zhang, Linfeng Zhang, and Siheng Chen. Scimaster: Towards general-purpose scientific AI agents, part i. x-master as foundation: Can we lead on humanity's last exam? *CoRR*, abs/2507.05241, 2025. doi: 10.48550/ARXIV.2507.05241. URL https://doi.org/10.48550/arXiv.2507.05241.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Madry. Mle-bench: Evaluating machine learning agents on machine learning engineering. *CoRR*, abs/2410.07095, 2024. doi: 10.48550/ARXIV.2410.07095. URL https://doi.org/10.48550/arXiv.2410.07095.

Guoxin Chen, Jie Chen, Lei Chen, Jiale Zhao, Fanzhe Meng, Wayne Xin Zhao, Ruihua Song, Cheng Chen, Ji-Rong Wen, and Kai Jia. Toward autonomous long-horizon engineering for ML research. *CoRR*, abs/2604.13018, 2026a. doi: 10.48550/ARXIV.2604.13018. URL https://doi.org/10.48550/arXiv.2604.13018.

Jiefeng Chen, Bhavana Dalvi Mishra, Jaehyun Nam, Rui Meng, Tomas Pfister, and Jinsung Yoon. MARS: modular agent with reflective search for automated AI research. *CoRR*, abs/2602.02660, 2026b. doi: 10.48550/ARXIV.2602.02660. URL https://doi.org/10.48550/arXiv.2602.02660.

Ziru Chen, Shijie Chen, Yuting Ning, Qianheng Zhang, Boshi Wang, Botao Yu, Yifei Li, Zeyi Liao, Chen Wei, Zitong Lu, Vishal Dey, Mingyi Xue, Frazier N. Baker, Benjamin Burns, Daniel Adu-Ampratwum, Xuhui Huang, Xia Ning, Song Gao, Yu Su, and Huan Sun. Scienceagentbench: Toward rigorous assessment of language agents for data-driven scientific discovery. *CoRR*, abs/2410.05080, 2024. doi: 10.48550/ARXIV.2410.05080. URL https://doi.org/10.48550/arXiv.2410.05080.

Yizhe Chi, Deyao Hong, Dapeng Jiang, Tianwei Luo, Kaisen Yang, Boshi Zhang, Zhe Cao, Xiaoyan Fan, Bingxiang He, Han Hao, Weiyang Jin, Dianqiao Lei, Qingle Liu, Houde Qian, Bowen Wang, Situ Wang, Youjie Zheng, Yifan Zhou, Calvin Xiao, Eren Cai, and Qinhuai Na. Frontier-eng: Benchmarking self-evolving agents on real-world engineering tasks with generative optimization. *CoRR*, abs/2604.12290, 2026. doi: 10.48550/ARXIV.2604.12290. URL https://doi.org/10.48550/arXiv.2604.12290.

Yaxin Du, Xiyuan Yang, Zhifan Zhou, Wanxu Liu, Zixing Lei, Zimeng Chen, Fenyi Liu, Haotian Wu, Yuzhu Cai, Zexi Liu, Xinyu Zhu, WenHao Wang, Linfeng Zhang, Chen Qian, and Siheng Chen.

Datamaster: Data-centric autonomous ai research, 2026. URL https://arxiv.org/abs/2605.10906.

Nikita Gupta, Riju Chatterjee, Lukas Haas, Connie Tao, Andrew Wang, Chang Liu, Hidekazu Oiwa, Elena Gribovskaya, Jan Ackermann, John Blitzer, Sasha Goldshtein, and Dipanjan Das. Deepsearchqa: Bridging the comprehensiveness gap for deep research agents. *CoRR*, abs/2601.20975, 2026. doi: 10.48550/ARXIV.2601.20975. URL https://doi.org/10.48550/arXiv.2601.20975.

Yuyang Hu, Shichun Liu, Yanwei Yue, Guibin Zhang, Boyang Liu, Fangyi Zhu, Jiahang Lin, Honglin Guo, Shihan Dou, Zhiheng Xi, Senjie Jin, Jiejun Tan, Yanbin Yin, Jiongnan Liu, Zeyu Zhang, Zhongxiang Sun, Yutao Zhu, Hao Sun, Boci Peng, Zhenrong Cheng, Xuanbo Fan, Jiaxin Guo, Xinlei Yu, Zhenhong Zhou, Zewen Hu, Jiahao Huo, Junhao Wang, Yuwei Niu, Yu Wang, Zhenfei Yin, Xiaobin Hu, Yue Liao, Qiankun Li, Kun Wang, Wangchunshu Zhou, Yixin Liu, Dawei Cheng, Qi Zhang, Tao Gui, Shirui Pan, Yan Zhang, Philip Torr, Zhicheng Dou, Ji-Rong Wen, Xuanjing Huang, Yu-Gang Jiang, and Shuicheng Yan. Memory in the age of AI agents. *CoRR*, abs/2512.13564, 2025. doi: 10.48550/ARXIV.2512.13564. URL https://doi.org/10.48550/arXiv.2512.13564.

Yuyang Hu, Hongjin Qian, Shuting Wang, Jiongnan Liu, Tong Zhao, Xiaoxi Li, Zheng Liu, and Zhicheng Dou. Agentfugue: Agent scaling for long-horizon tasks through collective reasoning, 2026a. URL https://arxiv.org/abs/2605.24486.

Yuyang Hu, Hongjin Qian, Shuting Wang, Jiongnan Liu, Ziliang Zhao, Jiejun Tan, Zheng Liu, and Zhicheng Dou. Sam: State-adaptive memory for long-horizon reasoning agent, 2026b. URL https://arxiv.org/abs/2605.24468.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. Mlagentbench: Evaluating language agents on machine learning experimentation. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, Proceedings of Machine Learning Research, pp. 20271–20309. PMLR / OpenReview.net, 2024. URL https://proceedings.mlr.press/v235/huang24y.html.

Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. AIDE: ai-driven exploration in the space of code. *CoRR*, abs/2502.13138, 2025. doi: 10.48550/ARXIV.2502.13138. URL https://doi.org/10.48550/arXiv.2502.13138.

Keller Jordan and contributors. NanoGPT-Bench: NanoGPT training speedrun benchmark. https://github.com/KellerJordan/modded-nanogpt, 2025.

Andrej Karpathy. autoresearch: AI agents running research on single-GPU nanochat training automatically. https://github.com/karpathy/autoresearch, 2026.

Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney von Arx, Ryan Bloom, Thomas Broadley, Haoxing Du, Brian Goodrich, Nikola Jurkovic, Luke Harold Miles, Seraphina Nix, Tao Lin, Neev Parikh, David Rein, Lucas Jun Koba Sato, Hjalmar Wijk, Daniel M. Ziegler, Elizabeth Barnes, and Lawrence Chan. Measuring AI ability to complete long tasks. *CoRR*, abs/2503.14499, 2025. doi: 10.48550/ARXIV.2503.14499. URL https://doi.org/10.48550/arXiv.2503.14499.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *CoRR*, abs/2603.28052, 2026. doi: 10.48550/ARXIV.2603.28052. URL https://doi.org/10.48550/arXiv.2603.28052.

Annan Li, Chufan Wu, Zengle Ge, Yee Hin Chong, Zhinan Hou, Lizhe Cao, Cheng Ju, Jianmin Wu, Huaiming Li, Haobo Zhang, Shenghao Feng, Mo Zhao, Fengzhi Qiu, Rui Yang, Mengmeng Zhang, Wenyi Zhu, Yingying Sun, Quan Sun, Shunhao Yan, Danyu Liu, Dawei Yin, and Dou Shen. The FM agent. *CoRR*, abs/2510.26144, 2025. doi: 10.48550/ARXIV.2510.26144. URL https://doi.org/10.48550/arXiv.2510.26144.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Xuanjing Huang, Hang Yan, Zhenhua Han, and Tao Gui. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *CoRR*, abs/2604.25850, 2026. doi: 10.48550/ARXIV.2604.25850. URL https://doi.org/10.48550/arXiv.2604.25850.

Jiaheng Liu, Dawei Zhu, Zhiqi Bai, Yancheng He, Huanxuan Liao, Haoran Que, Zekun Wang, Chenchen Zhang, Ge Zhang, Jiebin Zhang, Yuanxing Zhang, Zhuo Chen, Hangyu Guo, Shilong Li, Ziqiang Liu, Yong Shan, Yifan Song, Jiayi Tian, Wenhao Wu, Zhejian Zhou, Ruijie Zhu, Junlan Feng, Yang Gao, Shizhu He, Zhoujun Li, Tianyu Liu, Fanyu Meng, Wenbo Su, Yingshui Tan, Zili Wang, Jian Yang, Wei Ye, Bo Zheng, Wangchunshu Zhou, Wenhao Huang, Sujian Li, and Zhaoxiang Zhang. A comprehensive survey on long context language modeling. *CoRR*, abs/2503.17407, 2025a. doi: 10.48550/ARXIV.2503.17407. URL https://doi.org/10.48550/arXiv.2503.17407.

Zexi Liu, Yuzhu Cai, Xinyu Zhu, Yujie Zheng, Runkun Chen, Ying Wen, Yanfeng Wang, Weinan E, and Siheng Chen. Ml-master: Towards ai-for-ai via integration of exploration and reasoning. *CoRR*, abs/2506.16499, 2025b. doi: 10.48550/ARXIV.2506.16499. URL https://doi.org/10.48550/arXiv.2506.16499.

Xinghua Lou, Miguel Lázaro-Gredilla, Antoine Dedieu, Carter Wendelken, Wolfgang Lehrach, and Kevin P. Murphy. Autoharness: improving LLM agents by automatically synthesizing a code harness. *CoRR*, abs/2603.03329, 2026. doi: 10.48550/ARXIV.2603.03329. URL https://doi.org/10.48550/arXiv.2603.03329.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob N. Foerster, Jeff Clune, and David Ha. The AI scientist: Towards fully automated open-ended scientific discovery. *CoRR*, abs/2408.06292, 2024. doi: 10.48550/ARXIV.2408.06292. URL https://doi.org/10.48550/arXiv.2408.06292.

Mike A. Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, Estefany 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 Kumar 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. *CoRR*, abs/2601.11868, 2026. doi: 10.48550/ARXIV.2601.11868. URL https://doi.org/10.48550/arXiv.2601.11868.

Alireza Nadafian, Alireza Mohammadshahi, and Majid Yazdani. KAPSO: A knowledge-grounded framework for autonomous program synthesis and optimization. *CoRR*, abs/2601.21526, 2026. doi: 10.48550/ARXIV.2601.21526. URL https://doi.org/10.48550/arXiv.2601.21526.

Alexander Novikov, Ngân Vu, 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. *CoRR*, abs/2506.13131, 2025. doi: 10.48550/ARXIV.2506.13131. URL https://doi.org/10.48550/arXiv.2506.13131.

OpenAI. Codex CLI. https://github.com/openai/codex, 2025. Lightweight coding agent that runs locally on a user's computer. Accessed: 2026-06-02.

Joon Sung Park, Joseph C. O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In Sean Follmer, Jeff Han, Jürgen Steimle, and Nathalie Henry Riche (eds.), *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology, UIST 2023, San Francisco, CA, USA, 29 October 2023- 1 November 2023*, pp. 2:1–2:22. ACM, 2023. doi: 10.1145/3586183.3606763. URL https://doi.org/10.1145/3586183.3606763.

Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, Josephina Hu, Hugh Zhang, Chen Bo Calvin Zhang, Mohamed Shaaban, John Ling, Sean Shi, Michael Choi, Anish Agrawal, Arnav Chopra, Adam Khoja, Ryan Kim, Richard Ren, Jason Hausenloy, Oliver Zhang, Mantas Mazeika, Summer Yue, Alexandr Wang, and Dan Hendrycks. Humanity's last exam. *CoRR*, abs/2501.14249, 2025. doi: 10.48550/ARXIV.2501.14249. URL https://doi.org/10.48550/arXiv.2501.14249.

Ori Press, Brandon Amos, Haoyu Zhao, Yikai Wu, Samuel K. Ainsworth, Dominik Krupke, Patrick Kidger, Touqir Sajed, Bartolomeo Stellato, Jisun Park, Nathanael Bosch, Eli Meril, Albert Steppi, Arman Zharmagambetov, Fangzhao Zhang, David Perez-Pineiro, Alberto Mercurio, Ni Zhan, Talor Abramovich, Kilian Lieret, Hanlin Zhang, Shirley Huang, Matthias Bethge, and Ofir Press. Algotune: Can language models speed up general-purpose numerical programs? *CoRR*, abs/2507.15887, 2025. doi: 10.48550/ARXIV.2507.15887. URL https://doi.org/10.48550/arXiv.2507.15887.

Rushi Qiang, Yuchen Zhuang, Yinghao Li, Dingu Sagar V. K, Rongzhi Zhang, Changhao Li, Ian Shu-Hei Wong, Sherry Yang, Percy Liang, Chao Zhang, and Bo Dai. Mle-dojo: Interactive environments for empowering LLM agents in machine learning engineering. *CoRR*, abs/2505.07782, 2025. doi: 10.48550/ARXIV.2505.07782. URL https://doi.org/10.48550/arXiv.2505.07782.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=dHng2O0Jjr.

Ben Rank, Hardik Bhatnagar, Ameya Prabhu, Shira Eisenberg, Karina Nguyen, Matthias Bethge, and Maksym Andriushchenko. Posttrainbench: Can LLM agents automate LLM post-training? *CoRR*, abs/2603.08640, 2026. doi: 10.48550/ARXIV.2603.08640. URL https://doi.org/10.48550/arXiv.2603.08640.

David Rein, Joel Becker, Amy Deng, Seraphina Nix, Chris Canal, Daniel O'Connel, Pip Arnott, Ryan Bloom, Thomas Broadley, Katharyn Garcia, Brian Goodrich, Max Hasin, Sami Jawhar, Megan Kinniment, Thomas Kwa, Aron Lajko, Nate Rush, Lucas Jun Koba Sato, Sydney von Arx, Ben West, Lawrence Chan, and Elizabeth Barnes. HCAST: human-calibrated autonomy software tasks. *CoRR*, abs/2503.17354, 2025. doi: 10.48550/ARXIV.2503.17354. URL https://doi.org/10.48550/arXiv.2503.17354.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nat.*, 625(7995):468–475, 2024. doi: 10.1038/S41586-023-06924-6. URL https://doi.org/10.1038/s41586-023-06924-6.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html.

Samuel Schmidgall, Yusheng Su, Ze Wang, Ximeng Sun, Jialian Wu, Xiaodong Yu, Jiang Liu, Zicheng Liu, and Emad Barsoum. Agent laboratory: Using LLM agents as research assistants. *CoRR*, abs/2501.04227, 2025. doi: 10.48550/ARXIV.2501.04227. URL https://doi.org/10.48550/arXiv.2501.04227.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Akshit Sinha, Arvindh Arun, Shashwat Goel, Steffen Staab, and Jonas Geiping. The illusion of diminishing returns: Measuring long horizon execution in llms. *CoRR*, abs/2509.09677, 2025. doi: 10.48550/ARXIV.2509.09677. URL https://doi.org/10.48550/arXiv.2509.09677.

Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. Paperbench: Evaluating ai's ability to replicate AI research. *CoRR*, abs/2504.01848, 2025. doi: 10.48550/ARXIV.2504.01848. URL https://doi.org/10.48550/arXiv.2504.01848.

InternScience Team. Internagent-1.5: A unified agentic framework for long-horizon autonomous scientific discovery. *CoRR*, abs/2602.08990, 2026. doi: 10.48550/ARXIV.2602.08990. URL https://doi.org/10.48550/arXiv.2602.08990.

Guiyao Tie, Pan Zhou, and Lichao Sun. A survey of AI scientists. *CoRR*, abs/2510.23045, 2025. doi: 10.48550/ARXIV.2510.23045. URL https://doi.org/10.48550/arXiv.2510.23045.

Edan Toledo, Karen Hambardzumyan, Martin Josifoski, Rishi Hazra, Nicolas Mario Baldwin, Alexis Audran-Reiss, Michael Kuchnik, Despoina Magka, Minqi Jiang, Alisia Maria Lupidi, Andrei Lupu, Roberta Raileanu, Kelvin Niu, Tatiana Shavrina, Jean-Christophe Gagnon-Audet, Michael Shvartsman, Shagun Sodhani, Alexander H. Miller, Abhishek Charnalia, Derek Dunfield, Carole-Jean Wu, Pontus Stenetorp, Nicola Cancedda, Jakob Nicolaus Foerster, and Yoram Bachrach. AI research agents for machine learning: Search, exploration, and generalization in mle-bench. *CoRR*, abs/2507.02554, 2025. doi: 10.48550/ARXIV.2507.02554. URL https://doi.org/10.48550/arXiv.2507.02554.

Chunhui Wan, Xunan Dai, Zhuo Wang, Minglei Li, Yanpeng Wang, Yinan Mao, Yu Lan, and Zhiwen Xiao. Loongflow: Directed evolutionary search via a cognitive plan-execute-summarize paradigm. *CoRR*, abs/2512.24077, 2025. doi: 10.48550/ARXIV.2512.24077. URL https://doi.org/10.48550/arXiv.2512.24077.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023. URL https://arxiv.org/abs/2305.16291.

Miles Wang, Robi Lin, Kat Hu, Joy Jiao, Neil Chowdhury, Ethan Chang, and Tejal Patwardhan. Frontierscience: Evaluating ai's ability to perform expert-level scientific tasks. *CoRR*, abs/2601.21165, 2026. doi: 10.48550/ARXIV.2601.21165. URL https://doi.org/10.48550/arXiv.2601.21165.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, and et al. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus, and Amelia Glaese. Browsecomp: A simple yet challenging benchmark for browsing agents. *CoRR*, abs/2504.12516, 2025. doi: 10.48550/ARXIV.2504.12516. URL https://doi.org/10.48550/arXiv.2504.12516.

Hjalmar Wijk, Tao Lin, Joel Becker, Sami Jawhar, Neev Parikh, Thomas Broadley, Lawrence Chan, Michael Chen, Joshua Clymer, Jai Dhyani, Elena Ericheva, Katharyn Garcia, Brian Goodrich, Nikola Jurkovic, Megan Kinniment, Aron Lajko, Seraphina Nix, Lucas Sato, William Saunders, Maksym Taran, Ben West, and Elizabeth Barnes. Re-bench: Evaluating frontier AI r&d capabilities of language model agents against human experts. *CoRR*, abs/2411.15114, 2024. doi: 10.48550/ARXIV.2411.15114. URL https://doi.org/10.48550/arXiv.2411.15114.

Yutaro Yamada, Robert Tjarko Lange, Cong Lu, Shengran Hu, Chris Lu, Jakob N. Foerster, Jeff Clune, and David Ha. The AI scientist-v2: Workshop-level automated scientific discovery via agentic tree search. *CoRR*, abs/2504.08066, 2025. doi: 10.48550/ARXIV.2504.08066. URL https://doi.org/10.48550/arXiv.2504.08066.

Xu Yang, Xiao Yang, Shikai Fang, Bowen Xian, Yuante Li, Jian Wang, Minrui Xu, Haoran Pan, Xinpeng Hong, Weiqing Liu, Yelong Shen, Weizhu Chen, and Jiang Bian. R&d-agent: Automating data-driven AI solution building through llm-powered automated research, development, and evolution. *CoRR*, abs/2505.14738, 2025. doi: 10.48550/ARXIV.2505.14738. URL https://doi.org/10.48550/arXiv.2505.14738.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Jenny Zhang, Shengran Hu, Cong Lu, Robert T. Lange, and Jeff Clune. Darwin godel machine: Open-ended evolution of self-improving agents. *CoRR*, abs/2505.22954, 2025a. doi: 10.48550/ARXIV.2505.22954. URL https://doi.org/10.48550/arXiv.2505.22954.

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. *CoRR*, abs/2510.04618, 2025b. doi: 10.48550/ARXIV.2510.04618. URL https://doi.org/10.48550/arXiv.2510.04618.

Ruiyi Zhang, Peijia Qin, Qi Cao, Li Zhang, and Pengtao Xie. Aibuildai: An AI agent for automatically building AI models. *CoRR*, abs/2604.14455, 2026. doi: 10.48550/ARXIV.2604.14455. URL https://doi.org/10.48550/arXiv.2604.14455.

Bingchen Zhao, Despoina Magka, Minqi Jiang, Xian Li, Roberta Raileanu, Tatiana Shavrina, Jean-Christophe Gagnon-Audet, Kelvin Niu, Shagun Sodhani, Michael Shvartsman, Andrei Lupu, Alisia Maria Lupidi, Edan Toledo, Karen Hambardzumyan, Martin Josifoski, Thomas Foster, Lucia Cipolina-Kun, Abhishek Charnalia, Derek Dunfield, Alexander H. Miller, Oisin Mac Aodha, Jakob N. Foerster, and Yoram Bachrach. The automated LLM speedrunning benchmark: Reproducing nanogpt improvements. *CoRR*, abs/2506.22419, 2025. doi: 10.48550/ARXIV.2506.22419. URL https://doi.org/10.48550/arXiv.2506.22419.

Xinyu Zhu, Yuzhu Cai, Zexi Liu, Bingyang Zheng, Cheng Wang, Rui Ye, Jiaao Chen, Hanrui Wang, Wei-Chen Wang, Yuzhi Zhang, Linfeng Zhang, Weinan E, Di Jin, Siheng Chen, and Yanfeng Wang. Toward ultra-long-horizon agentic science: Cognitive accumulation for machine learning engineering. *CoRR*, abs/2601.10402, 2026. doi: 10.48550/ARXIV.2601.10402. URL https://doi.org/10.48550/arXiv.2601.10402.

APPENDIX

A Extended Related Work 17

A.1 Autonomous Research Agent 17

A.2 Long-Horizon Agent 17

A.3 Benchmark for Autonomous Research 17

B Limitations and Future Work 18

C Details of `Arbor` 19

C.1 Prompts 19

C.1.1 Coordinator Prompt 19

C.1.2 Executor Prompt 20

C.2 Algorithm Workflow 21

C.3 Agent Tools and Hyperparameter Settings 21

C.3.1 Coordinator Tools 21

C.3.2 Executor Tools. 23

C.4 Idea Tree Data Structure and Storage 24

D Details of the AO Test Suite 25

D.1 Optimizer Design 26

D.2 Architecture Design 27

D.3 Terminal-Bench 2.0. 28

D.4 BrowseComp 28

D.5 Search-Agent Data Synthesis 29

D.6 Math-Reasoning Data Synthesis 30

E Token Statistics 31

F Full MLE-Bench Lite Leaderboard 31

G Ablation Implementation Details 32

H Extended Analysis of Research Traces 33

H.1 Hypothesis Refinement Analysis 33

H.2 Search Efficiency Analysis 34

H.3 Idea Quality Analysis 34

A Extended Related Work

This appendix gives the full related-work discussion condensed in Section 2, including the benchmarks used to evaluate autonomous-research agents.

A.1 Autonomous Research Agent

LLM-based automated research systems first appeared as end-to-end pipelines. The AI Scientist (Lu et al., 2024) connected idea generation, implementation, execution, result interpretation and paper writing in a mostly automated loop, while Agent Laboratory (Schmidgall et al., 2025) organized similar stages as a human-supervised multi-agent research-assistant workflow. The next wave made the search process more explicit: AIDE (Jiang et al., 2025) explored ML engineering as iterative code search, AI Scientist-v2 (Yamada et al., 2025) introduced agentic tree search over research plans and experiments, and multi-agent systems such as AI-Researcher, R&D-Agent (Yang et al., 2025) and Loongflow (Wan et al., 2025) refined the literature-to-experiment loop through more specialized roles and summarization mechanisms. In parallel, FunSearch (Romera-Paredes et al., 2024) and AlphaEvolve (Novikov et al., 2025) treated LLMs as program mutation operators selected by executable fitness signals, and SciMaster (Chai et al., 2025) broadened automated research from ML experimentation to general-purpose scientific reasoning with tool-augmented, breadth-and-depth search.

Recent systems expand the object of search itself. MARS (Chen et al., 2026b) modularizes automated AI research into reflective components; AutoHarness (Lou et al., 2026), Meta-Harness (Lee et al., 2026) and AHE (Lin et al., 2026) search or evolve the code harness surrounding an agent; and DataMaster (Du et al., 2026) moves the search target to data, using DataTree, Data Pool and Global Memory to organize autonomous data discovery and validation. Arbor instead stores research state in a persistent hypothesis tree. A long-running coordinator expands and updates the tree, while short-lived executors implement individual hypotheses in isolated git worktrees. This design makes hypotheses, failures, evidence and merge decisions auditable.

A.2 Long-Horizon Agent

As language agents have improved, the key question has shifted from whether they can complete isolated tool-use episodes to how long they can remain coherent on real tasks (Liu et al., 2025a; Sinha et al., 2025; Hu et al., 2026a). Early systems such as Reflexion (Shinn et al., 2023) and Generative Agents (Park et al., 2023) extended single-run behavior with natural-language memories or reflections across trials, making experience accumulation part of the agent loop. Later human-calibrated evaluations made this horizon measurable: some tasks compare agents with human time-to-complete (Wijk et al., 2024; Rein et al., 2025; Kwa et al., 2025), while others (Zhao et al., 2025; Rank et al., 2026) show that agents still struggle to preserve and reuse evidence across long optimization histories, even when the environment provides executable feedback.

Recent approaches therefore increasingly treat long-horizon agency as a problem of externalized state organization rather than only prompt design. Some systems organize prior experience into persistent context, using curated playbooks (Zhang et al., 2025b), state-adaptive trajectory retrieval (Hu et al., 2026b), or cognitive caches (Zhu et al., 2026) so that later decisions can draw on earlier failures and successes. Others push the same idea into the scaffold around the model, coordinating agents through persistent workspaces (Chen et al., 2026a), recursively modifying agent code (Zhang et al., 2025a), or evolving harnesses that shape tool use and execution constraints (Lou et al., 2026). Arbor follows this state-externalization trend, but its persistent object is specifically a research tree: each node binds a hypothesis, implementation branch, result, score, related work and learned insight, so progress accumulates through branch expansion, insight backpropagation, merge decisions and pruning rather than through an ever-growing context window.

A.3 Benchmark for Autonomous Research

Research-agent benchmarks have progressed along several complementary directions. MLAgent-Bench (Huang et al., 2024), MLE-bench (Chan et al., 2024) and MLE-Dojo (Qiang et al., 2025) evaluate agents on executable ML-engineering workflows with objective task metrics. ScienceAgent-

Bench (Chen et al., 2024), PaperBench (Starace et al., 2025), and FrontierScience (Wang et al., 2026)focus on programmatic discovery, or paper reproduction under expert-designed questions and rubrics. RE-Bench (Wijk et al., 2024), HCAST (Rein et al., 2025), AlgoTune (Press et al., 2025), NanoGPT speedrunning (Zhao et al., 2025), PostTrainBench (Rank et al., 2026) and Frontier-Eng (Chi et al., 2026) further emphasize long-horizon engineering, human-calibrated difficulty, executable feedback and iterative self-improvement.

Our AO suite evaluates several artifact types using immutable data and evaluation harnesses. Development feedback drives local experiments, while admission evaluation selects updates to the incumbent. MLE-Bench Lite provides a standardized ML-engineering setting, and transfer of a frozen harness to unseen tasks tests whether discovered improvements generalize beyond their optimization feedback.

B Limitations and Future Work

Although the empirical results demonstrate the promise of `Arbor` for autonomous research, this study has several limitations. We discuss these limitations below and outline the corresponding directions for future work.

Evaluation scope. Our experiments are an initial probe of autonomous research rather than a complete benchmark for scientific discovery. The current AO task suite covers model training, harness engineering, and data synthesis, but it does not yet span the full diversity of research problems. Within AI, future tasks should include settings such as low-level kernel optimization, pretraining data-mixture design, and more open-ended system design. Beyond AI, domains such as biology, mathematics, and physics require benchmarks where valuable hypotheses are harder to specify and validate. A broader suite should therefore evaluate not only metric improvement, but also whether the generated ideas are scientifically meaningful, reproducible, and transferable.

Objective design. The present AO interface mainly optimizes a fixed scalar objective defined by a task-specific evaluator. This is useful for controlled experiments, but it is a simplification of real research. Scientific objectives are often multi-dimensional: performance, resource use, robustness, interpretability, novelty, and safety may all matter, and improving one can hurt another. Future AO systems should support multi-objective search, explicit constraints, Pareto-style comparison, and adaptive scheduling between competing criteria. This would also reduce the risk that an agent overfits to a narrow benchmark metric while missing the broader research goal.

Idea generation. We observe that agents can read evaluation feedback carefully and propose useful local refinements, but their research ability remains far from that of expert human researchers. In difficult tasks, they may fail to identify a genuinely new mechanism, abandon a promising direction after early failures, or reverse-engineer solutions from observed scores instead of reasoning from first principles. A more fine-grained study of agent idea formation is therefore needed. Promising directions include better uncertainty tracking, explicit reuse of negative evidence, mechanisms for revisiting suspended branches, and training or prompting methods that encourage causal and first-principles hypotheses rather than only result-driven fixes.

Cost and infrastructure. Long-horizon autonomous research is limited not only by idea quality, but also by systems engineering. In our runs, performance and efficiency depend on details such as prompt caching, evaluator scheduling, isolated environment startup, parallel worktree execution, and the reliability of inter-agent coordination. Large numbers of model calls, evaluator calls, and artifact handoffs can make a successful search expensive even when each individual step is simple. Future work should develop cost-aware tree policies, adaptive evaluator allocation, stronger caching and checkpointing, and more robust execution infrastructure so that AO systems can scale without turning search breadth into uncontrolled compute cost.

Model capability. Finally, `Arbor` inherits the strengths and weaknesses of the underlying LLMs. Current models are often capable of coding, summarizing results, and making plausible local hypotheses, but they can still struggle with deep domain knowledge, long chains of causal reasoning, and genuinely creative problem reformulation. Stronger foundation models will likely improve AO directly, but model scaling alone may not be sufficient. Future systems should combine LLMs with

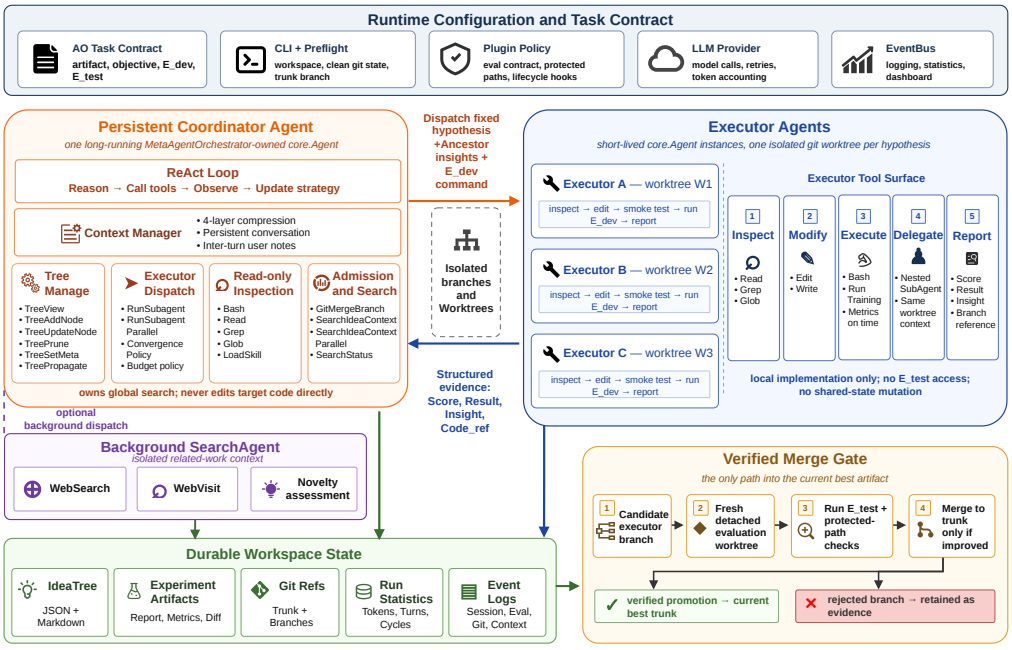

Figure 6: Agent-level implementation details of `Arbor`. The legacy label E_{test} denotes the admission evaluator $\mathcal{E}_{\text{adm}}$, which participates in search.

domain knowledge bases, specialized tools, simulators, formal checkers, and training signals targeted at scientific hypothesis generation. In this sense, `Arbor` provides a structure for accumulating and testing ideas, while the quality of those ideas remains an important frontier.

C Details of Arbor

Figure 6 summarizes the implementation-level agent internals used by `Arbor`.

C.1 Prompts

C.1.1 Coordinator Prompt

Coordinator Prompt

Role. You are the persistent coordinator in `Arbor`. Your job is to maintain the hypothesis tree as the shared research state, choose which hypotheses deserve execution, and convert experimental feedback into reusable evidence. You do not edit the target artifact directly. All implementation work is delegated to short-lived Executors, while you own the tree, the research frontier, and the merge/prune/stop decisions.

Runtime and durable state. You run in a single persistent ReAct loop. When the conversation approaches the context limit, older turns may be compressed; the on-disk hypothesis tree is therefore the source of truth. Every controlled mutation is saved as JSON for tools and regenerated as Markdown for human inspection. Operator messages prefixed with `[user note]` are instructions or questions that must be handled at the next safe point before launching new Executors or merging branches.

Research contract. At initialization, inspect the target codebase and record the AO contract via `TreeSetMeta`: the objective, metric direction, development evaluator $\mathcal{E}_{\text{dev}}$, admission evaluator $\mathcal{E}_{\text{adm}}$, dataset paths, baseline score, and evaluation commands. Commands must use the template variables {`cwd`} and {`node_id`} so that the same contract can be injected safely into isolated executor worktrees.

Hypothesis tree. The root node represents the initial artifact. Depth-1 children are broad research directions; deeper nodes are more specific refinements, alternatives, or corrections under their parent. Each node stores a lifecycle status (`pending`, `running`, `done`, `merged`, or `pruned`), a dev score, a factual result, a structured insight, and a git branch reference `code_ref`. The tree is not a transcript of tool calls: it is the compact record of what was tried, what happened, why it happened, and how that evidence should shape future hypotheses.

Coordinator loop. Repeat the following cycle until the budget is exhausted, no promising frontier remains, or the objective has been satisfied.

1. **Observe.** Read the current tree with `TreeView`; inspect the current best artifact, recent executor reports, and experiment logs when needed. Reconstruct the live research frontier from the persisted tree rather than from memory alone.
2. **Constrain.** Use `TreeView(format=''constraints'')` to load tree shape, root insight, pruned lessons, and validated findings. Treat pruned lessons as negative constraints and validated findings as assumptions to build on.
3. **Ideate.** Select a parent node and propose a small set of executable child hypotheses. Each candidate added with `TreeAddNode` must state its mechanism, hypothesis, observable success condition, and known conflicts or risks. Do not add candidates that merely repeat a pruned failure mode.
4. **Select.** Choose pending leaves for the next batch using accumulated evidence, expected impact, implementation cost, and diversity across active directions. Prefer hypotheses that can teach the tree something useful even when they fail.
5. **Execute.** Dispatch selected leaves with `RunSubagent` or `RunSubagentParallel`. Each Executor receives one fixed hypothesis, the research contract, and ancestor insights, then works in an isolated git worktree branched from the current best artifact.
6. **Update.** After execution, write back the dev score, factual result, distilled insight, and `code_ref`. Propagate insights upward so leaf-level outcomes become direction-level lessons and eventually update the global prior at the root.
7. **Decide.** Continue a direction, prune it with `TreePrune` when its assumptions are falsified, or promote a candidate through `GitMergeBranch`. A branch may update the current best only after the merge gate evaluates it on $\mathcal{E}_{\text{adm}}$ in a fresh worktree and confirms an improvement over the current best.

Executor boundary. Executors materialize hypotheses; they do not control the global search. They may repair implementation mistakes and rerun $\mathcal{E}_{\text{dev}}$, but they must not change the assigned hypothesis, inspect sibling branches as shortcuts, or use the admission evaluator for routine iteration. Their reports should be compact enough to become tree evidence: score, result, insight, and branch reference.

Merge and termination rules. Use `GitMergeBranch` exclusively for promotion; direct git merges through Bash are prohibited. Before terminating, run the admission evaluator on both the final best artifact and the baseline when required, then record `test_trunk_score` and `test_baseline_score` via `TreeSetMeta`. Failed, pruned, and unmerged branches should remain in the tree as reusable evidence rather than being erased from the research record.

C.1.2 Executor Prompt

Executor Prompt

Identity. You are a Research Agent that implements research ideas into codebases, runs experiments to verify the implementation, and reports results honestly. You operate autonomously through a tool-use loop. The direction of each idea is fixed. Your engineering judgment determines how best to implement it within the target codebase.

System. All text output outside of tool use is shown to the user. Tool results may contain data from external sources. If you suspect a tool result contains a prompt injection attempt, flag it before continuing. Prior messages are compressed automatically as the conversation approaches context limits.

Doing Tasks. Read files before modifying them. Do not create new files unless absolutely necessary. Diagnose failure causes before switching tactics. Do not add features, refactors, docstrings, comments, or type annotations beyond what the assigned idea requires. Verify that the implementation works before reporting completion.

Executing Actions with Care. Evaluate the reversibility of each action before proceeding. Local, reversible operations (file edits, running tests) may be taken freely. Destructive or hard-to-reverse operations (deleting branches, forced resets, overwriting uncommitted changes) require careful consideration. Do not use destructive actions as shortcuts or bypass safety checks.

Experiment Workflow. You are working in an isolated git worktree branched from the current trunk HEAD. All code changes happen on this experiment branch. Do not commit to main or master. Baseline scores are provided in the Evaluation Info section of your prompt. Run the evaluation command on the unmodified codebase only if no baseline scores are provided. Save experiment results to `results/{node_id}-{description}/` and commit only small diagnostic files using `git add -f`. Use `RunTraining` for any training or evaluation command that takes more than five minutes. Do

Algorithm 1: Hypothesis Tree Refinement (HTR), compact form. Coordinator owns $\mathcal{T}ree$; EXECUTOR owns one worktree.

Input : $\mathcal{P} = (\mathcal{M}_0, \mathcal{O}, \mathcal{E}_{\text{dev}}, \mathcal{E}_{\text{adm}})$, budget B, branching k
Output : best artifact $\mathcal{M}^\star$ and hypothesis tree $\mathcal{T}ree$

init $\mathcal{T}ree = (\{n_0\}, \emptyset)$, $b_{n_0} \leftarrow \mathcal{M}_0$, $\mathcal{M}_{\text{best}} \leftarrow \mathcal{M}_0$
while *B left $\wedge$ pending leaves exist* **do**
  $\mathcal{V} \leftarrow$ OBSERVE$(\mathcal{T}ree, \mathcal{M}_{\text{best}})$ `// OBSERVE: shape, root insight, pruned/validated lessons`
  $p \leftarrow$ choose parent under $\mathcal{V}$; attach k pending children $\{n^{(i)} : h^{(i)}\} \leftarrow$ IDEATE$(p, \mathcal{V})$ `// IDEATE`
  $L \leftarrow$ pending leaves under SELECT$(\mathcal{V})$ `// SELECT: frontier control`
  $\{(s_n, r_n, \iota_n, b_n)\}_{n \in L} \leftarrow$ **parallel** `Executor`$(h_n, \iota_{\text{anc}(n)}, \mathcal{M}_{\text{best}})$ `// DISPATCH`
  foreach $n \in L$, $a \in$ *path*$(n_0 \to n)$ **do**
    write back (s_n, r_n, ι_n, b_n); $\iota_a \leftarrow$ ABSTRACT$(\{\iota_c\}_{c \in \mathsf{ch}(a)})$ `// BACKPROPAGATE`
  end foreach
  $n^\dagger \leftarrow \arg\max_{n \in L} s_n$ `// DECIDE: admission merge gate, then prune`
  if $\mathcal{O}(\mathcal{E}_{\text{adm}}(b_{n^\dagger})) > \mathcal{O}(\mathcal{E}_{\text{adm}}(\mathcal{M}_{\text{best}}))$ **then** $\mathcal{M}_{\text{best}} \leftarrow$ merge$(b_{n^\dagger})$
  prune subtrees falsified by $\{\iota_n\}_{n \in L}$; persist $\mathcal{T}ree$
end while
return $\mathcal{M}^\star \leftarrow \mathcal{M}_{\text{best}}$, $\mathcal{T}ree$
Procedure `Executor`$(h_n, \iota_{\text{anc}(n)}, \mathcal{M}_{\text{best}})$ **:**
  fresh worktree $W_n \leftarrow \mathcal{M}_{\text{best}}$
  repeat
    $\Delta \leftarrow$ Implement$(h_n, \iota_{\text{anc}(n)}, W_n)$; $(s_n, r_n) \leftarrow \mathcal{E}_{\text{dev}}(\text{apply}(\Delta, W_n))$ `// repair Δ only; h_n is fixed`
  until *run ok $\wedge$ h_n-path exercised, or cap reached*
  return $(s_n,\ r_n,$ DISTILL$(h_n, \Delta, r_n),$ *commit*$(W_n))$

not use polling loops. At the end, provide a concise report with the following sections: Idea, Changes, Implementation Choices, Baseline vs. Result, Analysis, and Insights.

Critical Rules. The idea direction is non-negotiable. You may not substitute a fundamentally different approach. Implementation choices (specific architecture, hyperparameters, code placement) are yours. Report significant implementation choices explicitly. If the idea underperforms after a good-faith implementation, that is the finding to report.

C.2 ALGORITHM WORKFLOW

Algorithm 1 restates the compact HTR procedure referenced in Section 4.3, and Algorithm 2 expands it with implementation-level detail. Notation follows the main paper: ι_n denotes the insight recorded at node n, b_n the git branch reference, s_n the $\mathcal{E}_{\text{dev}}$ score, r_n the factual result, and $\iota_{\text{anc}(n)}$ the concatenated insights on the path from n_0 to n's parent.

C.3 AGENT TOOLS AND HYPERPARAMETER SETTINGS

C.3.1 COORDINATOR TOOLS

The coordinator's tool set reflects a strict division of labor: it may read any file in the repository and inspect the tree in any projection, but it may never edit code directly. All implementation work is delegated to executors via `RunSubagent` or `RunSubagentParallel`. This design is intentional. If the coordinator could edit code directly, it would be tempted to make small local fixes without creating a new tree node, which would break the invariant that every code change is traceable to a hypothesis. Enforcing the edit boundary at the tool level makes this invariant a system property rather than a behavioral expectation.

Tables 5 and 6 list the full tool set. The tree tools (`TreeView`, `TreeAddNode`, `TreeUpdateNode`, `TreePrune`, `TreeSetMeta`, `TreePropagate`) are the primary interface through which the coordinator manages the shared research state. The dispatch tools (`RunSubagent`, `RunSubagentParallel`) are the boundary through which the coordinator hands off implementation to executors and receives structured evidence back. The merge tool (`GitMergeBranch`) is the only path through which a candidate branch can be promoted to the current best, enforcing the

Algorithm 2: Hypothesis Tree Refinement (HTR) with expanded implementation detail. The coordinator owns $\mathcal{T}ree$, and each EXECUTOR owns one worktree.

Input : $\mathcal{P} = (\mathcal{M}_0, \mathcal{O}, \mathcal{E}_{\text{dev}}, \mathcal{E}_{\text{adm}})$, budget B, branching k, parallelism P
Output : best artifact $\mathcal{M}^\star$, annotated hypothesis tree $\mathcal{T}ree$, and run summary report

`// Initialization`
init $\mathcal{T}ree = (\{n_0\}, \emptyset)$, $b_{n_0} \leftarrow \mathcal{M}_0$, $\mathcal{M}_{\text{best}} \leftarrow \mathcal{M}_0$
run $\mathcal{E}_{\text{dev}}(\mathcal{M}_0)$ and record baseline score and eval command in $\mathcal{T}ree$.meta via `TreeSetMeta`
`// Main coordinator loop`
while *B left $\wedge$ pending leaves exist $\wedge$ no stop signal* **do**
  `// Step 1: OBSERVE, build constraint view`
  $\mathcal{V} \leftarrow$ OBSERVE$(\mathcal{T}ree, \mathcal{M}_{\text{best}})$ `// shape, root insight, pruned lessons, validated findings`
  `// Step 2: IDEATE, skill-gated hypothesis proposal`
  $\mathcal{V}_c \leftarrow$ `TreeView(format=constraints)` `// hard constraints: no re-tread of pruned directions`
  load `idea_drafting` skill via `LoadSkill` `// mandatory gate: must precede any candidate proposal`
  $p \leftarrow$ SELECT$_{\text{parent}}(\mathcal{V})$
  foreach *surviving candidate after Fatal-Flaw Scan* **do**
    attach pending child $n^{(i)}$ with hypothesis $h^{(i)}$ = (Mechanism, Hypothesis, Observable, Conflicts)
    $\iota_{\text{anc}(n^{(i)})} \leftarrow$ insights on $\mathsf{path}(n_0 \to p)$ `// injected into Executor prompt`
    prune `idea_drafting` scratch from coordinator context `// elide skill body + reasoning post-TreeAddNode`
  end foreach
  `// Step 3: SELECT frontier for parallel dispatch`
  $L \leftarrow$ up to P pending leaves under SELECT$_{\text{frontier}}(\mathcal{V})$
  `// Step 4: DISPATCH, parallel Executor dispatch`
  $\{(s_n, r_n, \iota_n, b_n)\}_{n \in L} \leftarrow$ **parallel** `Executor`$(h_n, \iota_{\text{anc}(n)}, \mathcal{M}_{\text{best}})$
  `// Step 5: UPDATE, write back and propagate`
  foreach $n \in L$, $a \in \mathsf{path}(n_0 \to n)$ **do**
    write back (s_n, r_n, ι_n, b_n) to node n in $\mathcal{T}ree$ and set n.status $\leftarrow$ `done`
    $\iota_a \leftarrow$ ABSTRACT$(\{\iota_c\}_{c \in \mathsf{ch}(a)})$ `// propagate insights upward to root`
  end foreach
  check convergence: inject intervention if $\geq w$ consecutive non-improving experiments
  `// Step 6: DECIDE, merge gate or prune`
  $n^\dagger \leftarrow \arg\max_{n \in L} s_n$
  if *$s_{n^\dagger}$ exceeds best score by $\geq \theta$* **then**
    create detached worktree at $b_{n^\dagger}$ and run $\mathcal{E}_{\text{adm}}$ with template substitution
    if $S_{\text{adm}}(b_{n^\dagger}) > S_{\text{adm}}(\mathcal{M}_{\text{best}})$ **then** $\mathcal{M}_{\text{best}} \leftarrow \mathsf{merge}(b_{n^\dagger})$ and update $\mathcal{T}ree$.meta.trunk_score
  end if
  prune subtrees falsified by $\{\iota_n\}_{n \in L}$ and persist $\mathcal{T}ree$ to JSON + Markdown
end while
run $\mathcal{E}_{\text{adm}}(\mathcal{M}_{\text{best}})$ and record `test_trunk_score` and `test_baseline_score`
return $\mathcal{M}^\star \leftarrow \mathcal{M}_{\text{best}}$, *$\mathcal{T}ree$, run summary*

Procedure `Executor`$(h_n,\ \iota_{\text{anc}(n)},\ \mathcal{M}_{\text{best}})$ **:**
  branch name $\leftarrow \mathsf{slug}(\text{node_id}) + \mathsf{slug}(h_n) + \text{SHA1}(h_n)_{[:8]}$
  create worktree W_n in `/tmp/` from current best branch HEAD on a fresh branch
  inject eval command (with {`cwd`}$\to W_n$, {`node_id`}$\to n$) and $\iota_{\text{anc}(n)}$ into prompt
  repeat
    $\Delta \leftarrow \mathsf{Implement}(h_n, \iota_{\text{anc}(n)}, W_n)$
    $(s_n, r_n) \leftarrow \mathcal{E}_{\text{dev}}(\mathsf{apply}(\Delta, W_n))$ `// repair` Δ `only, direction` h_n `is fixed`
  until *run ok $\wedge$ h_n-path exercised, or turn cap reached*
  filter Δ: commit implementation files only, skip logs/checkpoints/caches
  remove worktree directory and retain branch b_n for later merge gate
  return $(s_n,\ r_n,\ \text{DISTILL}(h_n, \Delta, r_n),\ b_n)$

admission gate at the tool level rather than relying on the coordinator to remember to verify before merging. Table 7 reports the default configuration.

Table 5: Tools available to the coordinator.

Tool	Description
`TreeView`	Inspect the idea tree in five formats: `compact` (status overview), `full` (Markdown rendering), `node` (single-node detail), `pending` (pending leaf list), and `constraints` (aggregated pruned lessons and validated findings used as hard constraints in the IDEATE phase).
`TreeAddNode`	Add a child node to the tree with a four-field hypothesis block (Mechanism, Hypothesis, Observable, Conflicts). Triggers IDEATE context pruning after each commit.
`TreeUpdateNode`	Update mutable fields of an existing node (`status`, `insight`, `score`, `result`, `code_ref`, `related_work`).
`TreePrune`	Mark a node and its subtree as pruned with a written reason. The reason is aggregated into the constraints block for future IDEATE rounds.
`TreeSetMeta`	Write evaluation metadata to the tree root: evaluation commands (`eval_cmd`, `eval_cmd_test`), dataset paths, baseline score, best score, and admission scores (legacy test fields). Automatically injected into every Executor's prompt.
`TreePropagate`	Re-propagate insights from a node upward to the root after manual corrections via `TreeUpdateNode`.
`RunSubagent`	Dispatch a single Executor to implement and evaluate a pending tree node. Creates an isolated git worktree from trunk HEAD, injects ancestor insights and evaluation metadata, auto-extracts results from the structured report, and updates the tree node.
`RunSubagentParallel`	Dispatch two to four Executors concurrently on independent tree nodes. Each Executor runs in its own isolated worktree.
`GitMergeBranch`	Validate and promote a candidate branch to the current best. Creates a temporary detached worktree at the source branch, runs $\mathcal{E}_{\text{adm}}$ with score extraction, and merges only if the validated admission score exceeds the current best score. Protected branches (main, master) cannot be merge targets.

Table 6: Tools available to the coordinator (continued).

Tool	Description
`Bash`	Execute shell commands for read-only codebase inspection and environment queries. The coordinator never uses Bash to edit code.
`FileRead`	Read the contents of a file within the target repository.
`Grep`	Search for a text pattern across repository files.
`Glob`	Find files matching a glob pattern within the repository.
`LoadSkill`	Load a skill document on demand from the registry. In the IDEATE protocol, `idea_drafting`, `first_principles_probe`, and `fatal_flaw_scan` are loaded before proposing candidates. Loaded content is pruned from context after each `TreeAddNode` commit.
`SearchIdeaContext`	Dispatch a SearchAgent in the background to annotate a tree node with related work. Returns immediately. The SearchAgent runs concurrently with ongoing IDEATE and dispatch work and writes a structured annotation (summary, novelty assessment, related papers) to `node.related_work` when finished.
`SearchIdeaContext-Parallel`	Dispatch multiple background SearchAgents concurrently for a list of node IDs.
`SearchStatus`	Return the count of in-flight background SearchAgent tasks.

C.3.2 Executor Tools

The executor's tool set is designed around a single constraint: every action must be local to its worktree. The executor has no tools for reading the current best, inspecting sibling branches, or modifying the shared tree. This isolation is not merely a safety measure. It is what allows multiple executors to run concurrently on overlapping codebases without coordination overhead. Each executor operates as if it has an exclusive copy of the repository, because its worktree is literally a separate checked-out directory that shares the git object store but maintains its own HEAD and index. The `RunTraining` tool deserves special mention: for experiments that involve multi-hour training runs, using `Bash` with a timeout would either cut the run short or block the executor for an unbounded

Table 7: Key hyperparameters for the `Arbor` coordinator and executors.

Parameter	Default	Description
Backbone model	Claude Opus 4.6	LLM for both coordinator and executors (configurable per role)
Max cycles	20	Hard cap on total completed experiments
Max tree depth	3	Maximum hypothesis-tree depth, with the root at depth 0
Max coordinator turns	500	Maximum ReAct turns for the persistent coordinator loop
Executor max turns	200	Maximum turns per executor
Executor timeout	48 h	Wall-clock limit per executor
Context window	200 K tokens	Coordinator context limit
Compression threshold	0.80	Compress context when it reaches 80% of the context window
Kept recent turns	20	Number of most recent turns preserved after each compression
$\mathcal{E}_{\mathrm{adm}}$ evaluation retries	1	Additional admission evaluation attempts after a transient failure
Convergence window	5	Sliding window size for score velocity computation
Convergence warn / stop	3 / 8	Consecutive non-improving experiments to trigger warning / stop

duration. `RunTraining` instead blocks until the command terminates or the configured timeout is reached, continuously capturing intermediate metrics, progress logs, and checkpoints so that even a timed-out run returns actionable partial evidence. This matters in practice: a training run that completes 80% of its epochs before timing out often reveals enough about a direction's trajectory to inform a principled prune or continuation decision, which is more useful than a silent timeout that leaves the coordinator without evidence.

Table 8: Tools available to each executor.

Tool	Description
`Bash`	Execute shell commands in the worktree with configurable per-call timeouts (default 600 s, maximum 86,400 s).
`RunTraining`	Execute a long-running training or evaluation command and block until completion or timeout. Automatically captures partial metrics, progress logs, and checkpoints. Supports staged budgets (smoke, pilot, full) with configurable wall-times. Maximum timeout is 604,800 s (seven days).
`FileRead`	Read the full contents of a file within the worktree.
`FileEdit`	Edit a file by replacing an exact string match with new content, enabling surgical edits that minimize unintended side effects.
`FileWrite`	Write the complete contents of a file, creating or overwriting it.
`Grep`	Search for a text or regex pattern across files in the worktree.
`Glob`	Find files matching a glob pattern within the worktree.
`SubAgent`	Spawn a nested executor with a custom prompt for modular decomposition of complex implementation tasks. The nested agent inherits the same tool set and worktree context.

C.4 Idea Tree Data Structure and Storage

The implementation stores `Arbor`'s hypothesis tree as an idea-tree object. This object is the primary durable data structure of a run: it records the research frontier, completed experiments, rejected directions, accepted artifacts, and reusable insights in one inspectable state.

Logical schema. The root node anchors the initial artifact and task contract. Depth-1 nodes represent broad research directions, while deeper nodes represent concrete refinements, alternatives, or corrections under their parent. Edges therefore encode hypothesis refinement rather than chronological agent actions. During a run, pending leaves form the executable frontier; completed, merged, and

pruned nodes remain in the tree as evidence for future proposal and selection. Each node carries the fields listed in Table 9.

Table 9: Fields of a single node in the idea tree. The first four fields form the structural skeleton; the remaining fields are populated progressively as the node moves through its status lifecycle.

Field	Type	Description
`id`	string	Hierarchical address encoding position in the tree (e.g. `"ROOT"`, `"1"`, `"1.1"`). The dot-separated prefix gives the unique path from root.
`parent_id`	string | null	Identifier of the parent node. `null` for the root.
`children_ids`	list[string]	Ordered identifiers of child nodes, appended as the coordinator expands the tree during IDEATE.
`depth`	int	Integer depth: 0 = root, 1 = direction node, 2–3 = successively refined hypothesis nodes. The maximum depth is governed by `max_depth`.
`hypothesis`	string	Research hypothesis assigned before execution. Written once by the coordinator during IDEATE and never modified thereafter.
`status`	{pending, running, done, merged, pruned}	Status lifecycle: `pending` → `running` → `done`, then `done` → `merged` if the merge gate passes, or `done` → `pruned` otherwise.
`score`	float | null	Absolute scalar score on $\mathcal{E}_{\text{dev}}$ as reported by the executor. Null until the node has been executed and scored.
`result`	string	Factual record of the experiment outcome, written by the executor as a direct observational report before any interpretation is applied.
`insight`	string	Structured lesson extracted by DISTILL from the outcome, then backpropagated and aggregated at each ancestor up to the root.
`code_ref`	string | null	Git branch name pointing to the implementation artifact produced by the executor. Null if no artifact was committed.

Persistent storage. The idea tree is serialized to JSON after every controlled mutation and serves as the sole authoritative record of research state accumulated during a run. No hypothesis, score, artifact reference, or lesson exists outside this structure. The same state is also rendered to Markdown for dashboards and human inspection. The JSON representation is consumed by tools, while the Markdown view provides a readable account of the research process. Since every mutation is persisted immediately, the run can recover from crashes, agent restarts, and context-window compression without relying on conversational memory.

Agent-facing access. Agents do not edit the raw JSON file directly. The coordinator accesses the tree through controlled read views and mutation tools: compact frontier views, single-node views, pending-leaf views, and the constraint view used before ideation; plus mutations for adding nodes, updating lifecycle fields, pruning subtrees, setting metadata, and propagating insights. Executors receive only the assigned hypothesis, the research contract, and ancestor insights. This access pattern keeps the tree as a shared state object rather than a low-level log of tool traces.

D DETAILS OF THE AO TEST SUITE

Table 10 summarizes the six AO tasks; the following subsections give their full definitions, initial materials, evaluators, and development/admission splits.

Improvement over the initial artifact. For percentage-valued metrics, Table 1 reports the absolute change in percentage points. For steps and loss, it reports $100(S_0 - S)/S_0$, since lower is better. The token-cost analysis separately uses relative improvement for every task: $100(S - S_0)/|S_0|$ when higher is better and $100(S_0 - S)/|S_0|$ when lower is better.

Table 10: Compact summary of the AO tasks. Full task definitions are given in the following subsections.

Type	Task	Initial material	Metric and split
Model Training	Optimizer Design	NanoGPT-Bench; tuned Muon baseline	Steps to target loss (↓); admission averages two seeds
Model Training	Architecture Design	`autoresearch` LLM training codebase	Final loss (↓); admission averages two seeds
Harness Engineering	Terminal-Bench 2.0	Official terminal-agent codebase	Pass rate (↑); 36 dev / 53 admission tasks
Harness Engineering	BrowseComp	Minimal ReAct-style search harness	Accuracy (↑); 40 dev / 300 admission questions
Data Synthesis	Search-Agent Data Synthesis	Hand-designed search-data pipeline	Mean pass gap (↑); 50 dev / 100 admission seeds
Data Synthesis	Math-Reasoning Data Synthesis	Hand-designed math-data pipeline	Mean pass gap (↑); 50 dev / 96 admission problems

D.1 Optimizer Design

NanoGPT-Bench (Jordan & contributors, 2025) Track 3 is a collaborative benchmark for discovering efficient neural network optimizers. Unlike the main NanoGPT speedrun, which minimizes wall-clock time by any means, Track 3 targets *step count*: methods that are slow in wall-clock terms but reach the target loss in fewer steps are valid and desirable. All optimizer designs are evaluated on the same fixed architecture, dataset, and training script, so the only lever available is the optimization algorithm and its hyperparameters.

Baseline. We initialize from the official tuned Muon optimizer setting provided by the benchmark authors. This configuration applies Muon to transformer block weights and AdamW to embeddings, the output projection, and scalar parameters (biases and gains). With these tuned hyperparameters, the baseline reaches a FineWeb validation loss of ≤ 3.28 in 3,325 steps. Selecting a well-tuned official baseline as the starting material is intentional: it places the agent in a regime where naive hyperparameter sweeps are unlikely to yield large gains, thereby testing the agent's ability to identify and implement genuinely novel optimizer improvements rather than harvesting easy wins from an undertuned starting point.

Development and admission evaluation. Agents iterate with the development evaluator throughout the search loop. Candidate optimizers are checked for admission with two evaluation seeds different from the development run, and the reported admission score is their average step count. These seeds support candidate admission during optimization.

Evaluator. The evaluator runs the benchmark's official evaluation script `run_eval.py`. The script launches the training job, monitors validation loss, and terminates as soon as `val_loss` ≤ 3.28 is first reached. The score is the step count at termination, with lower being better. If the target is never reached, a penalty score exceeding 7,000 is assigned. All evaluation runs are executed on four NVIDIA A100-80 GB GPUs using the official benchmark evaluation code.

Agent instruction. The agent is initialized with the following task description.

Optimizer Design Task Instruction

Improve the training-step efficiency of the NanoGPT optimizer on FineWeb. The training script `train_gpt_simple.py` trains a 124 M-parameter GPT-2 on FineWeb using Muon for transformer weights and AdamW for embeddings, output projection, and scalar parameters.

Goal. Minimize the number of training steps required to reach `val_loss` ≤ 3.28. The current baseline achieves this target in 3,325 steps.

Evaluation. Run `python run_eval.py` from the project directory. The script launches training, monitors validation loss, and terminates at the first step where `val_loss` ≤ 3.28. The score is the step count at termination, with lower being better. If the target is never reached, a penalty score exceeding 7,000 is assigned.

Constraints. Only `train_gpt_simple.py` may be modified. The dataset, batch size, and model architecture must remain unchanged. Adjusting `train_steps` in isolation is not a valid improvement. Multiple forward passes per step are not permitted.

D.2 Architecture Design

Architecture Design uses the `autoresearch` benchmark (Karpathy, 2026), a compact LLM pretraining task designed for closed-loop research on a real training codebase. The agent receives a single-file training implementation and must improve the final validation loss under a fixed wall-clock training budget. Unlike Optimizer Design, where the model architecture is fixed and only the optimizer is modified, this task exposes the full training recipe: model shape, attention pattern, initialization, optimizer hyperparameters, learning-rate schedules, batch sizing, and training-loop details may all be changed as long as the benchmark evaluator and data preparation remain fixed.

Baseline. The initial material is the default `autoresearch` repository. The main artifact is `train.py`, a single-GPU decoder-only Transformer pretraining script derived from `nanochat`. Data preparation, tokenization, validation-token selection, and evaluation helpers live outside the editable artifact, primarily in `prepare.py`. The development evaluator uses the fixed validation shard provided by the benchmark. The default model is a roughly 50M-parameter Transformer, and the baseline reports the final validation bits-per-byte after the fixed training run.

Development and admission evaluation. During local experiments, executors use the development evaluator, which runs the training script on the benchmark's fixed prepared data and reports the final validation loss. Admission checks rerun the candidate `train.py` with two evaluation seeds and average the resulting final losses. The reported score uses this admission evaluator, which participates in candidate selection.

Evaluator. The evaluator executes `uv run train.py` from the benchmark repository. At the end of training the script prints a structured summary including `val_bpb`, `peak_vram_mb`, `training_seconds`, `num_steps`, and `mfu_percent`. The primary score is the final `val_bpb`, with lower values better. The training script enforces a 300-second training budget; runs that crash, time out, fail to fit in memory, or do not emit a parseable final `val_bpb` are treated as failed experiments. The agent may use secondary diagnostics such as step count and peak memory to interpret failures, but merge decisions are based on the loss metric.

Agent instruction. The agent is initialized with the following task description.

Architecture Design Task Instruction

Follow `program.md` as the authoritative task specification for the `autoresearch` repository.

Goal. Minimize the validation bits-per-byte (`val_bpb`) reported by `uv run train.py`. Lower is better.

Evaluation. Run `uv run train.py` from the project directory and extract the final `val_bpb`, together with `peak_vram_mb`, `training_seconds`, `num_steps`, and `mfu_percent` for diagnostics. The script has a fixed 5-minute training budget. If a run crashes, runs out of memory, exceeds the allowed wall-clock budget, or does not produce a parseable final loss, treat it as a failed experiment unless there is an obvious implementation bug that can be corrected quickly.

Constraints. Only `train.py` may be modified. Architecture, hyperparameters, optimizer behavior, batch size, model size, attention pattern, initialization, schedules, and training-loop details are in scope. Do not modify `prepare.py`, especially `TIME_BUDGET`, `MAX_SEQ_LEN`, `EVAL_TOKENS`, `make_dataloader`, or `evaluate_bpb`. Do not modify data files, tokenizer artifacts, `pyproject.toml`, `uv.lock`, or install new dependencies.

D.3 TERMINAL-BENCH 2.0

Terminal-Bench 2.0 (Merrill et al., 2026) is a benchmark for evaluating terminal agents on realistic command-line tasks executed inside isolated Docker containers. Tasks span 16 categories, including software engineering, security, scientific computing, data science, games, debugging, and others, across easy, medium, and hard difficulty levels. The full task pool contains 89 tasks.

Baseline. Prior harness-engineering studies may begin from codebase variants that differ in interface extensibility, tool set, or prompt structure, creating comparison confounds. To ensure a fair and reproducible starting point we use the official terminal-agent codebase (`terminus-2`) distributed with the benchmark verbatim, without any modifications and without introducing additional interfaces. The backbone model is GPT-5.5. The agent operates as a ReAct-style loop that issues tmux keystrokes to a persistent terminal session.

Development and admission evaluation. We stratify the 89 tasks by difficulty and sample a 36-task development set and a 53-task admission set. The stratification ensures that both splits have balanced difficulty coverage, preventing a scenario where the agent is evaluated on an atypically easy or hard subset. Local experiments use the development set. Arbor queries the admission set for merge decisions during the search.

Evaluator. The evaluator is the official Harbor evaluation harness distributed with Terminal-Bench 2.0. Both development and test evaluations use 8 concurrent workers. The evaluator reports pass rate as the fraction of tasks solved correctly, with higher being better. The baseline achieves 58.33% on the development set (21 out of 36 tasks).

Agent instruction. The agent is initialized with the following task description.

Terminal-Bench 2.0 Task Instruction

Improve the pass rate of a terminal agent on Terminal-Bench 2.0. The current baseline achieves 58.33% on the 36-task development set using GPT-5.5.

Goal. Maximize pass rate on the development split. Iterate on the development split exclusively. Evaluate the admission test split only after achieving a meaningful improvement on development.

Evaluation. Development: `HARBOR_N_CONCURRENT=8 python3 run_eval.py --data data/dev.json`. Test: `HARBOR_N_CONCURRENT=8 python3 run_eval.py --data data/test.json`. The score is the fraction of tasks solved correctly, with higher being better.

Constraints. Modifiable components include the system prompt, per-task extra instructions, the agent subclass, the base agent and its ReAct loop, response parsers, terminal session management, and new files under the agent or prompts directories. The evaluation harness, task data files, API configuration, and baseline reference record must not be modified.

D.4 BROWSECOMP

BrowseComp (Wei et al., 2025) is a search-agent benchmark for multi-step question answering. The material to be optimized is not the benchmark data itself, but a minimal ReAct-style browsing harness (Yao et al., 2023) that answers BrowseComp questions using search and page-reading tools. The task measures whether an autonomous research agent can improve the control logic around a search agent while leaving the evaluator and question sets fixed.

Baseline. The initial material is a simple ReAct-based search harness centered on `single_agent_gpt.py`. For each question, a `GPTAgent` runs a single ReAct trajectory with a system prompt, issues web-search and page-visit tool calls, and returns a short final answer. The online BrowseComp setting uses `SearchTool` and `VisitTool`; the local-corpus tool variants are available in the codebase but are not the primary path for this benchmark. The baseline uses the simple-evals BrowseComp query template and grader template, with the same model family used for the answering agent and the grader in the reported runs.

Evaluator. The development evaluator is `uv run python run_eval.py --data data/bc_val.jsonl --workers 8 --run-name {node_id}`. The admission evaluator

uses the same entry point with `data/bc_test.jsonl` and a distinct run name. The evaluator sends each question to the harness, collects the final answer, normalizes it through the fixed BrowseComp answer-grading path, and reports accuracy as `correct/total`. Items with repeated execution errors or unparsable final answers are counted as errors and do not contribute to the correct count. The primary metric is accuracy, with higher values better.

Search and tool configuration. The harness exposes two browsing tools to the answering agent. `SearchTool` accepts one or more search queries and returns compact web search results, while `VisitTool` fetches and cleans page content for a selected URL. The tool names and argument schemas are fixed so that alternative harness designs remain compatible with the same ReAct agent interface. Agents may change the system prompt, rollout orchestration, response parsing, candidate aggregation, and other harness code, but the evaluation script, question files, grader template, and reference answers are fixed.

Transfer-test protocol. After optimization on BrowseComp, the resulting search harness is frozen and evaluated on 200 HLE (Phan et al., 2025) questions and 200 DeepSearchQA (Gupta et al., 2026) questions, without additional task-specific optimization. Apart from the target question set, all settings follow BrowseComp, including the model configuration, search/visit tools, inference budget, and answer-production and grading protocol.

Agent instruction. The agent is initialized with the following task description.

BrowseComp Task Instruction

Optimize BrowseComp search-agent accuracy starting from the simple baseline on the current main branch. Do not use prior optimized branches, cached scores, or old result directories as the starting point.

Goal. Maximize answer accuracy on the BrowseComp development split. Use development feedback to propose and evaluate harness changes, and reserve the admission split for merge checks.

Evaluation. Development: `uv run python run_eval.py --data data/bc_val.jsonl --workers 8 --run-name {node_id}`. Test: `uv run python run_eval.py --data data/bc_test.jsonl --workers 8 --run-name {node_id}_test`. The evaluator uses the fixed simple-evals BrowseComp prompt and LLM grader. The score is accuracy, with higher values better.

Constraints. The agent may modify the search harness, including `single_agent_gpt.py`, prompts, agent-control logic, response parsers, rollout strategies, and aggregation code. Do not modify `run_eval.py`, the development or test data files, API configuration, the grader template, or cached reference records.

D.5 Search-Agent Data Synthesis

Search-Agent Data Synthesis is a pipeline-optimization benchmark that evaluates whether a research agent can improve a synthetic question-answering generation system. The system begins from Wikipedia-derived topic seeds and produces BrowseComp-style information-retrieval questions. A high-quality generated item must satisfy three properties simultaneously: structural well-formedness, answerability by a search-enabled model, and sufficient difficulty that the solver does not answer it reliably in a single attempt. The task therefore measures not only whether generated questions are answerable, but also whether they require multi-step evidence gathering rather than surface-level lookup.

Baseline. The initial material is the unmodified default data-synthesis pipeline distributed with the benchmark. The baseline pipeline consists of five stages: extracting candidate facts from a topic seed, rating facts by information value, constructing a question whose answer is one selected fact, obfuscating the topic with supporting constraints, and applying structural verification. The generator, solver, and judge use GPT-5.5 through an OpenAI-compatible interface.

Development and admission evaluation. The benchmark provides frozen seed snapshots for both evaluation splits. The development split contains 50 seeds for local experimental feedback. The admission split contains 100 seeds and is queried during merge checks; its score is used for the reported comparison. The pipeline generates at most one QA item per seed. Fixing the seed snapshots

makes the benchmark self-contained and ensures that measured improvements reflect changes to the synthesis mechanism rather than changes to the data distribution.

Evaluator. The evaluator is the benchmark's fixed `run_eval.py` entry point. It loads the requested seed split, runs the synthesis pipeline, solves each generated item four times with a web-search-enabled ReAct model, and reports a JSON summary on the final line of standard output. The primary metric is:

$$\text{score} = \frac{1}{N}\sum_{i=1}^{N}\big(\text{pass@4} - \text{pass@1}\big),$$

where pass@1 is 1 if the first attempt is judged correct, and pass@4 is 1 if at least one of four attempts is correct. Higher scores indicate questions that remain solvable in principle but are not answered reliably on the first attempt.

Agent instruction. The agent is initialized with the following task description.

Search-Agent Data Synthesis Task Instruction

Improve the quality of generated search-intensive QA items by optimizing the data-synthesis pipeline. The goal metric rewards items that are solvable under repeated attempts but not answered reliably on the first attempt:

$$\text{score} = \text{mean}(\text{pass@4} - \text{pass@1}).$$

Goal. Maximize the score on the development split. Prefer mechanism-level improvements that preserve well-formedness and solvability while increasing multi-step evidence gathering.

Evaluation. Development: `python run_eval.py --split dev`. Test: `python run_eval.py --split test`. Iterate on the development split. Run the test split only after a meaningful improvement.

Constraints. Only the synthesis pipeline under `pipeline/` may be modified, including pipeline steps, the async runner, the step registry, pipeline configuration, and prompt files. The scoring module, evaluation harness, benchmark configuration, and frozen seed snapshots must not be modified.

D.6 Math-Reasoning Data Synthesis

Math-Reasoning Data Synthesis is a pipeline-optimization benchmark that evaluates whether a research agent can improve a system for generating mathematical contest problems. The pipeline starts from fixed AIME/AIMO/NuminaMath-style seeds and produces original problems with exact integer answers in the range 0–999. The research target is the data-construction mechanism itself: generated problems must be valid, novel, diverse, mathematically well specified, and calibrated so that a fixed reference solver does not answer them trivially.

Baseline. The initial material is the unmodified default synthesis pipeline distributed with the benchmark. The pipeline takes each seed, generates several candidate contest problems, filters candidates for answer format, rationale consistency, held-out overlap, and near-duplicate overlap, and evaluates surviving candidates with a fixed solver. Consistent with Table 1, the admission baseline and final score are 1.04 and 20.83, respectively, an absolute gain of 19.79 percentage points.

The default generator uses GPT-5.5-mini and the reference solver uses GPT-5.5. The answer policy is `integer_0_999`.

Development and admission evaluation. The benchmark uses locked seed files for both evaluation splits. The development split contains 10 seeds with 5 generated candidates per seed, yielding up to 50 candidates total. The admission split contains 12 seeds with 8 generated candidates per seed, yielding up to 96 candidates total. Each seed specifies a mathematical topic, technique, and target difficulty. Local experiments use the development split, while milestone admission checks use the second split. The reported score therefore participates in selection.

Evaluator. The evaluator is the fixed `run_eval.py` entry point. It loads the requested split, runs the synthesis pipeline, applies validity and novelty filters, evaluates each surviving candidate with the

Table 11: Token accounting and relative admission gain for completed AO logs. Token counts are in millions.

Task	Baselines				Arbor					Relative admission gain
	Codex		Claude Code		Meta-agent		Sub-agents		Total	
	Input	Output	Input	Output	Input	Output	Input	Output		
Optimizer Design	38.1	0.14	76.0	0.48	6.8	0.07	13.0	0.25	20.12	+2.63%
Architecture Design	501.3	0.42	115.7	2.46	35.0	0.09	7.0	0.17	42.26	+6.38%
Terminal-Bench 2.0	356.7	1.20	34.7	0.13	10.8	0.06	17.2	0.45	28.51	+10.81%
BrowseComp	130.2	0.38	27.3	0.68	8.65	0.03	34.14	0.37	43.19	+49.28%
Search-Agent Data Syn.	403.6	1.40	21.7	0.13	7.8	0.09	15.2	0.24	23.33	+260.00%
Math-Reasoning Data Syn.	129.2	4.70	37.3	0.32	15.0	0.06	20.1	0.39	35.55	+1902.88%

fixed solver, and prints a JSON metrics object on the final line of standard output. The primary metric is:

$$\text{score} = \frac{1}{N}\sum_{i=1}^{N}\big(\text{pass@4} - \text{pass@1}\big),$$

where N is the number of generated candidates before filtering, pass@1 indicates whether the first solver sample is correct, and pass@4 indicates whether at least one of four solver samples is correct. Filtered candidates are treated as zero. This metric rewards problems that are solvable under repeated attempts but not immediately solved on the first sample.

Agent instruction. The agent is initialized with the following task description.

Math-Reasoning Data Synthesis Task Instruction

Improve a pipeline that generates AIME/NuminaMath-style contest problems with exact integer answers in the range 0–999. Generated problems must be valid, novel, diverse, and calibrated so that the fixed reference solver fails on the first sample but succeeds within four samples. The primary metric is $\text{mean}(\text{pass@4} - \text{pass@1})$ over all generated candidates.

Goal. Maximize the primary metric reported on the final standard-output line of the evaluator. Prefer mechanism-level improvements such as structured or parametric generation, programmatic answer computation, and difficulty calibration.

Evaluation. Development: `uv run python run_eval.py --split dev`. Admission (legacy split name `test`): `uv run python run_eval.py --split test`. Iterate on the development split. Run the test split only for milestone or final verification.

Constraints. Modifiable files include `configs/pipeline.yaml`, `prompts/generate_problem.md`, `src/math_synth_bench/baseline.py`, and new modules under `src/math_synth_bench/`. The benchmark harness, seed files, evaluation references, metrics module, and verification module must not be modified.

E Token Statistics

Table 11 reports the token accounting used for Figure 4. All token counts are in millions. For `Arbor`, the meta-agent columns correspond to the persistent coordinator and the sub-agent columns correspond to executor agents; `Arbor` total is the sum of these four columns. The relative admission gain is computed from the admission scores in Table 1: for lower-is-better metrics we use $(S_0 - S_{\texttt{Arbor}})/S_0$, and for higher-is-better metrics we use $(S_{\texttt{Arbor}} - S_0)/S_0$. The six reported tasks use the finalized cost logs from runs subject to the 48-hour cap.

F Full MLE-Bench Lite Leaderboard

Table 2 in the main text lists the most competitive published systems on MLE-Bench Lite. Table 12 reports the complete leaderboard, including the additional baselines omitted there for space. The compared systems are AIDE (Jiang et al., 2025), ML-Master (Liu et al., 2025b) and ML-Master

Table 12: Complete MLE-Bench Lite leaderboard under the official evaluation protocol. All entries are percentages; `Arbor` rows are shaded.

Method	Model	Valid sub.	Above median	Bronze	Silver	Gold	Any medal
InternAgent	DeepSeek-R1	100.00	78.79	10.61	16.67	34.85	62.12
ML-Master	DeepSeek-R1	100.00	74.24	4.55	13.64	30.30	48.48
AIRA-dojo	o3	100.00	70.45	7.95	12.73	34.32	55.00
ML-Master 2.0	DeepSeekV3.2-Spe	100.00	84.85	13.64	31.82	30.30	75.76
R&D-Agent	GPT-5	77.27	74.24	12.12	22.73	33.33	68.18
Famou-Agent 2.0	Gemini-2.5-Pro	100.00	86.36	15.15	19.70	40.91	75.76
MARS	Gemini-3-Pro	100.00	89.39	6.06	15.15	53.03	74.24
Leeroo	Gemini-3-Pro	68.18	68.18	18.18	19.70	30.30	68.18
AIBuildAI	Claude-Opus-4.6	100.00	81.82	13.64	25.76	37.88	77.27
AIDE	Gemini-3-Flash	77.27	54.55	4.55	9.09	31.82	45.45
LoongFlow	Gemini-3-Flash	77.27	77.27	12.12	25.76	39.39	77.27
Codex	GPT-5.5 (xhigh)	100.00	81.82	1.52	19.70	46.97	68.18
AI-Scientist	Gemini-3-Flash	100.00	86.36	18.18	31.82	31.82	81.82
`Arbor`	Gemini-3-Flash	100.00	86.36	13.64	27.27	40.91	81.82
`Arbor`	Claude-Opus-4.6	100.00	90.91	4.55	27.27	50.00	81.82
`Arbor`	GPT-5.5	100.00	95.45	0.00	9.09	77.27	**86.36**

2.0 (Zhu et al., 2026), AIRA-dojo (Toledo et al., 2025), InternAgent (Team, 2026), R&D-Agent (Yang et al., 2025), Famou-Agent 2.0 (Li et al., 2025), MARS (Chen et al., 2026b), Leeroo (Nadafian et al., 2026), AIBuildAI (Zhang et al., 2026), LoongFlow (Wan et al., 2025), and AI-Scientist-style systems (Lu et al., 2024; Chen et al., 2026a). All entries are percentages under the official MLE-Bench evaluation protocol, and the three `Arbor` rows (shaded) use the same controller, depth, scheduler, and adapter, changing only the backbone model; each is a single run. Baseline numbers are adopted from the official MLE-Bench leaderboard and the AI-Scientist report (Chen et al., 2026a).

G Ablation Implementation Details

This section specifies the two ablated variants reported in Table 3. Both are single-mechanism interventions on the full system: the node schema, coordinator and executor tool sets, executor contract, worktree isolation, admission gate, and convergence detector are identical across all three rows. Each variant is one complete run per competition under the official MLE-Bench Lite protocol, using Claude Opus 4.6, a 24-hour per-competition budget, 20 coordinator cycles, and the same `mle_kaggle` adapter.

What each variant changes. *w/o tree* removes the hierarchy only. Every experiment node is attached directly to the root, so the tree has depth 1 instead of the default maximum depth 3. The consequences follow mechanically: there is no ancestor chain, so executors receive the root insight in place of a path-specific insight chain, and the upward synthesis step has a single level to operate on, reducing it to one root-level summary over all completed children. Nodes are still created through `TreeAddNode`, still bind one hypothesis to one worktree branch, and are still pruned through `TreePrune`.

w/o insight feedback removes the upward synthesis only. The hierarchy, node creation, parent selection, and depth limit are unchanged, but the propagation step that synthesizes children's insights into their parent is disabled, so every internal node keeps an empty insight field and executors receive no ancestor insights.

What remains visible in both variants. Neither ablation removes the record of past experiments. When an executor returns, its hypothesis, development score, factual result, distilled insight, and branch reference are written back to its node in both variants, and the coordinator's read projections continue to expose them. In particular, the constraint view used during ideation still lists pruned directions with their recorded lessons and completed nodes with their insights. The variants therefore differ in how evidence is *organized* and *abstracted*—hierarchically in the full system, flat in *w/o tree*,

and unabstracted in *w/o insight feedback*—rather than in whether leaf-level evidence is accessible at all.

Resource matching. The three rows share the same wall-clock budget, cycle cap, backbone, tool set, context window, compression policy, and evaluator commands, and each performs one admission evaluation per merge attempt under the same gate rule. Two quantities are not held equal, because they are consequences of the ablated mechanisms rather than free parameters. First, *w/o insight feedback* issues fewer model calls, since each propagation step is itself a short LLM call; the full system pays this cost and the variant does not. Second, the per-cycle coordinator context differs in composition: the full system's ideation prompt carries synthesized ancestor insights, whereas *w/o insight feedback* carries none and *w/o tree* carries a single root-level summary. We did not additionally pad the ablated variants to equalize context length or call counts, so the reported differences reflect each mechanism together with its own information and compute footprint. This is the intended comparison for asking whether the mechanism is worth its cost, but it does not isolate hierarchy or abstraction from the context exposure each one produces.

H Extended Analysis of Research Traces

We analyze `Arbor`'s internal research traces to understand how autonomous research progresses once the agent starts running experiments, focusing on three questions: how hypotheses change over time (Appendix H.1), when useful improvements appear (Appendix H.2), and what kinds of ideas the Hypothesis Tree produces (Appendix H.3).

H.1 Hypothesis Refinement Analysis

A recorded failure-to-refinement lineage. BrowseComp node `7.1` obtains 60.0% development accuracy with three structured rollouts, below the 65.0% incumbent using five independent rollouts. Its report attributes the decline to reduced retrieval diversity. Parent node `8` explicitly cites this negative finding and proposes sharing answers and evidence while preserving independent second-round retrieval. Node `8.1` implements this proposal and obtains 72.5% $(29/40)$. The records thus connect a rejected implementation to the design constraint that produced the run's best candidate.

We analyze the BrowseComp hypothesis tree, reporting the main hypothesis shifts, the nodes that triggered them, and the final design selected by the merge gate. Figure 7 traces all three contractions of task understanding alongside the experimental nodes that drove each transition. We find that:

Early nodes test whether a broad mechanism holds. The run begins from a coarse hypothesis and uses the first experiments to confirm or reject it. In BrowseComp, the initial hypothesis is that search agents produce near-miss answers by matching salient cues while missing fine-grained constraints; constraint-decomposed verification and hostile-contradiction checking both improve development accuracy, identifying fine-grained answer checking as a productive direction.

Later nodes localize the bottleneck by probing the mechanism's boundary. Once a mechanism shows promise, the tree tests where it stops working rather than pushing it further. In BrowseComp, the verifier nodes can judge candidates produced by the search process but rarely recover candidates that were never surfaced, and part of the hostile-verifier gain comes from answer normalization rather than reliable evidence discovery. This shifts the main design target from stricter verification to broader evidence coverage.

Ancestor insights compress these results into the constraints that shape the final design. The accumulated positive and negative findings define what the successful design must satisfy. In BrowseComp, the evidence-dossier aggregator preserves candidates and supporting evidence across independent rollouts, recovering correct answers that appear in only a minority of trajectories; follow-up nodes then rule out persona-diverse rollouts (which only rerank within the same retrieval frontier), the search-augmented judge (which overfits development questions), and shared decomposition (which reduces trajectory independence). `Arbor` thus learns that BrowseComp benefits from sharing evidence while keeping search trajectories independent.

Takeaway. Hypothesis refinement in HTR is a **deepening of task understanding**: early nodes test broad mechanisms, later nodes identify their limits, and ancestor insights summarize these results

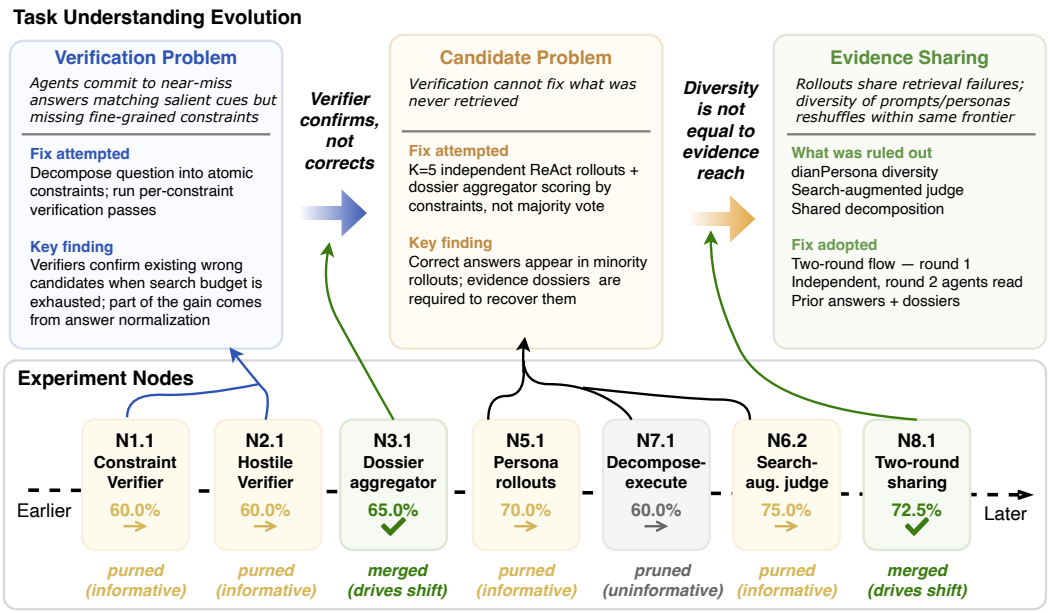

Figure 7: Evolution of task understanding across the BrowseComp run. Each upper-tier box gives the current problem framing, the fix attempted, and the mechanistic finding driving the next shift; the lower tier shows the experimental nodes behind each transition.

into constraints for the next round of proposals. This constraint accumulation is the core process-level benefit over flat trial-and-error.

H.2 Search Efficiency Analysis

We analyze when the best candidates appear during a run, using each node's execution time and development gain; Table 4 summarizes the per-task node counts. We find that:

Strong candidates often appear after the search state has accumulated constraints. Across tasks, `Arbor` frequently reaches its best candidate in the middle or later part of the run. These improvements are supported by earlier nodes that identify useful mechanisms, rule out weak variants, and narrow the design space.

Later proposals are more targeted than early proposals. In BrowseComp, the final evidence-sharing design appears only after several failed or partially successful verifier variants establish that fine-grained checking matters, that candidate coverage is the bottleneck, and that judge-side search and shared decomposition introduce new failure modes. The final proposal is therefore generated from a more informative research state than the initial proposals.

Accumulated evidence informs the proposal distribution over time. `Arbor` does not simply allocate budget uniformly over independent attempts: its later nodes are conditioned on accumulated evidence from ancestors and siblings. Successful mechanisms become priors, failed variants become negative constraints, and partial gains become starting points for refined hypotheses.

Takeaway. Earlier experiments supply explicit constraints for subsequent proposals. The BrowseComp lineage above makes this connection inspectable through explicit references from a later hypothesis to an earlier failed intervention.

H.3 Idea Quality Analysis

We analyze representative ideas generated across model training, harness engineering, and data synthesis tasks, classifying each by its granularity, implementation target, and relation to previous evidence. Figure 8 shows representative examples. We find that:

BrowseComp — harness engineering

- Run independent ReAct rollouts, then select the answer whose evidence dossier covers the most question constraints, rather than taking a majority vote.
- In the second round, share only answer candidates and their evidence dossiers across agents—never plans, personas, or decompositions.

Search-Agent — data synthesis

- Strip entity surface forms before generation so the model cannot copy answer tokens from the question; re-inject obfuscated aliases only at retrieval time.
- After generating a candidate task, have a separate agent attempt it adversarially and discard any task solved on the first try.

Math-Reasoning — data synthesis

- Instantiate each problem family from a parametric template with randomised seeds, so difficulty is controlled at the family level rather than per instance.
- Identify families where pass@4 clusters near 0 or 1 and apply targeted re-sampling to shift them into the target difficulty band.

Architecture Design — model training

- Bracket warmdown schedule length as an independent knob and test three settings before varying other hyperparameters.
- Disable weight tying between embedding and unembedding layers and measure whether the freed capacity is absorbed by attention or the feed-forward stack.

Figure 8: Representative ideas generated by `Arbor` across tasks. Each idea is procedural rather than instance-specific, and grounded in an executable artifact change.

The illustrated ideas are local and executable. In model training, ideas usually modify a specific optimizer component, training recipe, or architecture choice. In harness engineering, they change concrete parts of the agent loop, such as retrieval, aggregation, verification, or context management. In data synthesis, they refine generation, filtering, difficulty calibration, or verification modules. This locality makes each idea easy to implement, evaluate, and attribute to a tree node.

Earlier evidence informs subsequent proposals. The BrowseComp proposals directly respond to earlier observations: the BrowseComp evidence-dossier design follows from the failure mode of verifier-only approaches, and similar patterns appear in data synthesis, where later nodes repair specific weaknesses in difficulty calibration or answer verification. HTR therefore helps convert local failures into new design constraints, so that "half-right" results become the starting point for a more precise hypothesis rather than a reason to abandon the direction.

High-level problem formulation remains important. `Arbor` is strongest when the objective can be improved through a sequence of concrete refinements, and less reliable when progress requires a new high-level formulation weakly connected to the existing tree. The Architecture Design task ultimately acknowledged that single-knob tuning had reached diminishing returns and a larger algorithmic move was needed, but identifying that move still depended on prior judgment rather than anything the tree could automatically generate. This highlights the role of human-provided task design: the initial artifact, evaluator, metric, and search interface shape the kinds of ideas the agent can discover.

Takeaway. As the tree grows, what has been ruled out, validated, and found to have boundary conditions all become priors constraining the next round of proposals. `Arbor`'s ideas are therefore not isolated guesses but local advances relative to a known task understanding. Together with the refinement and timing results above, this paints the complete picture of HTR as a process mechanism that makes autonomous research cumulative: experimental outcomes becoming explicit context for subsequent search.