How Much of a Harness Does a Strong Agent Need for Autonomous ML Engineering?

Anonymous authors
Paper under double-blind review

Abstract

Recent autonomous machine learning engineering (MLE) agents have made significant progress on public leaderboards. Often motivated by progress stagnation over long-horizon cycles and limited Large Language Model (LLM) primitives, modern MLE agents are deployed on top of increasingly elaborate machinery: multi-agent orchestrators, dedicated retrieval subagents, and more. While such harnesses expand, the use of more primitive but improved coding agents — where LLMs have direct access to the execution environment by using `read`, `write`, and `bash` primitives — have not received enough attention in the field. In this paper we find that, under equal time budget and frontier LLM backbone, open-source state-of-the-art harnesses provide no advantages over a single session of a minimal-harness coding agent baseline, pointing to the backbone as the primary driver for performance. Via a series of large-scale systematic ablation studies, we argue that the machinery layers become redundant in the coding agent setting. We conclude that the effort spent elaborating hand-crafted harnesses around strong models yields poor returns for current MLE benchmarks.

1 Introduction

Autonomous machine learning engineering (MLE) has become a demanding testbed for LLM-based agents: solutions are open-ended ML pipelines, a single evaluation can take hours on expensive hardware, and the feedback signal — a validation score — is noisy. Since Jiang et al. (2025) framed the problem as tree search over code, the dominant design has been the *harness*: an outer program that queries an LLM for code, executes it, and orchestrates the exploration and combination of candidate solutions. Motivated by stagnation over long horizons and by the limits of a single LLM call, harnesses have grown steadily more elaborate, adding search trees, populations of programs, memory hierarchies, and teams of specialised agents.

Much of this machinery rests on two premises that deserve reconsideration. First, a single chat completion cannot run the code it writes, inspect the data, or react to an error, so the harness has to do it on its behalf. Modern LLMs, however, are increasingly post-trained for tool use and can operate inside *coding agent* environments such as OpenCode (OpenCode Contributors, 2026), where they read and write files, execute code, and debug within one self-contained session. Second, MLE is treated as a long-horizon problem that calls for dedicated memory and context management. Measured in wall-clock time, up to and beyond 24h, it is. Measured in tokens, it is less clear, now that million-token context windows are affordable. This raises a simple question: which parts of the harness still earn their keep?

The current literature is poorly placed to answer it. Harness performance has risen alongside backbone capability, so gains are hard to attribute to harness design. On MLE-bench (Chan et al., 2025), the standard testbed, published results are almost invariably reported under different backbones and hardware, seed counts rarely exceed three, and headline margins can be smaller than the run-to-run spread (Hambardzumyan et al., 2026; Zhang et al., 2026a; Yang et al., 2026). This is a matter of cost rather than neglect: a single 24h evaluation of three seeds over the 75 tasks already consumes thousands of GPU hours, so faithful re-runs of existing baselines are rare.

Evidence from neighbouring domains suggests the answer may be "very little". Across 35 releases of software-engineering harnesses, Sghaier et al. (2026) find no statistically significant improvement over a fixed backbone on SWE-bench (Jimenez et al., 2024); a single well-prompted agent can match

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.

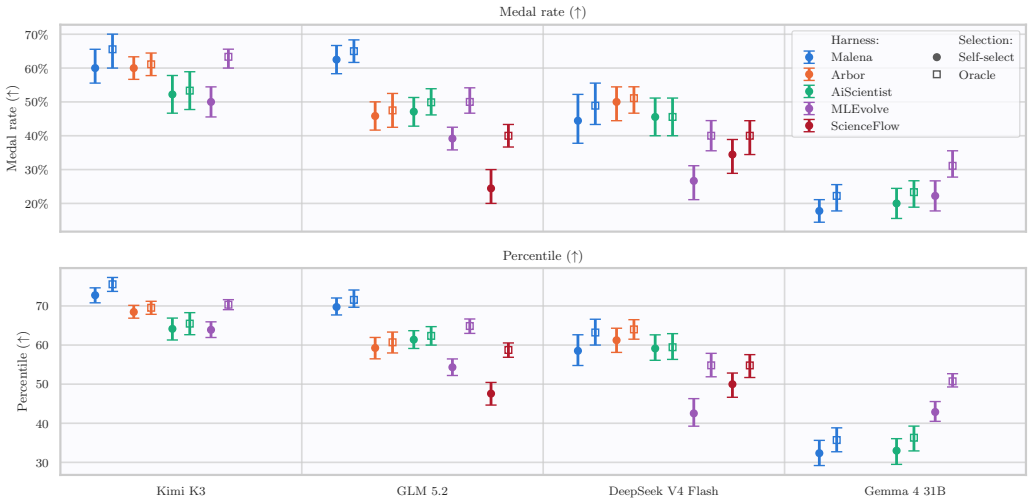

Figure 1: **Performance across backbones and harnesses.** Any-medal rate (↑, top) and mean percentile (↑, bottom), self-select (filled) vs. oracle (open), macro-averaged over the full `fixed30` set at 24h time budget. Brackets are 95% bootstrap CIs. Backbones are grouped along a single shared x-axis; color identifies the harness.

multi-agent workflows (Xu et al., 2026); multi-agent failures stem from coordination rather than capability (Cemri et al., 2025; Orogat et al., 2026); and added orchestration or ideation diversity does not reliably beat plain sampling (Orogat et al., 2026; Audran-Reiss et al., 2025). Coding agents have accordingly become a standard, strong baseline in agentic evaluation (Sghaier et al., 2026; Wang et al., 2026a; Li et al., 2026b; Snell et al., 2024). In MLE, however, this practice has arrived only recently (Jin et al., 2026; Yang et al., 2026; Chen et al., 2026a; Zhao et al., 2026), and the coding agent comparisons reported so far are mostly are sparsely documented.

We evaluate on a harder and broader task set than Lite and, holding backbone, hardware and time budget fixed, climb a ladder of harness interventions re-implemented in a single codebase (Section 4): from a chat-based iteration to multi-agent coding agent orchestration. The end point of this ladder is Malena, the **ma**chine **l**earning **en**gineer **a**gent baseline: a single well-prompted coding agent session with a minimal set of tools. Giving the model a coding agent environment is by far the largest effect we measure; beyond it, no intervention improves the performance significantly. We then test whether this holds for systems built specifically for MLE (Section 5): under matched conditions, Malena matches or outperforms four open-source state-of-the-art harnesses at every frontier backbone we test (Figure 1), on MLE-bench as well as on NatureBench (Wang et al., 2026b). Finally, a trace analysis (Section 5.2) suggests why: the agent carries out the search these harnesses hard-code on its own, balancing rare techniques against performance-oriented refinement while reusing the artifacts it has already produced.

Our contributions are:

- A controlled, budget- and backbone-matched ablation of the intervention families common to MLE harnesses — execution environment, search, autonomy, and multi-agent orchestration — showing that the coding agent environment is the dominant factor and that no further intervention yields a statistically significant gain (Section 4).
- A matched comparison against four open-source MLE harnesses on MLE-bench and NatureBench across multiple backbones, in which Malena is not outperformed at any frontier backbone (Section 5.1). Our results suggest that coding-agent post-training substitutes for the scaffolding harnesses hand-design as weaker backbones still benefit from hand-crafted workflow priors, but as agentic capability grows a coding agent loop closes the gap, echoing the bitter lesson (Sutton, 2019).
- A trace analysis showing that a capable coding agent performs adaptive search without harness-imposed structure (Section 5.2).

2 Related work

Jiang et al. (2025) targets autonomous MLE tasks by using tree search over single-file programs. Subsequent systems follow a common template: an outer harness prompts an LLM for code, and handles execution, scoring and exploration (Yang et al., 2025; Nam et al., 2025; Fang et al., 2025; Du et al., 2025; Kulibaba et al., 2026; Li et al., 2026a; Yuan et al., 2025; Qu et al., 2026; Le et al., 2026; Nadafian et al., 2026; Choi et al., 2026; Meng et al., 2026; Fu et al., 2026). The components these harnesses usually introduce cluster into a small number of claims about what LLMs cannot be trusted to do unaided.

Systems devote their principal contribution to compressing or partitioning accumulated state (Zhu et al., 2026; Zhao et al., 2026; Chen et al., 2026a; Jin et al., 2026). The rest of the interventions generally aid the search topology, either by validation score-driven Monte-Carlo tree search or bandit rules (Jiang et al., 2025; Du et al., 2026; Toledo et al., 2025; Zhang et al., 2026a), or via agentic-driven hypothesis search and multi-agent orchestration (Chen et al., 2026a; Zhao et al., 2026; Jin et al., 2026; Chen et al., 2026b; Qiang et al., 2026). Even under such systems where agents get higher autonomy, the harness still heavily dictates general principles on how the search shall be conducted, limiting adaptability.

A parallel line of work, instead, modifies the backbone itself, training it for MLE with supervised fine-tuning and reinforcement learning (Yang et al., 2026; Cai et al., 2026; Zhou et al., 2026; Qiang et al., 2026; Zhang et al., 2026b), supported by dedicated training and evaluation infrastructure (Qiang et al., 2025; Zou et al., 2026; Lyu et al., 2026). However, such systems are hard to scale, given the cost of collecting 24h traces and training on such trajectories (Zhou et al., 2026; Cai et al., 2026). In this study, we focus on the interventions present in hand-crafted harnesses.

3 Experimental settings

The coding agent environment framework we build our harness from is OpenCode v1.15.6 and our primary benchmark is MLE-bench (Chan et al., 2025), see the task selection in Appendix A.1. When reporting performance of a method that produces several submissions, we differentiate between self-selection: where the submission reported is based on the best validation score reported by the agents, and oracle: where the selection is based on the actual hidden test score obtained. If not specified, the performance reported is based on self-selection. Percentile indicates the percent of submissions in the human leaderboard that score worse than the agent. We report 95% Confidence Intervals (CIs) on macro-task averages, holding the set of tasks fixed and bootstrapping on the seeds available for each task. Unless otherwise stated, we report results using GLM 5.2 backbone LLM, see Appendix A.4 for our full list of models. Across all experiments, agents' resources stay fixed at $1\times$A100 80GB, 12 CPU cores and 144GB of RAM; backbones are served locally with vLLM (Kwon et al., 2023).

4 What interventions actually matter?

In order to study the importance of the harnesses built for MLE systems, we perform a series of systematic ablations to attribute performance impact to individual interventions. We identify four broad families of such interventions and study their effect. Each one isolates a class of intervention employed by contemporary harnesses, re-implemented inside a single codebase so that only the intervention varies. For each intervention, we instantiate the simplest faithful version of the published component rather than any one system's variant, so that an effect is generally attributed to the class of intervention and not to an implementation detail. End-to-end fidelity to specific published systems is deferred to Section 5, where we run those systems as released.

Across all intervention families studied in this work, the introduction of the coding agent environment is the highest-leverage an MLE system can employ. We empirically show how additional machinery provides no statistically significant advantages as we give more autonomy to the coding agent. Furthermore, in Section 5 we argue that every common additional harness intervention can be plausibly dismissed as redundant with a coding agent environment and further hypothesize that the problems such interventions are usually designed to solve become null under the common MLE benchmark setting with modern LLMs.

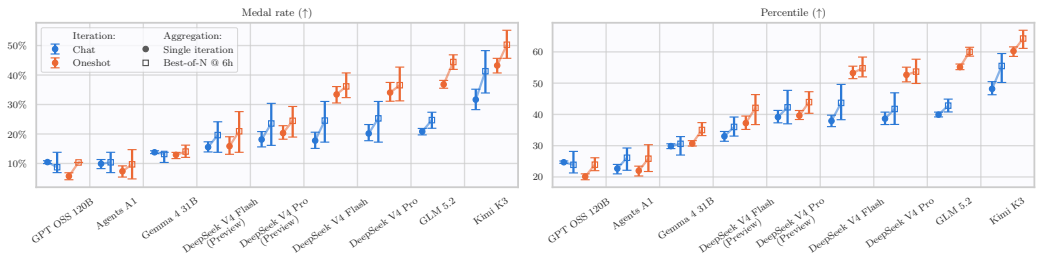

Figure 2: **Impact of a coding agent environment.** Average percentile and medal rate obtained by Chat and Oneshot iterations under different LLM backbones in the `default29` task set. Circled points represent single-iteration performance, while boxes indicate best-of-n performance at six hour budgets.

4.1 Coding agent environment

Many MLE systems call LLMs via a *chat*-like interface, where the LLMs receive a single prompt and can only generate output text, without any direct autonomous interaction of the LLM with the execution environment. In order to make a successful MLE system from this primitive, sophisticated machinery is needed, often including dedicated draft/debug/improve agents and careful handling of context management, retry loops, and hand-crafted data crawler scripts to give the agent enough information about the data to successfully solve the task. On the other hand, the coding agent environment exposes file manipulation and execution primitives to the LLM directly, making it possible to simplify such harnesses. We measure the impact of the coding agent environment by directly comparing the effectiveness of both interfaces. We ablate two different *iterations*, which serve as atomic units of work:

Chat The LLM is called through a chat API with a single problem prompt. The agent is handed the task description and an overview of the data structure and file contents. It must produce a full coding script solution as text in one response. Upon failure of executing the proposed script, the agent is queried with the error stacktrace for debugging.

Oneshot A single coding agent session is spawned. The agent spawns in a working directory containing the data, and is free to use filesystem and code-execution primitives to explore, write, execute and debug. In addition to the ML pipeline, the agent is tasked to document its solution in a markdown file.

Both iterations create a single submission file within a time limit of six hours and their prompt shares a common template, only deviating on instructions specific to their environments. The submission is later graded and the hidden test scores are never communicated back to the agents. More information about iteration implementation in Appendix B.1.

In Figure 2 we show the impact of the coding agent environment for MLE tasks. We observe that the performance of frontier LLMs — GLM 5.2 and Kimi K3 — dramatically improves when they get direct access to the execution environment. Interestingly, even weaker models like Gemma 4 still benefit from such environments, suggesting that tooling fluency becomes more important in such environments than raw model size. This is further evidenced by comparing such improvement in the DeepSeek V4 models, compared to their Preview versions. We also study the impact of time limit budget for Oneshot iterations in and Appendix C.1.

4.2 Search primitives

MLE systems usually drive their search by employing tree or graph search, over a set of ML pipelines to efficiently explore the search space. Previous work (Toledo et al., 2025) has explored the impact of different search algorithms for chat-based AIDE-like systems and found that a careful co-design of search policy and instructions yields an improved performance. We test whether such findings extend to coding agent systems by chaining Oneshot iterations to refine their solutions at a matched 24h budget under different tree search algorithms (details in Appendix B.3):

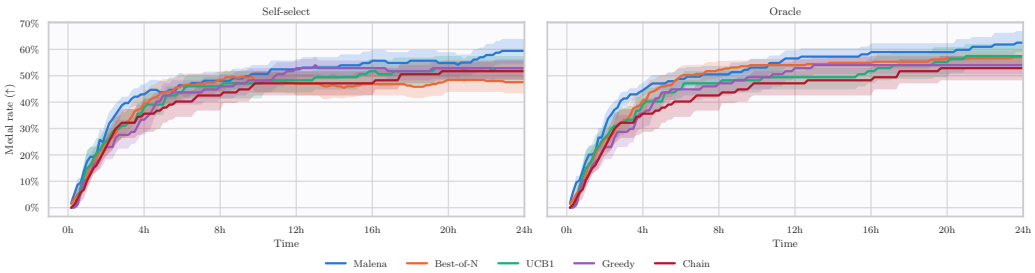

Figure 3: **Performance evolution at 24h budget.** Trajectories are extracted by selecting the best submission available at different times, aggregated in the `default29` task set. Left: self-select medal rate. Right: oracle-best medal rate. Malena achieves the highest medal rate. Chain, Greedy and UCB1's performance remains close to each other.

Chain always picks the most recent node as parent and represents pure iterative improvement.

Greedy picks the node with the best validation score as parent and represents pure exploitation.

UCB1 applies an upper-confidence-bound rule over visits and validation scores and represents the classical explore-exploit trade-off, using standard rank-normalised scores.

Best-of-N starts in an empty tree with no information from previous iterations and represents diverse, completely independent exploration.

We observe in Figure 3, that the search strategy has a very small effect on the performance and the CIs largely overlap. Interestingly, the Best-of-N method provides strong initial performance, and slows down later on, which may indicate a performance ceiling where refinement is needed in order to improve certain tasks. In addition to per-method marginal CIs, we individually study each pair of search methods using a paired-by-task interval. We construct paired-by-task 95% CIs on the performance difference between any two methods. Appendix A.5 provides further information about our statistical protocol, and Appendix C.2 provides detailed results of such experiment.

Under the paired-by-task analysis, no strategy provides any statistically significant advantage neither in percentile, nor medal rate. The pair with the largest gap is Best-of-N and UCB1, where UCB1 shows an average 2.71 percentile points (pp) advantage, with $(-1.44, +6.60)$ CIs. We conclude that when provided with some documentation of previous solutions and tools that facilitate data and execution exploration, the agent can already weigh different approaches and adapt to implicitly perform different searches based on previous results, as we illustrate in Section 5.2.

Finally, we note that Best-of-N's performance exhibits a large loss against its oracle (ceiling) of 6.832pp. This corresponds to the largest selection gap among all explored search strategies. Part of this gap comes from noise and is unavoidable (Hambardzumyan et al., 2026), but strategies to minimize the gap should be prioritized when designing MLE harnesses. Appendix C.3 explores a few mitigation strategies under our controlled environment. We leave an in-depth exploration of more strategies for future work.

4.3 Autonomy

In previous ablations, the search is manually orchestrated by the harness. In this section, we remove such orchestration altogether and arrive at an even simpler baseline, which we label Malena. The Malena iteration is also built on top of coding agent infrastructure, but instead of being prompted to produce a single submission, Malena's single session spans the entire time budget, only being briefly prompted to continue working if a turn ends. This gives the agent freedom to drive exploration, exploitation and knowledge bases autonomously. We build a minimal harness by providing access to a limited set of tools that could potentially help the agent stay grounded as the session grows:[1]

[1]We ablate the importance of such tools in Appendix C.6, and provide further details of them in Appendix B.2.

- `submission` tool: Adds a submission file to the registry. No test or leaderboard scores are communicated to the agent. Note that this tool is irrelevant in Chat and Oneshot iterations, as they produce a single submission and it is added manually to the registry by the harness.

`S` `system` tool: Upon usage, it returns the current hardware resources, utilization, and time remaining. Note that the available resources and time budget are also present in the prompt of all iterations (including Chat and Oneshot).

`J` `jobs` tools: Allows the agent to run multiple bash commands in the background, inspect stdout separately, wait for their completion and cancelling ongoing jobs.

As seen in Figure 3, the Malena iteration reaches the highest average medal rate. Paired-by-task analysis indicates a significant medal rate advantage over all manual search algorithms, between 0.4% and 5.6% depending on the method. Despite having also better average percentile, when compared with UCB1 and Greedy it becomes statistically indistinguishable. Regardless, Malena exhibits the lowest validation gap of all iterations at 1.876pp, presumably as reusing the exact validation pipeline between attempts allows to maintain higher consistency between internal validation scores than tree-search methods. We further explore the effect of Malena under different time budgets in Appendix C.4 and the effect of backbone models in Appendix C.5. One possible drawback from this single-session strategy may arise at the long-horizon regime, where the context overfills the LLM's window size. To test this, in Appendix C.7 we artificially cap Malena's context limit.

4.4 Multi-agent orchestration

MLE harnesses often guide the exploration by leveraging several (potentially parallel) agents. Such agents are typically organized under either a hierarchical structure or a network of peers. We attempt to isolate the effectiveness of such procedures by introducing three coordination interventions:

`D` Delegation. Enables background subagents via the native `task` tool, and encourages their use via a prompt addition. This represents the hierarchical structure of planner-executor.

`P3` Parallelism. When enabled, the harness spawns three independent Malena iterations under the same physical pod. While this allows for a larger amount of submissions and ideas tested, it also risks hardware contention. Parallelism represents the network of peers.

`B` `broadcast` tool. Enabled jointly with parallelism. Allows agents to broadcast messages to their peers, which are then queued and displayed after a turn end.

We present the results in Table 1. The base Malena iteration performs no significantly worse than any intervention tested. While the parallel addition provides a much higher ceiling in oracle-selected percentile score, the selection gap of independent Malena iterations increases under such setting so much that the final self-select submission is of similar quality. This suggests that multi-agent orchestration introduced in previous works is less critical for modern frontier LLMs with coding agent environment capabilities. How to properly coordinate strong agents towards a common goal remains an open question, and we leave a deeper examination for future work.

	Percentile (↑)		Medal rate (%, ↑)	
	Self-select	Oracle	Self-select	Oracle
`base`	66.51 [62.46, 69.97]	69.29 [65.82, 72.13]	**55.7** [48.2, 62.9]	60.4 [53.2, 66.8]
`+D`	63.62 [59.27, 68.26]	67.40 [63.22, 71.98]	45.2 [38.1, 54.8]	47.0 [38.1, 56.0]
`+P3`	**67.77** [64.58, 70.62]	**71.79** [69.34, 73.83]	52.4 [42.8, 61.9]	**60.7** [53.6, 67.3]
`+B+P3`	58.91 [53.50, 64.05]	68.57 [65.77, 71.14]	33.3 [23.8, 40.5]	53.0 [45.8, 60.1]

Table 1: **Performance of Malena under different ablations at 24h budget.** The 95% CI for each metric is reported in brackets. Results on the `fixed14` split at 24h budget. + indicates interventions on top of `base`, − indicates intervention removals.

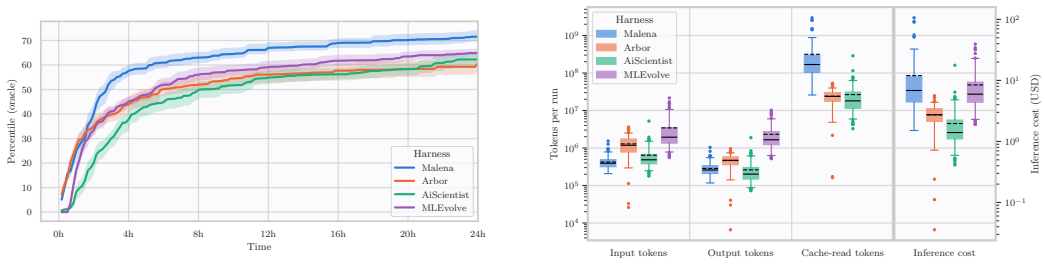

Figure 4: **Left:** oracle best-so-far vs. wall-clock time, macro-averaged over `fixed30` tasks, for different harnesses. **Right:** per-run distribution over all seeds of cumulative input, output and modelled cache-read tokens by the end of each run, plus a fourth group giving the resulting per-run USD cost (right-hand axis). Each box's solid black line indicates median; dashed line indicates mean. MLEvolve's cache is always exactly 0, since its tree-search nodes are independent LLM calls.

5 Generalization to production harnesses

We generalize the conclusions found in Section 4 to a wider selection of open-source state-of-the-art harnesses. We compare our base Malena single long-session loop with MLEvolve (Du et al., 2026), AiScientist (Chen et al., 2026a), Arbor (Jin et al., 2026), and ScienceFlow (Zhao et al., 2026); full descriptions of each baseline, and the full per-backbone MLE-bench comparison table, are in Appendix D. The selection of methods covers a wide range of harness architecture choices from the axes we independently ablated. One of them is representative of the older paradigm, in which the minimal unit of work is a single completion (MLEvolve); the others are more recent and put more emphasis on orchestrating multiple agentic sessions with tools. Malena itself is a single such session, so the comparison is not chat-vs-agent but rather a minimal single-session harness against harnesses that add scaffolding *on top of* agentic sessions or completions. Crucially, all methods are evaluated under the same compute resources, time budget and backbone models.[2] In addition to MLE-bench results, we also report NatureBench (Wang et al., 2026b), a benchmark of open scientific-research tasks that is comparatively more novel and less saturated by prior published baselines than MLE-bench; Appendix A.3 has the full NatureBench benchmark description.

5.1 Results

Table 2, Figure 1, and Figure 4 summarize the headline results: Malena matches or beats every frontier backbone harness pair we compare against, a pattern that also holds on NatureBench. Only at the smaller Gemma 4 31B backbone does the picture change, with MLEvolve ahead by mean estimate, though its medal rate confidence intervals still overlap Malena's. A detailed paired-by-task comparison is in Appendix D.2.

Table 2: **NatureBench results.** Surpassed-SOTA rate (↑, % of tasks where the oracle-best run surpasses the published SOTA), macro-averaged over tasks. A run with no valid submission counts as a fail. Brackets are 95% CIs.

	Surpassed SOTA (%) ↑		
Backbone	Malena	AiScientist	MLEvolve
GLM-5.2	21.7 [15.0, 27.5]	17.5 [12.5, 22.5]	10.8 [7.5, 15.0]
Kimi-K3	26.7 [20.0, 32.5]	27.1 [20.0, 35.0]	11.7 [7.5, 15.0]

The left panel of Figure 4 shows all four methods following a broadly similar quality-vs-wall-clock-time profile: quality rises quickly in the first few hours, plateaus by around the 5-10 hour segment, and further gains afterward arrive mostly as occasional step jumps – when a new best submission lands – rather than smooth continued improvement. AiScientist generally lags behind, except at the ≈4 hour tail, where it quickly recovers performance, and MLEvolve exhibits the slowest plateau.

The right panel of Figure 4 provides critical insights regarding inference cost per method. Even if the total tokens produced remains comparable between different agentic harnesses, the methods' cost

[2]Note that every harness is run at its backbone's maximum reasoning effort, with the exception of MLEvolve on DeepSeek V4 Flash, which runs at medium reasoning effort instead — see Appendix D.3 for more details.

is driven almost entirely by cache-read tokens. At GLM 5.2, Malena's modeled USD cost ($12.12) is ≈6.2x AiScientist's ($1.95), even though its input and output token volumes are comparable to Arbor's and AiScientist's – only its cache-read usage sits roughly an order of magnitude higher. This follows directly from Malena running as a single ever-growing session that resends its whole prior context every turn (the same-session cache model still prices that resent context, even though it credits most of it as a cache hit); Arbor and AiScientist instead reset into fresh, shorter-lived threads per branch or submission, so their modeled cache-read volume never grows as large. Nonetheless, we remark that the overall cost is still dominated by the hardware.[3]

5.2 Trace Analysis

We now examine whether Malena actually exhibits the failure mode that motivates prior designs, and how severely. Of the failure modes such designs target, the one prior designs are built most explicitly around is an inability to reliably *search* the space of ML solutions, so we focus this trace analysis on search. For that we study in-depth traces of Kimi K3 and GLM 5.2 under MLEvolve, AiScientist, Malena and Oneshot iterations.

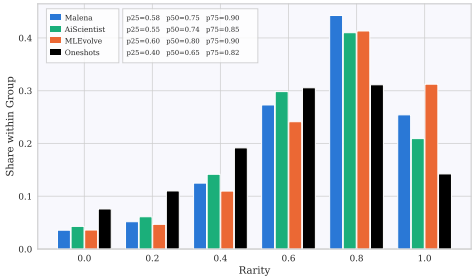

Figure 5: **Per-technique rarity by method**. Rarity of a technique is computed once per task over every method/backbone group's runs combined (see Appendix E.2).

Prior designs typically impose an explicit search structure – a tree over candidate ideas (Arbor, MLEvolve), a population of competing programs, or a stagnation-triggered switch to a fresh idea (ScienceFlow) – built around a discrete unit of work or checkpoint that is either refined or abandoned. Malena's workflow, at GLM 5.2 and Kimi K3, differs qualitatively: it produces a single general pipeline spanning a task's natural stages (e.g., the two-stage embed-then-retrieve pipeline on `whale`; see Appendix E.5), then spends the rest of the run doing hyperparameter search, but over code patches instead of parameter values. This lets it reuse checkpoints and utility code (data loading, preprocessing, validation splits) across iterations, improving efficiency and narrowing the generalization gap.

Going further, we annotate each trajectory's checkpoints with the ML techniques used (see Appendix E for the full labeling methodology) to see how each method searches and what it searches over. Two findings stand out. Methods with LLM-driven search (Malena and AiScientist) start out balanced across the stages of a standard ML pipeline – data preprocessing, feature engineering, architecture, training, and beyond – with AiScientist showing a slight early skew toward model selection, and grow more performance-oriented toward the end of the run, concentrating on ensembling and, at times, techniques borrowed from the Kaggle community such as pseudo-labeling (Appendix E.5 gives a worked example of Malena independently rediscovering this technique under two different backbones); this late-run shift is more pronounced for Malena than for AiScientist, with the shares of ensembling, model selection, and post-processing all increasing up to the very end of the run. MLEvolve, in contrast, settles into a roughly stable technique distribution after about the first 10% of the run and stays there (Figure 6). However, this skew toward performance does not limit Malena to only well-known recipes: Figure 5 shows Malena and MLEvolve are consistently the top two in technique rarity (see Appendix E.2 for how rarity is computed) – the techniques they use tend to be reused by fewer other groups' runs. Malena can therefore keep suggesting comparatively rare techniques while still becoming more performance-oriented late in the run.

Together, these results suggest that a sufficiently capable coding-agent backbone can drive this kind of search largely on its own: Malena balances exploring genuinely rare techniques against a growing performance focus as the run progresses, all while efficiently reusing the checkpoints, utility code, and other artifacts it has already produced rather than starting over – without needing to manually develop a harness for such desired behaviour.

[3]For our hardware configuration, 24h of A100 80GB compute can total between $24 and $85, depending on the provider.

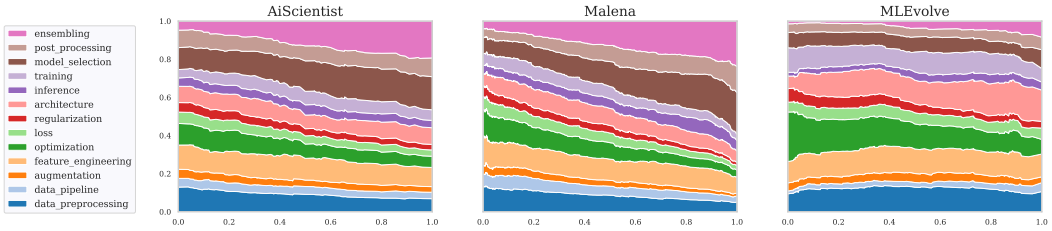

Figure 6: **Category composition of techniques used so far, over the course of a run**, one column per method. The x-axis is checkpoint fraction through the run ($\in [0, 1]$); the y-axis is the decayed share of technique-category tags at that point (stacked, summing to 1). Each technique contributes to a checkpoint's composition with exponential decay by recency, giving a balanced view of the solution's techniques that still favours ones added recently. Per-task and alternate half-life versions are in Appendix E.3.

6 Conclusion

Systems for autonomous machine learning engineering have grown steadily more elaborate: search trees over candidate solutions, populations of competing programs, teams of specialised agents. That machinery was designed when the underlying unit of work was a single model completion, which cannot run the code it writes, remember what it already tried, or react to an error. Coding-focused post-training has since changed the primitive itself: a model strongly capable of autonomous work plans, executes and debugs inside one long-running session, without an outer program telling it what to do next. This paper asks which of the accumulated machinery still earns its keep once the primitive can do these things itself, and answers it by ablating that machinery layer by layer at a fixed model, budget and task set.

Almost all of the gain comes from the runtime and LLM backbone, and almost none from what is built on top of it. Giving the model a shell and a filesystem instead of a chat interface is the single largest effect we measure. Once that is in place, the techniques accumulated by the test-time-compute literature over the last few years, barely move the result as you give more autonomy to the coding agent. A model strongly capable of autonomous work already does internally what were designed to supply from outside, so supplying it again buys little. A single long-running agent session with minimal machinery on top turns out to be a strong baseline, and none of the standard components we added improved on it.

We then confirm the same picture from the outside, against state-of-the-art systems built specifically for this task. Holding the model, hardware, and budget fixed, the plain agent session is not significantly outperformed by neither published harnesses tested with frontier models. With GLM 5.2, Malena earns a medal on 62.5% of the competitions, compared with 47.1% achieved by the best external harness tested. Our statistical analysis across all backbones consistently either favours Malena over other harnesses, or discards any medal rate advantage for either.

The practical reading is that effort spent elaborating harnesses around a model that is already strong at working autonomously has poor returns. The leverage is in the model and in the runtime it is given, not in the scaffolding built around them.

Limitations Most of our results rely on MLE-bench, which has known task-preparation issues and, being built from public Kaggle competitions, is a plausible contamination target; we mitigate this by fixing broken tasks, directly checking for contamination, and additionally evaluating on NatureBench. A second limitation is statistical power: some results still have confidence intervals wide enough to preclude strong claims about effect size, though we believe the weaker *no worse than* claims are well supported. A third is scope: most of the harness interventions tested are within a single worker, and we leave more sophisticated coordination protocols between agents to be explored more in-depth for future work. A fourth is possible asymmetry in tuning effort between Malena and the external harnesses it is compared against; Appendix D.3 documents the backbone-specific bug fixes and configuration ablations we invested in each external harness to mitigate this. Full discussion in Appendix G.

AI USE STATEMENT

In this work, we used generative AI tools, accessed through coding agent environments, to help with implementing code, running and monitoring experiments, and drafting and polishing the paper's text. We have reviewed all AI-assisted work and take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work does not involve human subjects, personally identifiable information, or crowd-sourced annotation. Both benchmarks we evaluate on draw from public, non-sensitive sources: MLE-bench (Chan et al., 2025) from public Kaggle machine learning competitions, and NatureBench (Wang et al., 2026b) from published Nature-family research questions. Where a task's license was restrictive, we substituted an alternative task rather than proceed (Appendix A.1).

Because this work centers on autonomous coding agents with direct `read`/`write`/`bash` access to an execution environment, we bound their side effects with standard safety precautions: agents run inside isolated, resource-limited containers and are restricted to a short whitelist of shell commands (Appendix B.2).

REPRODUCIBILITY STATEMENT

Full experimental settings are described in the Experimental Settings section and in Appendix B: the coding agent framework and tool set exposed to the model in each iteration (Appendix B.2), the search strategies compared (Appendix B.3), and the exact LLM backbones and hosting configuration used, with their HuggingFace identifiers (Appendix A.4). The task sets used for both benchmarks are listed in full in Appendix A.1 (MLE-bench, including which tasks we fixed to remove data or ground-truth-leakage issues, and why) and Appendix A.3 (NatureBench); the exact code diffs we applied to fix the affected MLE-bench preparation scripts are given in Appendix A.2. Our statistical methodology — how macro-task averages, bootstrap confidence intervals, and paired-by-task comparisons are computed — is described in Appendix A.5.

REFERENCES

Alexis Audran-Reiss, Jordi Armengol-Estapé, Karen Hambardzumyan, Amar Budhiraja, Martin Josifoski, Edan Toledo, Rishi Hazra, Despoina Magka, Michael Shvartsman, Parth Pathak, Justine T Kao, Lucia Cipolina-Kun, Bhavul Gauri, Jean-Christophe Gagnon-Audet, Emanuel Tewolde, Jenny Zhang, Taco Cohen, Yossi Adi, Tatiana Shavrina, and Yoram Bachrach. What does it take to be a good ai research agent? studying the role of ideation diversity, 2025. URL https://arxiv.org/abs/2511.15593.

Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2-3):235–256, May 2002.

Lei Bai, Zongsheng Cao, Yang Chen, Zhiyao Cui, Shangheng Du, Yue Fan, Shiyang Feng, Zijie Guo, Haonan He, Liang He, Xiaohan He, Shuyue Hu, Yusong Hu, Songtao Huang, Yichen Jiang, Hao Li, Xin Li, Dahua Lin, Weihao Lin, Fenghua Ling, Dongrui Liu, Zhuo Liu, Wenjie Lou, Runmin Ma, Chunjiang Mu, Haoyang Peng, Tianshuo Peng, Jinxin Shi, Luohe Shi, Boyuan Sun, Zelin Tan, Shengji Tang, Yan Teng, Qianyi Wang, Xiaosong Wang, Yiming Wu, Yi Xie, Xiangchao Yan, Jingqi Ye, Peng Ye, Fangchen Yu, Jiakang Yuan, Bihao Zhan, Bo Zhang, Chen Zhang, Shufei Zhang, Shuaiyu Zhang, Wenlong Zhang, Yiqun Zhang, Junpeng Zhao, Zhijie Zhong, Bowen Zhou, and Yuhao Zhou. Scaling the horizon, not the parameters: Reaching trillion-parameter performance with a 35b agent, 2026. URL https://arxiv.org/abs/2606.30616.

Yuzhu Cai, Zexi Liu, Xinyu Zhu, Cheng Wang, Yanfeng Wang, and Siheng Chen. Acegrpo: Adaptive curriculum enhanced group relative policy optimization for autonomous machine learning engineering, 2026. URL https://arxiv.org/abs/2602.07906.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E.

Gonzalez, and Ion Stoica. Why do multi-agent llm systems fail?, 2025. URL https://arxiv.org/abs/2503.13657.

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 Mądry. Mle-bench: Evaluating machine learning agents on machine learning engineering, 2025. URL https://arxiv.org/abs/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, 2026a. URL https://arxiv.org/abs/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, 2026b. URL https://arxiv.org/abs/2602.02660.

Daewon Choi, Kyunghyun Park, Woomin Song, Saket Dingliwal, Sai Muralidhar Jayanthi, Jinwoo Shin, and Aram Galstyan. Idlespec: Exploiting idle time via speculative planning for llm agents, 2026. URL https://arxiv.org/abs/2605.22154.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

Shangheng Du, Xiangchao Yan, Dengyang Jiang, Jiakang Yuan, Yusong Hu, Xin Li, Liang He, Bo Zhang, and Lei Bai. Automlgen: Navigating fine-grained optimization for coding agents, 2025. URL https://arxiv.org/abs/2510.08511.

Shangheng Du, Xiangchao Yan, Jinxin Shi, Zongsheng Cao, Shiyang Feng, Zichen Liang, Boyuan Sun, Tianshuo Peng, Yifan Zhou, Xin Li, Jie Zhou, Liang He, Bo Zhang, and Lei Bai. Mlevolve: A self-evolving framework for automated machine learning algorithm discovery, 2026. URL https://arxiv.org/abs/2606.06473.

Haoyang Fang, Boran Han, Nick Erickson, Xiyuan Zhang, Su Zhou, Anirudh Dagar, Jiani Zhang, Ali Caner Turkmen, Cuixiong Hu, Huzefa Rangwala, Ying Nian Wu, Bernie Wang, and George Karypis. Mlzero: A multi-agent system for end-to-end machine learning automation, 2025. URL https://arxiv.org/abs/2505.13941.

Shaokang Fu, Yulong Tao, Linbo Jin, Jiarong Zhao, Qiming Shi, Tianjun Pan, Haonan Li, Chengyu Wang, Jia Wu, and Chengfu Huo. Beyond solution-centric search: Adaptive inquiry and knowledge revision for autonomous ml engineering, 2026. URL https://arxiv.org/abs/2608.02143.

Gemma Team. Gemma 4 technical report, 2026. URL https://arxiv.org/abs/2607.02770.

GLM-5-Team, Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, Chenzheng Zhu, Congfeng Yin, Cunxiang Wang, Gengzheng Pan, Hao Zeng, Haoke Zhang, Haoran Wang, Huilong Chen, Jiajie Zhang, Jian Jiao, Jiaqi Guo, Jingsen Wang, Jingzhao Du, Jinzhu Wu, Kedong Wang, Lei Li, Lin Fan, Lucen Zhong, Mingdao Liu, Mingming Zhao, Pengfan Du, Qian Dong, Rui Lu, Shuang-Li, Shulin Cao, Song Liu, Ting Jiang, Xiaodong Chen, Xiaohan Zhang, Xuancheng Huang, Xuezhen Dong, Yabo Xu, Yao Wei, Yifan An, Yilin Niu, Yitong Zhu, Yuanhao Wen, Yukuo Cen, Yushi Bai, Zhongpei Qiao, Zihan Wang, Zikang Wang, Zilin Zhu, Ziqiang Liu, Zixuan Li, Bojie Wang, Bosi Wen, Can Huang, Changpeng Cai, Chao Yu, Chen Li, Chengwei Hu, Chenhui Zhang, Dan Zhang, Daoyan Lin, Dayong Yang, Di Wang, Ding Ai, Erle Zhu, Fangzhou Yi, Feiyu Chen, Guohong Wen, Hailong Sun, Haisha Zhao, Haiyi Hu, Hanchen Zhang, Hanrui Liu, Hanyu Zhang, Hao Peng, Hao Tai, Haobo Zhang, He Liu, Hongwei Wang, Hongxi Yan, Hongyu Ge, Huan Liu, Huanpeng Chu, Jia'ni Zhao, Jiachen Wang, Jiajing Zhao, Jiamin Ren, Jiapeng Wang, Jiaxin Zhang, Jiayi Gui, Jiayue Zhao, Jijie Li, Jing An, Jing Li, Jingwei Yuan, Jinhua Du, Jinxin Liu, Junkai Zhi, Junwen Duan, Kaiyue Zhou, Kangjian Wei, Ke Wang, Keyun Luo, Laiqiang Zhang, Leigang Sha, Liang Xu, Lindong Wu, Lintao Ding, Lu Chen, Minghao Li, Nianyi Lin, Pan Ta, Qiang Zou, Rongjun Song, Ruiqi Yang, Shangqing Tu, Shangtong Yang, Shaoxiang Wu, Shengyan Zhang, Shijie Li, Shuang Li, Shuyi Fan, Wei Qin, Wei Tian, Weining Zhang, Wenbo Yu, Wenjie Liang, Xiang

Kuang, Xiangmeng Cheng, Xiangyang Li, Xiaoquan Yan, Xiaowei Hu, Xiaoying Ling, Xing Fan, Xingye Xia, Xinyuan Zhang, Xinze Zhang, Xirui Pan, Xu Zou, Xunkai Zhang, Yadi Liu, Yandong Wu, Yanfu Li, Yidong Wang, Yifan Zhu, Yijun Tan, Yilin Zhou, Yiming Pan, Ying Zhang, Yinpei Su, Yipeng Geng, Yong Yan, Yonglin Tan, Yuean Bi, Yuhan Shen, Yuhao Yang, Yujiang Li, Yunan Liu, Yunqing Wang, Yuntao Li, Yurong Wu, Yutao Zhang, Yuxi Duan, Yuxuan Zhang, Zezhen Liu, Zhengtao Jiang, Zhenhe Yan, Zheyu Zhang, Zhixiang Wei, Zhuo Chen, Zhuoer Feng, Zijun Yao, Ziwei Chai, Ziyuan Wang, Zuzhou Zhang, Bin Xu, Minlie Huang, Hongning Wang, Juanzi Li, Yuxiao Dong, and Jie Tang. Glm-5: from vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763.

Karen Hambardzumyan, Nicolas Baldwin, Edan Toledo, Rishi Hazra, Michael Kuchnik, Bassel Al Omari, Thomas Simon Foster, Anton Protopopov, Jean-Christophe Gagnon-Audet, Ishita Mediratta, Kelvin Niu, Michael Shvartsman, Alisia Lupidi, Alexis Audran-Reiss, Parth Pathak, Tatiana Shavrina, Despoina Magka, Hela Momand, Derek Dunfield, Nicola Cancedda, Pontus Stenetorp, Carole-Jean Wu, Jakob Nicolaus Foerster, Yoram Bachrach, and Martin Josifoski. Aira_2: Overcoming bottlenecks in ai research agents, 2026. URL https://arxiv.org/abs/2603.26499.

N Hansen and A Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evol. Comput.*, 9(2):159–195, 2001.

Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. Aide: Ai-driven exploration in the space of code, 2025. URL https://arxiv.org/abs/2502.13138.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Jiajie Jin, Yuyang Hu, Kai Qiu, Qi Dai, Chong Luo, Guanting Dong, Xiaoxi Li, Tong Zhao, Xiaolong Ma, Gongrui Zhang, Zhirong Wu, Bei Liu, Zhengyuan Yang, Linjie Li, Lijuan Wang, Hongjin Qian, Yutao Zhu, and Zhicheng Dou. Toward generalist autonomous research via hypothesis-tree refinement, 2026. URL https://arxiv.org/abs/2606.11926.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, M. C., Jianfeng Cai, Xinyuan Cai, Peizhou Cao, Yuxuan Cao, Ziwei Chai, Y. Charles, H. S. Che, Guanduo Chen, Guangyu Chen, Guanzheng Chen, Huarong Chen, Jia Chen, Jianlong Chen, Jun Chen, Kexin Chen, Peng Chen, Ruijue Chen, Wentao Chen, Xin Chen, Yang Chen, Yanru Chen, Yifei Chen, Yingjiang Chen, Yuankun Chen, Yujie Chen, Yutian Chen, Zhirong Chen, Dazhi Cheng, Yean Cheng, Jialei Cui, Jingbing Cui, Anqi Dai, Jiaqi Deng, Hao Ding, Rui Ding, Shaofeng Ding, Mengfan Dong, Mengnan Dong, Yuhao Dong, Yuxin Dong, Angang Du, Chenzhuang Du, Dikang Du, Jusen Du, Yulun Du, Yu Fan, Jing Feng, Qiulin Feng, Yichen Feng, Kelin Fu, Qiang Fu, Fuxuan Gao, Hongcheng Gao, Jingyue Gao, Tong Gao, Weijia Gao, Shangyi Geng, Jie Gong, Linhu Gong, Shengao Gong, Xiaochen Gong, Qizheng Gu, Yicheng Gu, Shuhao Guan, Haiqing Guo, Shiqi Guo, Xiang Guo, Zhengyan Guo, Beixi Hao, Wenxin Hao, Xiaoru Hao, Dailan He, Haotian He, Lehan He, Qi He, Weiran He, Xinran He, Xinyi He, Yibo He, Yunjia He, Chao Hong, Tiange Hong, Hao Hu, Jiaxi Hu, Ruikun Hu, Weiming Hu, Yangyang Hu, Zhenxing Hu, Liang Hua, Jinbin Huang, Ke Huang, Ruiyuan Huang, Siying Huang, Weixiao Huang, Yan Huang, Zhengjie Huang, Zhiqi Huang, Yulong Hui, Chaobo Jia, Yutong Jiang, Zhejun Jiang, Zuoyou Jiang, Wenyi Jin, Xinyi Jin, Yu Jing, Huanjun Kong, Guokun Lai, Aidi Li, Cheng Li, Chengyuan Li, Cong Li, Fang Li, Guanyu Li, Haoyang Li, Jia Li, Junxiong Li, Lei Li, Letian Li, Lincan Li, Weihong Li, Wentao Li, Xintong Li, Yang Li, Yishen Li, Yiwei Li, Yuxiao Li, Zhaowei Li, Zhaoxi Li, Zheming Li, Zhengxiao Li, Zhiyuan Li, Jiawei Lin, Xiaohan Lin, Yibo Lin, Zichao Lin, Ziyan Lin, Bill Liu, Boxiao Liu, Chuan Liu, Liang Liu, Shaowei Liu, Shudong Liu, Shuran Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yanming Liu, Yibo Liu, Yipeng Liu, Zhengying Liu, Zhiheng Liu, Enzhe Lu, Haoyu Lu, Linqiang Lu, Tingzhan Lu, Zhiyuan Lu, Aotian Luo, G. Luo, Junyu Luo, Yifan Luo, B. Lyu, Wenzhou Lyu, Shaoguang Mao, Yuan Mei, Xin Men, Minqing Ni, Yixuan Niu, Siyuan Pan, Shujun Peng, Zhangyang Qi, Ruoyu Qin, ZeChao Qin, Zeyu Qin, Haiquan Qiu, Jianxin Qiu, Jiezhong Qiu, Bowen Qu, Yuhao Qu, Zeyu Shang, Youbo Shao, Han Shen, Jincheng Shi, Juanfeng Shi, Lidong Shi, Shengyuan Shi, Wingchun Siu, Pengwei Song, Xiaoxi Song, Jianlin Su, Yunfeng Su, Zhaochen Su, Lin Sui, Jingsong Sun, Junyao Sun, Shaoning Sun, Shuzhe Sun, Tongyu Sun,

Yujun Sun, Yunpeng Tai, Chuning Tang, Heyi Tang, Sirui Tang, Zecheng Tang, Chaoran Tian, Rongpeng Tian, Yu Tian, Wei Tu, Chensi Wang, Chuang Wang, Chunjie Wang, Dinglu Wang, Feng Wang, Hailong Wang, Haiming Wang, Hao Wang, Hao Wang, Huaqing Wang, Hui Wang, Jiayi Wang, Jinglong Wang, Jinhong Wang, Jiuzheng Wang, Linian Wang, Shaobo Wang, Shenzhi Wang, Shuyi Wang, Si Wang, Siyuan Wang, Tianfu Wang, Wenjue Wang, Xingran Wang, Xinmei Wang, Xinyuan Wang, Xusheng Wang, Yalin Wang, Yangkun Wang, Yao Wang, Yaoyu Wang, Yejie Wang, Yiqin Wang, Yucheng Wang, Yuzhi Wang, Zhaoji Wang, Zhaowei Wang, Zhengtao Wang, Zhenhao Wang, Zhongsheng Wang, Zifan Wang, Chu Wei, Ming Wei, Shouxin Wei, Zichen Wen, Fan Wu, Haoning Wu, Rucong Wu, Wenhao Wu, Xiaoxue Wu, Yingcong Wu, Yongqi Wu, Yuxin Wu, Zijian Wu, Xinglang Xian, Chenxuan Xiang, Yuye Xiang, Bocheng Xiao, Chenjun Xiao, Xin Xiao, Jin Xie, Xiaotong Xie, Yifeng Xie, Zhe Xie, Bowei Xing, Yiming Xiong, Baosheng Xu, Boyu Xu, Jiale Xu, Jianfan Xu, Jing Xu, Jinjing Xu, L. H. Xu, Qingtao Xu, Shuyao Xu, Suting Xu, Tiantian Xu, Tianxiang Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ye Xu, Yueni Xu, Ziyao Xu, Haonan Xue, Junjie Yan, Yaoyao Yan, Fan Yang, Guangyao Yang, Hao Yang, Junwei Yang, Ruoyu Yang, Wenjie Yang, Xiaofei Yang, Xinyu Yang, Yi Yang, Yiling Yang, Ying Yang, Yuchen Yang, Zhen Yang, Zhilin Yang, Zian Yang, Zuhao Yang, Haotian Yao, Dan Ye, Haoran Ye, Wenjie Ye, Zhanbo Ye, Bohong Yin, Haoxiang Yin, Xietong Yin, Chengzhen Yu, Haozhen Yu, Longhui Yu, Shengnan Yu, Shuying Yu, Tianxiang Yu, Enming Yuan, Mengjie Yuan, Tongtian Yue, Wei Yue, Yang Yue, Dunyuan Zha, Haobing Zhan, B. H. Zhang, Dehao Zhang, Fei Zhang, Hao Zhang, Haoyuan Zhang, Huanyu Zhang, Jiapei Zhang, Jiaxuan Zhang, Jin Zhang, Kaiyi Zhang, Miaozhen Zhang, Puqi Zhang, Qinglei Zhang, Rong Zhang, Rui Zhang, Shaoshuai Zhang, Shiyi Zhang, Xiaobin Zhang, Xiaoyun Zhang, Y. Zhang, Yangkun Zhang, Ye Zhang, Yichi Zhang, Yikun Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Zijing Zhang, Bin Zhao, Chenguang Zhao, Feifan Zhao, Jinglun Zhao, Jinxiang Zhao, Shuai Zhao, Wenshuo Zhao, Xiangyu Zhao, Xuanle Zhao, Yikai Zhao, Zijia Zhao, Haozhi Zheng, Huabin Zheng, Ruihan Zheng, Shaojie Zheng, Tengyang Zheng, Haofeng Zhong, Lei Zhong, Longguang Zhong, M. Zhou, Qiankang Zhou, Runjie Zhou, Ruozhang Zhou, Xinyu Zhou, Yiqiao Zhou, Zaida Zhou, Jinguo Zhu, Liya Zhu, Xinhao Zhu, Yangjunfeng Zhu, Yuxuan Zhu, Zhen Zhu, Chen Zhuang, Weiyu Zhuang, and Xinxing Zu. Kimi k3: Open frontier intelligence, 2026. URL https://arxiv.org/abs/2607.24653.

Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *Lecture Notes in Computer Science*, Lecture Notes in Computer Science, pp. 282–293. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

Stepan Kulibaba, Artem Dzhalilov, Roman Pakhomov, Oleg Svidchenko, Alexander Gasnikov, and Aleksei Shpilman. Kompeteai: Accelerated autonomous multi-agent system for end-to-end pipeline generation for machine learning problems, 2026. URL https://arxiv.org/abs/2508.10177.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Dat Le, Duc-Cuong Le, Anh-Son Nguyen, Tuan-Dung Bui, Thu-Trang Nguyen, Son Nguyen, and Hieu Dinh Vo. iml: Executable, problem-grounded, and broadly exploratory code-driven automl, 2026. URL https://arxiv.org/abs/2602.13937.

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, 2026a. URL https://arxiv.org/abs/2510.26144.

Xiaochuan Li, Ryan Ming, Pranav Setlur, Abhijay Paladugu, Andy Tang, Hao Kang, Shuai Shao, Rong Jin, and Chenyan Xiong. Benchmark test-time scaling of general llm agents, 2026b. URL https://arxiv.org/abs/2602.18998.

Bohan Lyu, Yucheng Yang, Siqiao Huang, Jiaru Zhang, Qixin Xu, Xinghan Li, Xinyang Han, Yicheng Zhang, Huaqing Zhang, Runhan Huang, Kaicheng Yang, Zitao Chen, Wentao Guo, Junlin Yang, Xinyue Ai, Wenhao Chai, Yadi Cao, Ziran Yang, Kun Wang, Dapeng Jiang, Huan ang

Gao, Shange Tang, Chengshuai Shi, Simon S. Du, Max Simchowitz, Jiantao Jiao, Dawn Song, and Chi Jin. Mls-bench: A holistic and rigorous assessment of ai systems on building better ai, 2026. URL https://arxiv.org/abs/2605.08678.

Rui Meng, Bhavana Dalvi Mishra, Jiefeng Chen, Chun-Liang Li, Palash Goyal, Mihir Parmar, Yiwen Song, Yale Song, Rajarishi Sinha, Parthasarathy Ranganathan, Burak Gokturk, Jinsung Yoon, and Tomas Pfister. Scientistone: Towards human-level autonomous research via chain-of-evidence, 2026. URL https://arxiv.org/abs/2605.26340.

Alireza Nadafian, Alireza Mohammadshahi, and Majid Yazdani. Kapso: A knowledge-grounded framework for autonomous program synthesis and optimization, 2026. URL https://arxiv.org/abs/2601.21526.

Jaehyun Nam, Jinsung Yoon, Jiefeng Chen, Jinwoo Shin, Sercan Ö. Arık, and Tomas Pfister. Mle-star: Machine learning engineering agent via search and targeted refinement, 2025. URL https://arxiv.org/abs/2506.15692.

OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

OpenCode Contributors. OpenCode, 2026. URL https://github.com/anomalyco/opencode.

Abdelghny Orogat, Ana Rostam, and Essam Mansour. Architectural design, not only model intelligence, governs multi-agent llm performance, 2026. URL https://arxiv.org/abs/2602.03128.

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, 2025. URL https://arxiv.org/abs/2505.07782.

Rushi Qiang, Changhao Li, Haotian Sun, Yuchen Zhuang, Chao Zhang, and Bo Dai. Matryoshka agent: Unfolding sub-agents for long-horizon machine learning engineering, 2026. URL https://arxiv.org/abs/2607.25090.

Ao Qu, Han Zheng, Zijian Zhou, Yihao Yan, Yihong Tang, Shao Yong Ong, Fenglu Hong, Kaichen Zhou, Chonghe Jiang, Minwei Kong, Jiacheng Zhu, Xuan Jiang, Sirui Li, Cathy Wu, Bryan Kian Hsiang Low, Jinhua Zhao, and Paul Pu Liang. Coral: Towards autonomous multi-agent evolution for open-ended discovery, 2026. URL https://arxiv.org/abs/2604.01658.

Oussama Ben Sghaier, Hao Li, Bram Adams, and Ahmed E. Hassan. Don't blame the large language model: How agent harness evolution shapes coding agent quality, 2026. URL https://arxiv.org/abs/2607.03691.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, Akshay Nathan, Alan Luo, Alec Helyar, Aleksander Madry, Aleksandr Efremov, Aleksandra Spyra, Alex Baker-Whitcomb, Alex Beutel, Alex Karpenko, Alex Makelov, Alex Neitz, Alex Wei, Alexandra Barr, Alexandre Kirchmeyer, Alexey Ivanov, Alexi Christakis, Alistair Gillespie, Allison Tam, Ally Bennett, Alvin Wan, Alyssa Huang, Amy McDonald Sandjideh, Amy Yang, Ananya Kumar, Andre Saraiva, Andrea Vallone, Andrei Gheorghe, Andres Garcia Garcia, Andrew Braunstein, Andrew Liu, Andrew Schmidt, Andrey Mereskin, Andrey Mishchenko, Andy Applebaum, Andy Rogerson, Ann Rajan, Annie Wei, Anoop Kotha, Anubha Srivastava, Anushree Agrawal, Arun Vijayvergiya, Ashley Tyra, Ashvin Nair, Avi Nayak, Ben Eggers, Bessie Ji, Beth Hoover, Bill Chen, Blair Chen, Boaz Barak, Borys Minaiev, Botao Hao, Bowen Baker, Brad Lightcap, Brandon McKinzie, Brandon Wang, Brendan Quinn, Brian Fioca, Brian Hsu, Brian Yang, Brian Yu, Brian Zhang, Brittany Brenner, Callie Riggins Zetino, Cameron Raymond, Camillo Lugaresi, Carolina Paz, Cary Hudson, Cedric Whitney, Chak Li, Charles Chen, Charlotte Cole, Chelsea Voss, Chen Ding, Chen Shen, Chengdu Huang, Chris Colby, Chris Hallacy, Chris Koch, Chris Lu, Christina Kaplan, Christina Kim, CJ Minott-Henriques, Cliff Frey, Cody Yu, Coley Czarnecki, Colin Reid, Colin Wei, Cory Decareaux, Cristina Scheau, Cyril Zhang, Cyrus Forbes, Da Tang, Dakota Goldberg, Dan Roberts, Dana

Palmie, Daniel Kappler, Daniel Levine, Daniel Wright, Dave Leo, David Lin, David Robinson, Declan Grabb, Derek Chen, Derek Lim, Derek Salama, Dibya Bhattacharjee, Dimitris Tsipras, Dinghua Li, Dingli Yu, DJ Strouse, Drew Williams, Dylan Hunn, Ed Bayes, Edwin Arbus, Ekin Akyurek, Elaine Ya Le, Elana Widmann, Eli Yani, Elizabeth Proehl, Enis Sert, Enoch Cheung, Eri Schwartz, Eric Han, Eric Jiang, Eric Mitchell, Eric Sigler, Eric Wallace, Erik Ritter, Erin Kavanaugh, Evan Mays, Evgenii Nikishin, Fangyuan Li, Felipe Petroski Such, Filipe de Avila Belbute Peres, Filippo Raso, Florent Bekerman, Foivos Tsimpourlas, Fotis Chantzis, Francis Song, Francis Zhang, Gaby Raila, Garrett McGrath, Gary Briggs, Gary Yang, Giambattista Parascandolo, Gildas Chabot, Grace Kim, Grace Zhao, Gregory Valiant, Guillaume Leclerc, Hadi Salman, Hanson Wang, Hao Sheng, Haoming Jiang, Haoyu Wang, Haozhun Jin, Harshit Sikchi, Heather Schmidt, Henry Aspegren, Honglin Chen, Huida Qiu, Hunter Lightman, Ian Covert, Ian Kivlichan, Ian Silber, Ian Sohl, Ibrahim Hammoud, Ignasi Clavera, Ikai Lan, Ilge Akkaya, Ilya Kostrikov, Irina Kofman, Isak Etinger, Ishaan Singal, Jackie Hehir, Jacob Huh, Jacqueline Pan, Jake Wilczynski, Jakub Pachocki, James Lee, James Quinn, Jamie Kiros, Janvi Kalra, Jasmyn Samaroo, Jason Wang, Jason Wolfe, Jay Chen, Jay Wang, Jean Harb, Jeffrey Han, Jeffrey Wang, Jennifer Zhao, Jeremy Chen, Jerene Yang, Jerry Tworek, Jesse Chand, Jessica Landon, Jessica Liang, Ji Lin, Jiancheng Liu, Jianfeng Wang, Jie Tang, Jihan Yin, Joanne Jang, Joel Morris, Joey Flynn, Johannes Ferstad, Johannes Heidecke, John Fishbein, John Hallman, Jonah Grant, Jonathan Chien, Jonathan Gordon, Jongsoo Park, Jordan Liss, Jos Kraaijeveld, Joseph Guay, Joseph Mo, Josh Lawson, Josh McGrath, Joshua Vendrow, Joy Jiao, Julian Lee, Julie Steele, Julie Wang, Junhua Mao, Kai Chen, Kai Hayashi, Kai Xiao, Kamyar Salahi, Kan Wu, Karan Sekhri, Karan Sharma, Karan Singhal, Karen Li, Kenny Nguyen, Keren Gu-Lemberg, Kevin King, Kevin Liu, Kevin Stone, Kevin Yu, Kristen Ying, Kristian Georgiev, Kristie Lim, Kushal Tirumala, Kyle Miller, Lama Ahmad, Larry Lv, Laura Clare, Laurance Fauconnet, Lauren Itow, Lauren Yang, Laurentia Romaniuk, Leah Anise, Lee Byron, Leher Pathak, Leon Maksin, Leyan Lo, Leyton Ho, Li Jing, Liang Wu, Liang Xiong, Lien Mamitsuka, Lin Yang, Lindsay McCallum, Lindsey Held, Liz Bourgeois, Logan Engstrom, Lorenz Kuhn, Louis Feuvrier, Lu Zhang, Lucas Switzer, Lukas Kondraciuk, Lukasz Kaiser, Manas Joglekar, Mandeep Singh, Mandip Shah, Manuka Stratta, Marcus Williams, Mark Chen, Mark Sun, Marselus Cayton, Martin Li, Marvin Zhang, Marwan Aljubeh, Matt Nichols, Matthew Haines, Max Schwarzer, Mayank Gupta, Meghan Shah, Melody Y. Guan, Melody Huang, Meng Dong, Mengqing Wang, Mia Glaese, Micah Carroll, Michael Lampe, Michael Malek, Michael Sharman, Michael Zhang, Michele Wang, Michelle Pokrass, Mihai Florian, Mikhail Pavlov, Miles Wang, Ming Chen, Mingxuan Wang, Minnia Feng, Mo Bavarian, Molly Lin, Moose Abdool, Mostafa Rohaninejad, Nacho Soto, Natalie Staudacher, Natan LaFontaine, Nathan Marwell, Nelson Liu, Nick Preston, Nick Turley, Nicklas Ansman, Nicole Blades, Nikil Pancha, Nikita Mikhaylin, Niko Felix, Nikunj Handa, Nishant Rai, Nitish Keskar, Noam Brown, Ofir Nachum, Oleg Boiko, Oleg Murk, Olivia Watkins, Oona Gleeson, Pamela Mishkin, Patryk Lesiewicz, Paul Baltescu, Pavel Belov, Peter Zhokhov, Philip Pronin, Phillip Guo, Phoebe Thacker, Qi Liu, Qiming Yuan, Qinghua Liu, Rachel Dias, Rachel Puckett, Rahul Arora, Ravi Teja Mullapudi, Raz Gaon, Reah Miyara, Rennie Song, Rishabh Aggarwal, RJ Marsan, Robel Yemiru, Robert Xiong, Rohan Kshirsagar, Rohan Nuttall, Roman Tsiupa, Ronen Eldan, Rose Wang, Roshan James, Roy Ziv, Rui Shu, Ruslan Nigmatullin, Saachi Jain, Saam Talaie, Sam Altman, Sam Arnesen, Sam Toizer, Sam Toyer, Samuel Miserendino, Sandhini Agarwal, Sarah Yoo, Savannah Heon, Scott Ethersmith, Sean Grove, Sean Taylor, Sebastien Bubeck, Sever Banesiu, Shaokyi Amdo, Shengjia Zhao, Sherwin Wu, Shibani Santurkar, Shiyu Zhao, Shraman Ray Chaudhuri, Shreyas Krishnaswamy, Shuaiqi, Xia, Shuyang Cheng, Shyamal Anadkat, Simón Posada Fishman, Simon Tobin, Siyuan Fu, Somay Jain, Song Mei, Sonya Egoian, Spencer Kim, Spug Golden, SQ Mah, Steph Lin, Stephen Imm, Steve Sharpe, Steve Yadlowsky, Sulman Choudhry, Sungwon Eum, Suvansh Sanjeev, Tabarak Khan, Tal Stramer, Tao Wang, Tao Xin, Tarun Gogineni, Taya Christianson, Ted Sanders, Tejal Patwardhan, Thomas Degry, Thomas Shadwell, Tianfu Fu, Tianshi Gao, Timur Garipov, Tina Sriskandarajah, Toki Sherbakov, Tomek Korbak, Tomer Kaftan, Tomo Hiratsuka, Tongzhou Wang, Tony Song, Tony Zhao, Troy Peterson, Val Kharitonov, Victoria Chernova, Vineet Kosaraju, Vishal Kuo, Vitchyr Pong, Vivek Verma, Vlad Petrov, Wanning Jiang, Weixing Zhang, Wenda Zhou, Wenlei Xie, Wenting Zhan, Wes McCabe, Will DePue, Will Ellsworth, Wulfie Bain, Wyatt Thompson, Xiangning Chen, Xiangyu Qi, Xin Xiang, Xinwei Shi, Yann Dubois, Yaodong Yu, Yara Khakbaz, Yifan Wu, Yilei Qian, Yin Tat Lee, Yinbo Chen, Yizhen Zhang, Yizhong Xiong, Yonglong Tian, Young Cha, Yu Bai, Yu Yang, Yuan Yuan, Yuanzhi Li, Yufeng Zhang, Yuguang Yang, Yujia Jin, Yun Jiang, Yunyun Wang, Yushi Wang, Yutian Liu, Zach Stubenvoll, Zehao Dou, Zheng Wu, and Zhigang Wang. Openai gpt-5

system card, 2026. URL https://arxiv.org/abs/2601.03267.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024. URL https://arxiv.org/abs/2408.03314.

Richard Sutton. The bitter lesson, 2019. URL http://www.incompleteideas.net/IncIdeas/BitterLesson.html.

Edan Toledo, Karen Hambardzumyan, Martin Josifoski, Rishi Hazra, Nicolas 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, 2025. URL https://arxiv.org/abs/2507.02554.

Yike Wang, Huaisheng Zhu, Zhengyu Hu, Yige Yuan, Zhengyu Chen, Shakti Senthil, Hannaneh Hajishirzi, Yulia Tsvetkov, Pradeep Dasigi, and Teng Xiao. Rethinking the evaluation of harness evolution for agents, 2026a. URL https://arxiv.org/abs/2607.12227.

Yuru Wang, Lejun Cheng, Yuxin Zuo, Sihang Zeng, Bingxiang He, Che Jiang, Junlin Yang, Yuchong Wang, Kaikai Zhao, Weifeng Huang, Kai Tian, Zhenzhao Yuan, Jincheng Zhong, Weizhi Wang, Ning Ding, Bowen Zhou, and Kaiyan Zhang. Naturebench: Can coding agents match the published sota of nature-family papers?, 2026b. URL https://arxiv.org/abs/2606.24530.

Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. Natural evolution strategies. *Journal of Machine Learning Research*, 15(27):949–980, 2014. URL http://jmlr.org/papers/v15/wierstra14a.html.

Jiawei Xu, Arief Koesdwiady, Sisong Bei, Yan Han, Baixiang Huang, Dakuo Wang, Yutong Chen, Zheshen Wang, Peihao Wang, Pan Li, and Ying Ding. Rethinking the value of multi-agent workflow: A strong single agent baseline, 2026. URL https://arxiv.org/abs/2601.12307.

Junlin Yang, Che Jiang, Yu Fu, Tianwei Luo, Can Ren, Weizhi Wang, Kaikai Zhao, Hongyi Liu, Yuxin Zuo, Yuru Wang, Yuchen Fan, Kai Tian, Zhenzhao Yuan, Xiaojian Lin, Li Sheng, Rushi Qiang, Guoli Jia, Xingtai Lv, Ermo Hua, Dianqiao Lei, Youbang Sun, Ning Ding, Bowen Zhou, and Kaiyan Zhang. Frontis-ma1: Training an ai4ai model towards recursive self-improvement in machine learning engineering, 2026. URL https://arxiv.org/abs/2607.28568.

Xu Yang, Xiao Yang, Shikai Fang, Yifei Zhang, Jian Wang, Bowen Xian, Qizheng Li, Jingyuan Li, Minrui Xu, Yuante Li, Haoran Pan, Yuge Zhang, Weiqing Liu, Yelong Shen, Weizhu Chen, and Jiang Bian. R&d-agent: An llm-agent framework towards autonomous data science, 2025. URL https://arxiv.org/abs/2505.14738.

Zhuowen Yuan, Tao Liu, Yang Yang, Yang Wang, Feng Qi, Kaushik Rangadurai, Bo Li, and Shuang Yang. Archpilot: A proxy-guided multi-agent approach for machine learning engineering, 2025. URL https://arxiv.org/abs/2511.03985.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025.

Yifei Zhang, Xu Yang, Xiao Yang, Bowen Xian, Qizheng Li, Shikai Fang, Jingyuan Li, Jian Wang, Mingrui Xu, Weiqing Liu, and Jiang Bian. Reasoning as gradient: Scaling mle agents beyond tree search, 2026a. URL https://arxiv.org/abs/2603.01692.

Yunxiang Zhang, Kang Zhou, Zhichao Xu, Kiran Ramnath, Yun Zhou, Sangmin Woo, Haibo Ding, and Lin Lee Cheong. Learning to ideate for machine learning engineering agents, 2026b. URL https://arxiv.org/abs/2601.17596.

Mingming Zhao, Jiqian Dong, Kangping Xu, Zadid Hasan, Chengrui Fan, Shan Jiang, Shuai Mao, Yating Ling, Linyi Zou, Tailin Zhou, Yun Hin Chan, Wenkai Zhang, Zhanhong Zhou, Guowei Huang, Hongliang Li, Wenjing Cun, Zhitang Chen, Mingxuan Yuan, and Yanhui Geng. Scienceflow: A long-horizon agent for ml research, scientific discovery and beyond, 2026. URL https://arxiv.org/abs/2608.14354.

Yuhang Zhou, Lizhu Zhang, Yifan Wu, Jiayi Liu, Xiangjun Fan, Zhuokai Zhao, and Hong Yan. Synthetic sandbox for training machine learning engineering agents, 2026. URL https://arxiv.org/abs/2604.04872.

Xinyu Zhu, Yuzhu Cai, Zexi Liu, Bingyang Zheng, Cheng Wang, Rui Ye, Yuzhi Zhang, Linfeng Zhang, Weinan E, Siheng Chen, and Yanfeng Wang. Toward ultra-long-horizon agentic science: Cognitive accumulation for machine learning engineering, 2026. URL https://arxiv.org/abs/2601.10402.

Qiran Zou, Hou Hei Lam, Wenhao Zhao, Yiming Tang, Tingting Chen, Samson Yu, Tianyi Zhang, Chang Liu, Xiangyang Ji, and Dianbo Liu. Fml-bench: Benchmarking machine learning agents for scientific research, 2026. URL https://arxiv.org/abs/2510.10472.

A EXPERIMENTAL SETTING DETAILS

A.1 MLE-BENCH TASKS USED

Our primary dataset used in this paper is MLE-bench (Chan et al., 2025), which consists of 75 tasks. In this paper we focus on a subset of 30 tasks. We largely follow Hambardzumyan et al. (2026), which used the 30 tasks reported in the GPT-5 system card (Singh et al., 2026) for their experiments. We, however, substitute the task `bms-molecular-translation` in favour of `alaska2-image-steganalysis`, due to its more permissive license. Our final split consists of 5 Lite, 21 Medium and 4 High tasks. All tasks used in the paper are shown in Table 3, along with the split they belong to and the shorthand name by which they may be referenced in this paper.

Results presented in Section 4 are reported on the original MLE-bench without any intervention, and in the 29 task split (without either `bms` or `alaska2`). This split is referred to in this work as the `default29` task split.

We note that a subset of our tasks has been previously reported containing issues in the MLE-bench preparation scripts. Such issues may include missing data needed to correctly train ML pipelines, data present that leaks the ground truth predictions, etc. We have identified six competitions that could either lead to impossible tasks or are susceptible to cheating. In order to maintain a fair comparison set between our baselines and other tested public harnesses, we fixed such issues and report their performance for all results in this paper when comparing against existing baselines. Such tasks are marked with ‡ in Table 3 and the split is labelled as the `fixed30` set in this paper. Appendix A.2 lists the exact code diffs applied to the MLE-bench preparation scripts, together with the underlying bugs they address.

The selection criterion for our 14-task and 10-task challenging splits presented in Section 4.4 and Appendix C.7 was based on removing the floor – tasks solved by Gemma4 31B – and removing ceiling – tasks not solved by Kimi K3. These selections are shown in Table 3 as tasks with * marks and † marks, for `fixed14` and `fixed10`, respectively. As their name suggests, this subset is running with the fixed task version, when applicable.

A.2 FIXES APPLIED TO MLE-BENCH TASKS

Below we list the fixes for `champs`, `hubmap`, `smartphone`, and `multi-modal`: the bug in each, and the change we made to its MLE-bench `prepare.py`. All four issues are documented by the MLE-bench maintainers on GitHub, in the “Known Issues” section of the official repository’s README (`https://github.com/openai/mle-bench#known-issues`): `champs` in pull request #70 (`https://github.com/openai/mle-bench/pull/70`), `multi-modal` in issue #77 (`https://github.com/openai/mle-bench/issues/77`), `smartphone` in issue #93 (`https://github.com/openai/mle-bench/issues/93`), and `hubmap` directly in the README. The maintainers deferred fixing them to avoid invalidating existing leaderboard submissions, so the upstream preparation scripts still contain these bugs; the diffs below are our own fixes. The diffs are shortened for space (`...` marks omitted context, comments are trimmed, and long lines are re-wrapped). In all four cases the train/test splits and private answers are identical to the original MLE-bench preparation; only the files given to the agent change.

`champs-scalar-coupling`: missing test geometry. The original script kept `structures.csv` and the per-molecule `.xyz` files only for training molecules. On Kaggle, the 3D structures are provided for both train and test (only the five quantum-chemistry “additional data” files are train-only). We keep structures for every train and test molecule.

```
--- a/mlebench/competitions/champs-scalar-coupling/prepare.py
+++ b/mlebench/competitions/champs-scalar-coupling/prepare.py
+    all_molecule_names = (set(new_train["molecule_name"])
+                          | set(new_test["molecule_name"]))
     structures = read_csv(raw / "structures.csv")
-    structures = structures[
-        structures["molecule_name"].isin(new_train["molecule_name"])]
+    structures = structures[
+        structures["molecule_name"].isin(all_molecule_names)]
 ...
-    data_csvs = {
```

Table 3: **Full MLE-bench task suite used in the paper.** Competitions marked with ‖ are not part of the `default29` split. Tasks with * indicate the `fixed14` task subset, and † indicates `fixed10`. Tasks with ‡ were identified to have breaking issues, and results aggregated on them in `fixed` subsets come from our fixed version of such tasks.

	Full competition ID	Split
alaska2*†‖	alaska2-image-steganalysis	Medium
aptos2019	aptos2019-blindness-detection	Lite
billion-word*	billion-word-imputation	Medium
cassava*†	cassava-leaf-disease-classification	Medium
champs*†‡	champs-scalar-coupling	Medium
freesound*†	freesound-audio-tagging-2019	Medium
h&m	h-and-m-personalized-fashion-recommendations	Medium
hms‡	hms-harmful-brain-activity-classification	High
hotel	hotel-id-2021-fgvc8	Medium
hubmap*†‡	hubmap-kidney-segmentation	Medium
imet*†	imet-2020-fgvc7	Medium
jigsaw*‡	jigsaw-unintended-bias-in-toxicity-classification	Medium
kuzushiji*†	kuzushiji-recognition	Medium
mlsp	mlsp-2013-birds	Lite
multi-modal*†‡	multi-modal-gesture-recognition	Medium
new*	new-york-city-taxi-fare-prediction	Lite
nfl*†	nfl-player-contact-detection	High
nomad2018	nomad2018-predict-transparent-conductors	Lite
osic*	osic-pulmonary-fibrosis-progression	Medium
petfinder*†	petfinder-pawpularity-score	Medium
plant	plant-pathology-2021-fgvc8	Medium
smartphone‡	smartphone-decimeter-2022	High
spooky	spooky-author-identification	Lite
stanford	stanford-covid-vaccine	High
tensorflow2	tensorflow2-question-answering	Medium
tweet	tweet-sentiment-extraction	Medium
us	us-patent-phrase-to-phrase-matching	Medium
uw	uw-madison-gi-tract-image-segmentation	Medium
ventilator	ventilator-pressure-prediction	Medium
whale	whale-categorization-playground	Medium

```
-        "structures": structures,
+    assert set(structures["molecule_name"]) == all_molecule_names
+    train_only_csvs = {
         "dipole_moments": dipole_moments,
         ...
 ...
     for molecule_name in tqdm(
-        new_train["molecule_name"].unique(), desc="Copying ..."
+        sorted(all_molecule_names), desc="Copying ..."
     ):
```

`hubmap-kidney-segmentation`: leaked test annotations. The original script copied `{image_id}.json` into `public/test/`. This file is the unencoded glomerulus polygon annotation, i.e. the same information as the RLE-encoded target. Kaggle's real test set only ships the `-anatomical-structure.json` sidecar. We drop the annotation from the test copy and assert that only the sidecar remains.

```
--- a/mlebench/competitions/hubmap-kidney-segmentation/prepare.py
+++ b/mlebench/competitions/hubmap-kidney-segmentation/prepare.py
     for image_id in tqdm(new_test["id"], desc="Copying test images"):
         shutil.copy(raw / "train" / f"{image_id}.tiff",
                     public / "test" / f"{image_id}.tiff")
-        shutil.copy(raw / "train" / f"{image_id}.json",
-                    public / "test" / f"{image_id}.json")
```

```
            shutil.copy(
                raw / "train" / f"{image_id}-anatomical-structure.json",
                public / "test" / f"{image_id}-anatomical-structure.json",
 ...
+    assert len(list((public / "test").glob("*.json"))) == len(new_test)
```

`smartphone-decimeter-2022`: leaked reference trajectory. The original script copied each test drive's full raw directory into `public/test/` and removed only `ground_truth.csv`. The latitude/longitude in `ground_truth.csv` is linearly interpolated from the reference NovAtel SPAN receiver's NMEA log, and that log stayed in the public test folder. We delete every SPAN-reference `.nmea` file and keep the per-phone onboard-GNSS `.nmea` logs, which Kaggle legitimately provides.

```
--- a/mlebench/competitions/smartphone-decimeter-2022/prepare.py
+++ b/mlebench/competitions/smartphone-decimeter-2022/prepare.py
     for fpath in (public / "test").rglob("ground_truth.csv"):
         fpath.unlink()  # don't include ground truth in public test data
+    span_nmea_files = [
+        fpath for fpath in (public / "test").rglob("*.nmea")
+        if "span" in fpath.name.lower()
+    ]
+    assert len(span_nmea_files) > 0
+    for fpath in span_nmea_files:
+        fpath.unlink()  # reference trajectory ground truth is derived from
 ...
+    assert not any("span" in fpath.name.lower()
+                   for fpath in (public / "test").rglob("*.nmea"))
```

`multi-modal-gesture-recognition`: labelled test samples. The original script built the test set by copying the raw `training4.tar.gz` byte for byte. `training4` is fully labelled training data, so every sample's `*_data.mat` still contained the ground-truth gesture sequence in `Video.Labels`. We rebuild the archive one sample at a time, empty `Video.Labels`, and check that every repacked sample has empty labels.

```
--- a/mlebench/competitions/multi-modal-gesture-recognition/prepare.py
+++ b/mlebench/competitions/multi-modal-gesture-recognition/prepare.py
+def _clear_video_labels(mat_bytes: bytes) -> bytes:
+    mat = scipy.io.loadmat(io.BytesIO(mat_bytes))
+    video = mat["Video"]
+    labels = video["Labels"][0, 0]
+    ...
+    video["Labels"][0, 0] = np.empty((1, 0), dtype=labels.dtype)
+    buf = io.BytesIO()
+    scipy.io.savemat(buf, mat)
+    return buf.getvalue()
+
+def _relabel_test_archive(training4_dir: Path, dst_tar_gz: Path):
+    # for each Sample*.zip: rewrite the zip with its *_data.mat
+    # passed through _clear_video_labels, other members unchanged,
+    # and add it to dst_tar_gz
+    ...
 ...
-    shutil.copyfile(src=raw / "training4.tar.gz",
-                    dst=public / "test.tar.gz")
+    _relabel_test_archive(raw / "training4", public / "test.tar.gz")
+    # reopen test.tar.gz; assert every sample's Video.Labels is empty
```

A.3 NATUREBENCH TASKS USED

NatureBench (Wang et al., 2026b) asks an agent to try to match or exceed the *published* state-of-the-art result of a recent Nature-family paper on that paper's own held-out evaluation protocol. Unlike `fixed30`'s Kaggle competitions, there is no external leaderboard to compute a percentile against: each task ships a dedicated evaluation service that the agent may query online, unlimited times, via an `/evaluate` endpoint that returns the candidate's score. We normalize each score against the task's own naive baseline (0.0) and published SOTA result (1.0), so a value above 1.0 means the published SOTA is beaten. We report the beat-SOTA rate — the fraction of tasks on which the best score achieved anywhere in the run (oracle) surpasses the published SOTA — macro-averaged over a licensed task cohort at an 8h time budget, for both a GLM 5.2 and a Kimi K3 backbone.

Table 4 lists the 40 NatureBench tasks used for the results in Section 5.

Table 4: **NatureBench task cohort used in the paper.** Task IDs are the paper's Nature-family DOI suffix.

s41467-025-65557-7	s41592-025-02662-x	s42256-023-00630-8	s42256-025-01010-0
s41551-024-01257-9	s41592-025-02826-9	s42256-023-00636-2	s42256-025-01026-6
s41551-024-01312-5	s41592-025-02886-x	s42256-023-00654-0	s43588-024-00689-2
s41587-024-02414-w	s42256-022-00447-x	s42256-023-00712-7	s43588-024-00698-1
s41592-023-01940-w	s42256-022-00459-7	s42256-024-00790-1	s43588-024-00716-2
s41592-023-02035-2	s42256-022-00518-z	s42256-024-00795-w	s43588-024-00732-2
s41592-023-02124-2	s42256-022-00526-z	s42256-024-00815-9	s43588-024-00757-7
s41592-024-02257-y	s42256-022-00541-0	s42256-024-00892-w	s43588-025-00842-5
s41592-024-02316-4	s42256-023-00627-3	s42256-024-00956-x	s43588-025-00917-3
s41592-024-02372-w	s42256-023-00628-2	s42256-025-01002-0	s43588-025-00920-8

A.4 LLM BACKBONES USED

Unless otherwise stated, we report results using GLM 5.2 backbone LLM. All backbones are served locally with vLLM (Kwon et al., 2023) on 8×B200 or 8×A100 80GB nodes for smaller models. Unless otherwise specified, all models run with reasoning enabled at their maximum available configuration, and at their canonical maximum sequence length. Input and output costs, when reported, are the weighted effective per-token prices listed for each backbone on `https://openrouter.ai/` — a marketplace that aggregates multiple inference providers per model and reports a usage-weighted effective price across them, rather than any single provider's list price — snapshotted around September 2026. Cache-read cost is modelled as $0.1\times$ the input cost: none of the harnesses compared here report real cache-hit token counts directly (see the per-method telemetry notes in Section 5.1 and Appendix D.4), so this is a fixed discount applied uniformly rather than a per-backbone reported cache price. The following are all the LLMs reported in this paper, along with their HuggingFace identifier; backbones we compute a dollar cost for elsewhere in the paper are annotated with their September-2026 OpenRouter price:

- GPT OSS 120B: `openai/gpt-oss-120b` (OpenAI, 2025).
- Agents A1: `InternScience/Agents-A1` (Bai et al., 2026).
- Gemma 4 31B: `google/gemma-4-31B-it` (Gemma Team, 2026) ($0.1641 / M input, $0.5146 / M output).
- DeepSeek V4 Flash (Preview): `deepseek-ai/DeepSeek-V4-Flash-0731` (DeepSeek-AI, 2026).
- DeepSeek V4 Flash: `deepseek-ai/DeepSeek-V4-Flash` (DeepSeek-AI, 2026).
- DeepSeek V4 Pro (Preview): `nvidia/DeepSeek-V4-Pro-NVFP4` (DeepSeek-AI, 2026).
- DeepSeek V4 Pro: `nvidia/DeepSeek-V4-Pro-0813-NVFP4` (DeepSeek-AI, 2026).
- GLM 5.2: `nvidia/GLM-5.2-NVFP4` (GLM-5-Team et al., 2026) ($0.3477 / M input, $3.123 / M output).
- Kimi K3: `moonshotai/Kimi-K3` (Kimi Team et al., 2026) ($0.5714 / M input, $13.99 / M output).

A.5 STATISTICAL PROTOCOL

Throughout the paper, our main performance reported is based on the macro-task average. For an experiment with T different tasks, each task t containing N_t seeds, for metric metric, we report:

$$\frac{1}{T}\sum_{t=1}^{T}\frac{1}{N_t}\sum_{r=1}^{N_t}\mathrm{metric}(\mathrm{run}_{t,r}).$$

Our confidence intervals are estimated via bootstrap on the fixed set of T tasks, sampling for each task from the N_t available seeds before computing the macro average, and taking the quantiles of

the simulated distribution. For each plot and table presented, the comparison is always based on the same task set. In the tables presented in this paper, we mark as bold numbers the best results obtained across arms, and when present, with underlined and italics the second and third best, respectively.

For our A vs B comparisons, we compute the paired-by-task difference to isolate positive or negative effects on the ablation:

$$\frac{1}{T}\sum_{t=1}^{T}\left(\frac{1}{N(A)_t}\sum_{r=1}^{N(A)_t}\text{metric}(\text{run}(A)_{t,r}) - \frac{1}{N(B)_t}\sum_{r=1}^{N(B)_t}\text{metric}(\text{run}(B)_{t,r})\right).$$

To determine statistical significance, we compute the 95% CIs of the paired-by-task difference (again, on the fixed set of T tasks, bootstrapping from $N(A)_t$ and $N(B)_t$). If 0 does not lie in the CI — which we mark with * in our paired-by-task tables throughout the paper — we can establish a statistically significant difference in the metric in the ablation. When 0 lies in the CI, the arms become statistically indistinguishable at our power level based on the amount of seeds we ran in our experiments. With this method, between-task difficulty cancels, which results in narrower intervals than regular marginal CIs.

B Implementation details

In this section, we provide more details about our core iterations ablated, along with their optional tool and infrastructure addons below.

B.1 Iterations

All three iterations are compiled from a single prompt-template family. All iterations hare a base prompt with the agent persona, the scientific-approach questions, the resource and time budget line and a few general recommendations. A specialized task description and instruction block is then chosen for each iteration and aligns with the goal and environment available for each method.

In the task description, the Oneshot and Malena iterations are handed only the data paths (raw data directory, description file and sample submission file) since they can explore the data themselves through their tools. On the other hand, Chat receives the task description verbatim and an inline overview of the data computed by the harness: a directory tree of the data folder, per-file previews (per-column statistics for CSVs — dtype, cardinality/value counts, mean, range and NaN counts — auto-generated schemas for JSON files, and the first lines of text files, subject to a global character budget), and the first rows of the sample submission. All iterations execute inside a git worktree checked out at their parent node's commit, and each produces a node whose recorded validation score tuple is what the search rules of Section 4.2 (Appendix B.3) consume.

1. **Chat**. A single chat-completion call with no tool access. The model receives the base prompt with the inline data overview and is instructed to reply with a complete, self-contained Python script that implements the full pipeline (data → model → training → inference) and writes `submission.csv`. The response text is saved verbatim to `main.py` after stripping any enclosing code fences, and executed once with the remaining time budget.

   If the run succeeds and the submission validates against the expected format, the agent is queried again with the output of the command to extract the validation score, and the iteration ends. Otherwise the harness re-prompts the model — appending to the same conversation — with a failure report containing the failure reason (non-zero exit, missing or invalid submission), the truncated stdout/stderr, a hardware-utilisation summary, and a footer that disambiguates system kills (GPU-memory watchdog, RAM watchdog) from ordinary crashes. The model must answer with the full corrected script. This loop repeats until a valid submission is produced.

2. **Oneshot**. One coding-agent session, one conversation turn, under the default tool environment of Appendix B.2. The prompt instructs the agent to treat the worktree as a scratch directory to confirm its design with cheap verification runs (fast data-inspection scripts, then minimal *test runs*), and to converge to exactly one *production run*. After one production

run succeeds, no further heavy training is permitted, and failures may only be diagnosed with cheap runs, never with a second production attempt.

Four deliverables are required: `main.py` (the pipeline), `submission.csv`, `design.md` (a self-contained write-up of the design, results and rationale, meant to serve as the node's documentation for later iterations), and `score.txt` (a single floating-point validation score on the task's metric and direction). These deliverables are expressed as *guarantees*: when the agent ends its turn, the harness statically verifies the file set and re-prompts the agent with the list of missing or malformed guarantees (e.g. an unparsable score file) until all hold.

When the parent node already carries a scripted solution (a previous `main.py` and `design.md`), the prompt switches to an *improve* variant: the agent is instructed to study the documented design, to decide between refining and pivoting using the recorded execution cost and results, and to overwrite the same files (any other scripts are discarded). This is the mechanism by which the search strategies of Section 4.2 share information between nodes.

The implementation also supports appending optional instructions to the prompt as described in Appendix B.2. This addition is only added when the corresponding intervention is enabled.

3. **Malena**. The same coding-agent environment as Oneshot, but the session is allowed to span the iteration's entire time budget in place of a single production run, and the one-production-run and documentation requirements are lifted. The instructions grant the agent freedom to design any number of pipelines and produce any number of submissions, dividing its budget as it sees fit. The harness drives the session as a turn loop: whenever the agent ends a turn before the budget expires, it is re-prompted with a short follow-up message that reports the remaining time.

   Submissions are generally tracked using the submission plugin. The agent calls `submissions_register` itself (optionally consulting `submissions_list`), reporting each submission together with its validation score and free-form metadata, with the prompt making explicit that only registered submissions count and that each registration is atomic and immediate. When the submission plugin is disabled, the agent simply saves submission files under a `submissions/` directory and the harness salvages and registers every file found there at wrap-up (without a validation score). Unless explicitly stated, all experiments reported use the submission plugin with validation score signals.

B.2 Infrastructure

The discussion in this section only applies to coding agent iterations (Oneshot and Malena).

Our core coding agent loop harnesses over OpenCode. It preserves all default tools, except we explicitly disabled `doom_loop`, `plan_exit`, and `question`. Moreover, we overwrite the native `bash` tool with our own implementation (more information in the following overview). The most important and used tools for the coding agent are:

- `read`: Reads the contents of a file.
- `write`: Writes the contents of a file.
- `edit`: Edits the contents of a file.
- `bash`: Executes a command. Note that we disabled the native `bash` implementation, and used our own version. Both versions remain highly similar in usage, description and output format, except for the fact that we manually control the maximum command execution timeout (set to 24h in all experiments unless otherwise stated), general resource statistics are shown along with the output of the command upon completion (command duration and average and max CPU, GPU, RAM and VRAM usage by the command), along with a background watchdog service that automatically kills running commands shortly before the system runs out of memory, to avoid hard pod ejection by our compute provider. We only allow our agent to execute commands from a short whitelist, including `pwd`, `ls*`, `python*`, `nvidia-smi` and more.

- `task`: Allows the coding agent to launch multiple subagents with different prompts. The tool blocks until all subagents finish their work.
- `todowrite`: Allows the agent to maintain a task list of goals during its session.

On top of this, we additionally ablated the following tools. The default Malena configuration has `submission`, `system`, and `jobs` tools enabled. The other coding agent iterations run with them disabled, unless otherwise specified for any particular ablation.

- `submission_register` tool: Adds a submission file to the registry. No test or leaderboard scores are communicated to the agent. Additionally, the agent may use `submission_list` to list the previous submissions it has produced during the session. Whenever this tool is enabled, we also add a short text in the session's prompt to guide the agent regarding its usage.

S `system` tool: Returns the current hardware resources, utilization, and time remaining.

J `jobs` tools: Allow running bash commands in the background, waiting for them to complete and cancelling ongoing jobs. Whenever this tool is enabled, we also set a maximum time limit of 10m to all `bash` calls to encourage its usage, and add a short usage guide in the session's prompt about how to use it. This suite is composed of the following tools. `job_submit`: non-blocking, executes a command in the background and returns its general information (job ID, stdout path, etc.), `job_wait`: waits for a job to finish or, if specified, until a certain amount of time passes, `job_queue`: lists the running jobs, and `job_info`: a more detailed view of a particular job, including hardware usage, stdout location, and more.

s `skill`: Loads a skill matching the input's identifier. The agent is given the name and a one-line description of each available skill up front, and only loads the full body of a skill (via this tool) when it decides to consult it. We equip the agent with eight skills encoding common ML engineering strategies for iterating on a competition: *submission-blueprint* (build and verify the end-to-end submission pipeline first, using a trivial predictor, before training a real model); *quick-strong-baseline* (get the highest score for the least training and implementation effort, leaning on pretrained models and off-the-shelf libraries); *cost-saving-techniques* (a toolbox of levers — frozen backbones, low resolution, few epochs, data subsampling — for getting a faithful signal cheaply, meant to be folded into the other skills); *simplification-probe* (shrink the task, either by holding one axis at full difficulty while collapsing the rest, or by reformulating it as a smaller proxy task, to isolate where a score gap comes from); *ablation-probe* (design a controlled comparison that isolates one component's contribution, holding everything else fixed); *runtime-estimate* (measure a few scaled-down configurations and extrapolate wall-clock and memory cost before committing to a full-scale run); *stagnation-pivot* (detect when recent attempts within the same paradigm have stopped improving and force the next attempt to pivot the model, loss, input representation, or problem decomposition rather than tweak a hyperparameter); and *cross-iteration-synthesis* (step back and read across several past attempts together to compose ideas, extract best practices, standardize the validation protocol, or resolve contradictions).

H `huggingface_hub_repo_search`: The official HuggingFace MCP. Enables the agent to search for models in the HuggingFace Hub. Whenever this tool is enabled, we also add a short text in the session's prompt to guide the agent regarding its usage.

B `broadcast`: Enabled jointly with parallelism. Allows agents to broadcast messages to their peers, which are then queued and displayed after a turn end.

Additionally, we also ablate the following set of wider interventions, that each may include prompt changes or other environment differences for the agent:

D Delegation encouragement. This intervention enables background subagents via the native `task` tool, and encourages their use via a prompt addition.

N Time nudges: Scheduled at 25%, 50%, 75% and 90%, the system automatically queues a brief message to the session and informs the agent about elapsed time.

R Resource nudge: Whenever the CPU or GPU utilization falls under 50%, the system queues a brief message to the agent to encourage more efficient use of resources. There is a 5 minute waiting time between nudges to avoid overcrowding the session with messages.

V Strict validation guidelines: We inject to the prompt a set of strict validation guidelines the agent must adhere to, with the hopes to minimize data leakage.

I Minimal prompt: The initial prompt of the agent is simplified to contain only the crucial information, removing general guidelines to follow and examples.

In the entire paper, unless otherwise specified, Oneshot iteration is evaluated without any intervention, and Malena makes use of the `submission`, `system` and `jobs` tools only.

B.3 Search strategies

The Chain and Greedy strategies are straightforward and are implemented as described in Section 4.2. The UCB1 strategy follows a deterministic selection rule, applying the UCB1 index (Auer et al., 2002) to the pool of existing nodes. Unlike the tree-descent formulation of UCT (Kocsis & Szepesvári, 2006) adopted by prior MLE systems (Du et al., 2026; Chen et al., 2026b), which recursively selects among the children of the current node, we score every node in the tree and take a single global supremum.

In the following, assume the competition metric is higher-is-better (the lower-is-better case is analogous, replacing each normalised score by its complement). Let T be the current tree, $V(T)$ its nodes, and $\mathrm{s}(n)$ the validation score recorded on node n. Then, $\hat{s}(n) = \max_{n' \in \mathrm{subtree}(n)} \mathrm{s}(n')$, so that a node's value reflects what its descendants eventually achieved. This maximum backup replaces the running mean fitness used in canonical UCT implementations (Toledo et al., 2025; Du et al., 2026), which is appropriate here because candidates are deterministic artefacts to be kept rather than stochastic returns to be averaged. UCB1 then computes

$$R(n) = \frac{\sigma(\hat{s}(n))}{|V(T)| - 1} + c\sqrt{\frac{\ln\left(|V(T)| + 1\right)}{|C(n)| + 1}},$$

where $\sigma(x)$ is the 0-indexed rank of x among all available scores, $C(n)$ are the children of n, and $c > 0$ is the exploration-exploitation hyperparameter (fixed at $c = 0.75$ for all runs).

Ranking rather than using the raw metric is what makes a single c transferable across competitions: MLE-bench metrics are arbitrarily scaled and of mixed direction, and a rank transform is invariant to any order-preserving reparameterisation of the metric while remaining robust to outlying runs, in the manner of the rank-based fitness shaping used by evolution strategies (Hansen & Ostermeier, 2001; Wierstra et al., 2014). Prior MLE harnesses instead min-max normalise the metric (Chen et al., 2026b), substitute a bounded hand-designed reward (Du et al., 2026), or leave the metric raw and retune the exploration constant per setting (Toledo et al., 2025). Keeping the first term in $[0, 1]$ also lets us adopt the exploration constant that the UCT analysis prescribes for bounded rewards, $c \approx 1/\sqrt{2}$, without per-task tuning (Kocsis & Szepesvári, 2006).

Our strategy for Best-of-N methods consists of running independent single Oneshot iterations, and then composing a set of bootstrapped-trajectories $B_j = \{T_1^j, \ldots, T_N^j\}$ where each T_i^j is a composition of multiple independent Oneshot iterations of the requested time limit budget. Each trajectory is packed greedily so that the summed *effective* time of its iterations does not exceed the target budget, and an iteration that would overflow the trajectory is discarded rather than carried into the next one. The budget is therefore matched on the time the iterations actually consume, not on their nominal per-iteration time limit. We can then extract all metrics for each trajectory T_i^j by concatenating the submissions of all iterations in the trajectory. Finally, we sample multiple independent bootstrapped-trajectory sets $B_1, \ldots, B_M$ and report the average-of-averages across this set.

C Additional Chat, Oneshot and Malena experiments

C.1 Oneshot time limit effect

We ablate the effect of the per-iteration time limit in our default Oneshot experimental setting for DeepSeek V4 Pro (Preview) and GLM 5.2 models. Results are in Figure 7. We observe, as expected, a monotonic improvement of per-iteration performance as the time budget increases. Nonetheless, the actual performance of the aggregated Best-of-N composites quickly saturates at the three hour

mark. In our experimental setup, the six hour time limit default provides an reasonable choice to balance the performance and cost tradeoff. Note that, even when prompted to use the highest possible time budget of 12h, the agents still have a much lower average effective time usage, which is what allows us to aggregate Best-of-N at the 6h budget.

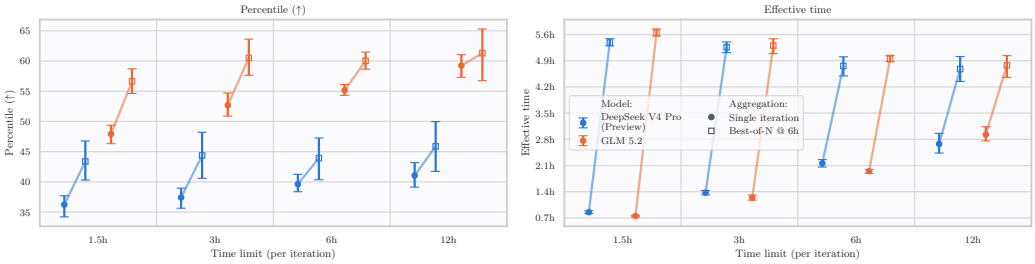

Figure 7: **Impact of per-iteration time limits.** Average percentile and effective-time obtained by Oneshot iterations under different per-iteration time budgets in the `default29` set. Circled points represent single-iteration performance, while boxes indicate Best-of-N performance at six hour budgets. Error bars represent 95% CIs across our fixed 29-task set. The effective time measures the time taken for the run to finish. Best-of-N aggregates may remain below the target of 6h in the case of failing iterations that would exceed the maximum budget allocated.

C.2 Pairwise Oneshot search strategy and Malena agent comparison

Table 5: **Pairwise paired-by-task comparison.** Results reported on the 24h budget, using the `default29` task split. In an A vs B comparison, positive percentile differences indicate better average performance of A. Percentile difference reported along with 95% CIs.

	Percentile (↑)	Medal rate (%, ↑)	
	Δ	Δ	won A/B/=
Oneshot search strategy			
Best-of-N vs UCB1	-2.71 [-6.60, +1.44]	-5.3 [-11.8, +1.1]	4/7/18
Best-of-N vs Greedy	-1.98 [-5.86, +1.89]	-5.3 [-11.2, +0.6]	3/7/19
Best-of-N vs Chain	-0.51 [-4.66, +3.27]	-4.2 [-11.2, +2.9]	4/7/18
UCB1 vs Greedy	+0.73 [-2.85, +4.35]	+0.0 [-5.7, +5.7]	5/5/19
UCB1 vs Chain	+2.20 [-1.71, +5.92]	+1.1 [-5.7, +8.0]	3/2/24
Greedy vs Chain	+1.47 [-2.31, +5.24]	+1.1 [-5.7, +8.0]	4/3/22
The Malena agent			
Malena vs Best-of-N	+5.09 [+1.34, +8.98]*	+11.9 [+5.6, +18.6]*	11/2/16
Malena vs UCB1	+2.38 [-1.06, +5.88]	+6.6 [+0.4, +12.9]*	7/3/19
Malena vs Greedy	+3.11 [-0.39, +6.71]	+6.6 [+0.5, +12.8]*	8/4/17
Malena vs Chain	+4.58 [+0.97, +8.35]*	+7.7 [+1.1, +14.8]*	9/3/17

Table 6: **Selection gap for different methods.** Results reported on the 24h budget, with the `default29` split, along with 95% CIs.

	Percentile (↑)		Selection gap (↓)
	Self-select	Oracle	
Malena	**69.30** [66.74, 71.78]	**71.18** [69.02, 73.46]	**1.876** [0.805, 3.205]
Best-of-N	64.21 [61.19, 67.07]	71.04 [69.49, 72.66]	6.832 [4.336, 9.634]
UCB1	66.92 [64.30, 69.36]	70.89 [68.45, 73.35]	3.973 [2.484, 5.367]
Greedy	66.19 [63.43, 68.71]	68.98 [67.32, 70.56]	2.788 [1.011, 5.054]
Chain	64.72 [61.97, 67.42]	66.87 [64.08, 69.57]	2.152 [1.409, 2.899]

In Section 4.2 and 4.3, we provided a head-to-head comparison in the paired-by-task regime between all search methods tested, and the Malena agent. In Table 5 we give a more detailed examination of such comparisons, and Table 6 gives a more detailed view on the selection gap between these methods.

C.3 HIDDEN VALIDATION GAP

In Section 4 we observed that the gap of the performance between self-select (based on validation score) and oracle-best (based on actual test scores) becomes a problem, particularly for strategies that share less information between their iterations. In this section, we analyse three different interventions that can help mitigate such issue. Our experiment setting is on the default Oneshot iteration configuration (six hour per-iteration time limit, GLM 5.2), and the baseline behaviour control is the Best-of-N aggregation at 12h budget setting. We compare:

- `V` Strict validation guidelines: We inject to the prompt a set of strict validation guidelines the agent must adhere to, with the hopes to minimize data leakage.
- `BI` Baseline iteration: At the beginning of the run, a Baseline iteration is spawned. The Baseline iteration aims to provide a simple, fast-to-run ML pipeline baseline to solve the task, while ensuring the validation split strategy follows a strict set of guidelines. Once this iteration ends, and until the 12h time limit, new Oneshot iterations use the Baseline's workspace as starting point, while still remaining independent from each other. The goal for this intervention is to provide a common validation strategy for the iterations, and minimize the inconsistencies that come from differently designed sets.
- `HV` Hidden validation: At the beginning of the iteration, the harness prepares a deterministic validation-train split from the training data, using MLE-bench's data preparation scripts. The agent does not have access to the target predictions of the hidden validation split. This strategy has the drawback of reducing the total amount of training data available to train the model in the first place. Note that, as described, this strategy cannot be deployed in practice, due to the hard requirement of the deterministic splitter which would require human intervention, and is therefore reported as a ceiling of what pre-splitting strategies can achieve.

Table 7: **Selection gap mitigation strategies.** Performance reported with 95% CIs. Top set: `default29` task split infrastructure ablation at 24h. Bottom set: `fixed10` split multi-agent orchestration ablation at 12h budget.

	Percentile (↑)		Selection gap (↓)
	Self-select	Oracle	
`BI`	61.24 [58.85, 63.82]	63.26 [61.28, 65.38]	**2.024** [1.125, 2.901]
`base`	62.56 [60.34, 64.63]	**67.99** [66.74, 69.26]	5.433 [3.681, 7.359]
`+V`	**63.74** [59.93, 67.04]	67.57 [66.23, 68.69]	3.833 [0.936, 7.263]
`HV`	61.55 [59.73, 64.36]	66.03 [63.38, 68.10]	4.479 [2.412, 5.944]

Table 7 shows the results of this experiment. We see that the base independent Best-of-N Oneshot configuration provides the best percentile ceiling (oracle), while the introduction of the strict validation guidelines in the prompt alone allows for a slightly better self-select performance, at the cost of also lowering the oracle ceiling. Finally, the introduction of the initial baseline iteration provides the smallest gap of all interventions ablated, at the cost of a significantly worse performance ceiling. A plausible mechanism for these differences is the comparability of the validation scores themselves, rather than the number of submissions available to choose from: a single session produces all of its candidates under one validation split, so their reported scores are mutually comparable, whereas independent Best-of-N iterations each design their own split and the harness is then asked to rank scores that were never on a common scale. The `BI` intervention is consistent with this reading, as it is the one that supplies a shared validation workspace. We report this as a hypothesis and not as a result: it predicts that the gap should grow with the number of distinct validation methodologies in the pool rather than with the number of submissions, which we do not test here. While these results

provide a quick overview about possible validation gap mitigation strategies, we acknowledge that at our current statistical power level, we cannot make strong conclusions about these results, and we encourage a more in-depth future work in this direction, including on the mechanism above.

C.4 MALENA BUDGET SCALING

We study how Malena scales under different time budgets. Our experimental setting consists of a DeepSeek V4 Pro (Preview) backbone, with a *bare* Malena iteration, meaning with the `submission`, `system`, and `jobs` tools disabled. This serves as the minimal harness over a coding agent. In our implementation, we use the `submission` tool to register the validation scores of each submission, and without it, we have no self-select mechanism. For this reason, in this section we report only the oracle ceiling score which, despite not being a production-ready configuration, still gives a realistic range of the temporal behaviour of the agent, especially since the Malena iteration shows the lowest validation gap in our tested iterations (see Appendix C.2).

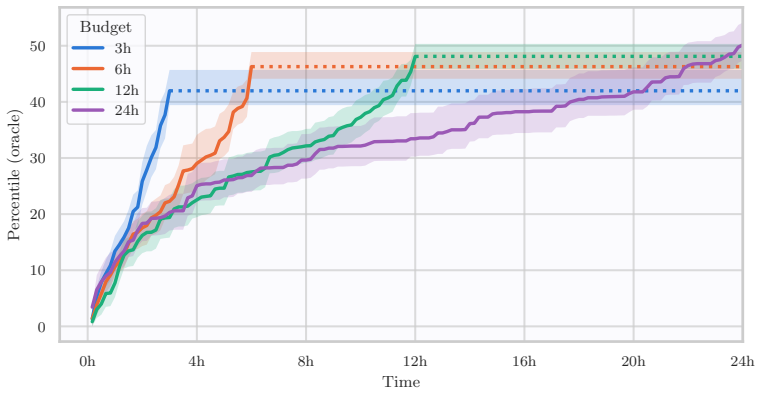

Figure 8: **Behaviour of Malena under different time budgets.** Average oracle percentile obtained by Malena on the `default29` split with a DeepSeek V4 Pro (Preview) backbone model. Trajectories are extracted from the full run by scoring the submissions available at each time. Dotted lines are shown after a timeout target is hit, and maintain the performance achieved.

Figure 8 shows the results of Malena under different time limits. We observe that the performance steadily increases as more budget is available. Interestingly, despite having an almost identical prompt where the only difference is the time limit mentioned itself, and the automatic time left reminders mentioned at the end of the turn,[4] the performance of the agent at identical timeframes varies significantly depending on the total time budget allocated. This suggests that the agent does adapt its exploration vs exploitation strategy depending on the time budget.

C.5 MALENA WITH DIFFERENT BACKBONES

We test the performance of the Malena iteration with different backbone models, and compare it with the Best-of-N Oneshot composite at the 24h time budget target. Our selection of backbones is obtained from the results observed in Section 4.1, and we selected two frontier LLMs (GLM 5.2 and Kimi K3), both final releases of the DeepSeek V4 family, and Gemma 4 31B — a smaller model that still shows clear benefits from the agentic environment. We plot our results in Figure 9, and detailed paired-by-task comparison in Table 8. While the frontier models show a clear benefit from the long single-session iteration, the percentile of the weaker model favours Best-of-N in comparison — although the medal rate remains consistently better in Malena iterations. This may suggest that larger models can better leverage the autonomy in the single-session, or that the smaller context window of 256k in the weaker model becomes a bottleneck at this time scale, as some important details may be lost in the summarization during the context window compaction that is triggered when the context limit is reached. We discuss this possibility further in Appendix C.7.

[4]Not to be confused with the Time nudge (`N`) intervention — which is also disabled in this experiment — which schedules automatic messages periodically regardless of the progress in the session turn.

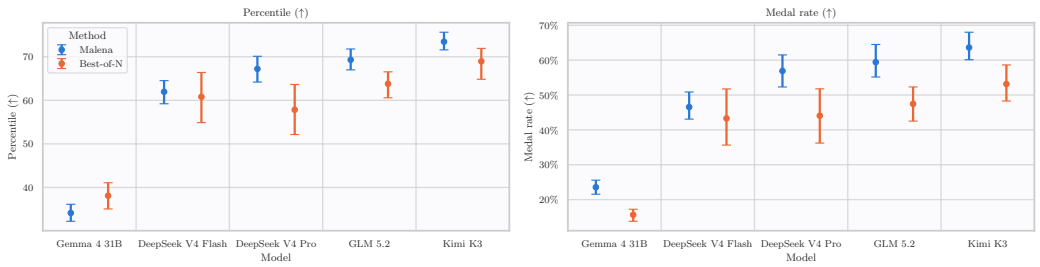

Figure 9: **Malena model ablation.** Average percentile and medal rate obtained by the Malena iteration and Best-of-N Oneshot iterations at the 24h time budget in the `default29` set. Error bars represent 95% CIs across our fixed 29-task set.

Table 8: **Pairwise paired-by-task comparison of different backbones.** Reported is the performance difference between Malena (A) vs Best-of-N (B); higher numbers indicate stronger average Malena performance. Results reported using the `default29` task split, with 95% CIs.

	Percentile (↑)	Medal rate (%, ↑)	
	Δ	Δ	won A/B/=
Gemma 4 31B	-3.97 [-7.98, -0.35]*	+7.9 [+4.3, +11.8]*	4/1/24
DeepSeek V4 Flash	+1.15 [-5.42, +7.77]	+3.3 [-6.3, +12.4]	9/6/14
DeepSeek V4 Pro	+9.39 [+3.36, +15.98]*	+12.8 [+2.9, +22.4]*	11/4/14
GLM 5.2	+5.52 [+1.93, +9.57]*	+12.0 [+5.7, +18.4]*	12/2/15
Kimi K3	+4.51 [+0.83, +8.96]*	+10.5 [+4.3, +18.1]*	8/4/17

C.6 Malena infrastructure ablation

In this section, we ablate further interventions of the Malena iteration itself. We first ablate the importance of all interventions of the Malena harness presented in Section 4.3, and we introduce a fourth intervention:

`N` Time nudges: Scheduled at 25%, 50%, 75% and 90%, the system automatically queues a brief message to the session and informs the agent about elapsed time. Note that for our default budget of 24h, the first time nudge is queued at 6h, which corresponds to the hard time limit of each Oneshot iteration.

Results are shown in Table 9, top row set. We observe that for such long 24h sessions, the performance of the agent is negatively impacted when removing the `system` tool, suggesting that the agent struggles to track time correctly without it. On the other hand, the addition of the time nudges results in a performance loss, perhaps due to a small distraction effect of unexpected messages. Finally, the removal of the `jobs` results in marginally better performance.

Furthermore, we incorporate on top of the time nudges (`N`) intervention a wider set of general-purpose infrastructure enhancements, with the objective of better understanding *where* could the Malena iteration be aided by the external harness. Namely, we test: a resource nudge that reminds the agent when hardware utilization is low (`R`), the HuggingFace MCP that allows the agent to look for pretrained models (`H`), a set of strict validation guidelines in the prompt (`V`) and a set of eight skills covering common ML engineering strategies for iterating on a competition, from establishing baselines to diagnosing stagnation (`s`). More information about these interventions in Appendix B.2. In Table 9, bottom row set, we show the results of this experiment. None of the interventions provide significant performance improvements at our current statistical power. We conclude that Malena as presented here serves as a capable enough baseline and further interventions either hurt performance, or do not produce significant improvements. This is in contrast to Oneshot iterations, which may benefit from more complex harnesses as explored in Appendix C.8.

Finally, Table 10 shows the pairwise comparison of all our ablations, including the ones presented in Section 4.4.

Table 9: **Malena infrastructure ablation.** Performance reported with 95% CIs in the `fixed14` split, at a matched 12h budget.

	Percentile (↑)		Medal rate (%, ↑)	
	Self-select	Oracle	Self-select	Oracle
Infrastructure ablation (24h, 29 tasks)				
`base`	69.30 [66.74, 71.78]	71.18 [69.02, 73.46]	59.4 [54.8, 63.9]	**62.5** [58.4, 66.3]
`+N`	63.85 [59.73, 67.77]	65.98 [62.05, 69.83]	48.9 [43.1, 54.0]	52.3 [46.6, 57.5]
`-J`	**70.47** [66.89, 74.25]	**72.84** [69.55, 76.10]	**62.1** [56.9, 67.2]	62.1 [56.9, 67.2]
`-S-J`	65.33 [62.05, 68.61]	68.36 [65.16, 71.45]	50.6 [44.8, 56.3]	56.9 [50.6, 62.6]
Additional ablation (12h, 14 tasks)				
`+N`	58.84 [55.58, 62.11]	63.18 [60.51, 65.83]	40.3 [34.1, 46.5]	45.2 [39.7, 50.7]
`+N+D`	61.28 [56.43, 65.46]	63.16 [58.41, 67.36]	**44.3** [36.2, 51.9]	47.1 [39.5, 54.8]
`+N+P3`	59.78 [55.23, 64.27]	**70.36** [68.95, 71.99]	43.2 [33.9, 52.1]	**57.1** [52.4, 61.9]
`+N+B+P3`	56.29 [52.41, 60.34]	63.35 [60.29, 66.57]	35.1 [29.2, 41.1]	41.9 [36.0, 47.9]
`+N+R`	51.08 [45.76, 55.70]	53.79 [48.45, 58.58]	30.4 [21.4, 38.7]	33.9 [25.0, 42.3]
`+N+D+H+V`	**63.10** [58.70, 66.74]	64.18 [59.75, 67.90]	**44.3** [37.1, 50.0]	52.4 [45.7, 58.6]
`+N+s`	58.05 [54.68, 61.02]	61.57 [57.80, 65.58]	38.1 [31.0, 45.2]	42.9 [35.7, 50.0]

Table 10: **Malena pairwise paired-by-task comparison.** In an A vs B comparison, positive percentile differences indicate better average performance of A. Differences reported along with 95% CIs. Top set: 24h, `default29` split. Middle set: 24h, `fixed14` split. Bottom set: 12h, `fixed14` split.

	Percentile (↑)	Medal rate (%, ↑)	
	Δ	Δ	won A/B/=
Infrastructure ablation (24h, 29 tasks)			
`base` vs `+N`	+5.46 [+0.70, +10.67]*	+10.6 [+3.7, +17.9]*	8/1/20
`base` vs `-J`	-1.17 [-5.64, +3.34]	-2.6 [-10.2, +4.7]	3/6/20
`base` vs `-S-J`	+3.97 [-0.07, +7.97]	+8.9 [+1.7, +15.9]*	9/3/17
Orchestration ablation (24h, 14 tasks)			
`base` vs `+D`	+2.88 [-2.94, +8.73]	+10.5 [-1.0, +22.0]	4/4/6
`base` vs `+P3`	-1.26 [-6.09, +3.21]	+3.3 [-8.2, +14.9]	4/3/7
`base` vs `+B+P3`	+7.59 [+0.99, +13.86]*	+22.4 [+10.8, +33.7]*	7/2/5
Additional ablation (12h, 14 tasks)			
`+N` vs `+N+D`	-2.43 [-7.77, +3.48]	-4.0 [-14.4, +6.0]	3/6/5
`+N` vs `+N+P3`	-0.94 [-6.73, +4.80]	-2.9 [-14.0, +8.5]	3/5/6
`+N` vs `+N+B+P3`	+2.55 [-2.55, +7.49]	+5.2 [-3.7, +14.2]	5/4/5
`+N` vs `+N+R`	+7.76 [+1.67, +13.87]*	+10.0 [-0.9, +20.4]	4/3/7
`+N` vs `+N+D+H+V`	-4.26 [-9.16, +0.98]	-4.0 [-12.9, +4.8]	3/5/6
`+N` vs `+N+s`	+0.79 [-3.86, +5.24]	+2.2 [-6.7, +11.0]	3/5/6

C.7 MALENA UNDER A CONSTRAINED CONTEXT BUDGET

We additionally test how Malena performs under an extremely short context budget with frequent summarization, on the full `fixed10` task subset, comparing against the standing, unconstrained Malena baseline on the same tasks. We compare the behaviour on GLM 5.2 running at its native 1M context limit, with an artificially constrained version that at whenever the session hits the 64k context limit, it triggers OpenCode's native session summarization compaction routine to reduce the context window.

Table 11: Avg. compactions before first medal.

Task	alaska2	cassava	champs	freesound	hubmap	imet	kuzushiji	multi-modal	nfl	petfinder
Avg	–	1.2	–	1.5	–	2.0	4.2	12.5	2.5	1.7

Table 12: **Malena under a shrunk context/output budget.** Any-medal rate (↑) and mean percentile (↑), with 95% CIs, self-selected and oracle (best-of-pool), on the full `fixed10` task subset (10 tasks $\times$ 4 seeds, $n = 40$ per arm).

	Self-selected		Oracle	
Group	Any-medal	Mean percentile	Any-medal	Mean percentile
Malena 1M	57.5% [42.2, 71.5]	77.1 [68.0, 86.2]	65.0% [49.5, 77.9]	78.8 [69.8, 87.8]
Malena 64k	50.0% [35.2, 64.8]	73.5 [65.3, 81.7]	50.0% [35.2, 64.8]	74.5 [66.3, 82.7]

Compaction fires reliably under the shrunk budget. Every shrunk-context run triggers OpenCode's built-in compaction at least once, at an average rate of 9.5 compactions over the 24h budget (it ranges from 3 to 63 compactions; over the $n = 40$ runs we observe 381 compaction events total). The context size right before compaction averages 65.4k tokens, tightly hugging the 64k threshold; the reset immediately after averages 15.6k tokens, i.e. each compaction discards roughly 76% of the accumulated context. Checking submission timing against compaction timing shows the run does not simply "beat" context pressure by locking in its medal before compaction kicks in: Table 11 counts, per task, how many compactions occur before a run first reaches any medal. Across the 20/40 runs that achieve a medal, the model keeps making progress after 3.25 compactions on average (range 0–19) before its first medal, so repeated compaction cycles do not by themselves prevent the run from reaching a medal.

Performance drops under the shrunk budget. Table 12 compares self-selected and oracle (best-of-pool) any-medal rate and mean percentile, with 95% CIs, between the full-context and shrunk-context Malena arms, on the full `fixed10` task subset ($n = 40$ runs per arm, 10 tasks $\times$ 4 seeds). Performance drops under the shrunk budget at both selection strategies, though the CIs overlap.

We read this drop as evidence that Malena is sensitive to operating near a hard context ceiling, though the point estimates remain within overlapping 95% CIs of the full-context arm.

C.8 Oneshot infrastructure ablation

We extend the experiments reported in Appendix C.6 and ablate different interventions in the Oneshot iteration. More information about these interventions in Appendix B.2. We report the average performance per iteration on GLM 5.2 and DeepSeek V4 Pro (Preview) backbones in Table 13. While most interventions provide a modest point-estimate improvement and only `H` resolving as statistically significant for GLM 5.2, and `I` for DeepSeek V4 Pro (Preview), only when running all of them together we can observe a larger performance improvement, as shown in Table 14.

D Additional experiments with external harnesses

D.1 Detailed external harness description

MLEvolve. (Du et al., 2026) MLEvolve maintains a *tree* of candidate solutions, each node a complete training script with a single parent, and expands it by UCT: nodes carry visit counts and a backpropagated reward, and selection descends by the usual upper-confidence-bound rule with an exploration constant that decays over the run. The backpropagated reward is not the task metric but a small hand-designed signal — a bonus for beating the incumbent, a penalty for a node that fails to run. Past the halfway point of the budget the selector blends UCT with rank-weighted sampling from a global, greedily-ranked top-K, so exploration gives way to exploitation on a schedule rather than through the search rule itself. Six operators produce nodes: `draft`, `improve`, `evolution` (improve using the branch's own trajectory), `debug`, and two cross-branch operators, `fusion` and `fusion_draft`, which synthesise a candidate from the best nodes of several branches. Crossover

Table 13: **Oneshot infrastructure ablation.** Average performance observed per Oneshot iteration reported with 95% CIs in the `default29` split. Top set: GLM 5.2 backbone. Bottom set: DeepSeek V4 Pro (Preview) backbone.

	Percentile (↑)	Medal rate (↑)
`base`	55.15 [54.31, 55.98]	36.8 [35.5, 38.2]
`+S+J`	56.58 [54.78, 58.41]	37.9 [34.5, 41.4]
`+S+J+N`	55.99 [54.18, 57.85]	37.1 [33.6, 40.1]
`+H`	57.08 [55.39, 58.80]	38.8 [35.8, 41.8]
`+D`	56.95 [55.12, 58.81]	39.2 [35.8, 42.7]
`+V`	54.47 [52.62, 56.38]	35.9 [32.4, 39.4]
`+S+J+N+H+D+V`	**57.94** [56.07, 59.77]	**41.8** [38.8, 44.8]
`base`	39.63 [38.10, 41.11]	20.3 [18.0, 22.4]
`+H`	39.99 [37.97, 42.07]	19.8 [16.8, 22.8]
`+V`	41.08 [38.92, 43.27]	21.1 [17.9, 24.6]
`+I`	**43.72** [41.94, 45.42]	**23.9** [21.0, 26.8]

Table 14: **Pairwise paired-by-task comparison.** Average per-iteration difference observed. Results reported using the `default29` task split. In an A vs B comparison, positive percentile differences indicate better average performance of A. Percentile difference reported along with 95% CIs. Top set: GLM 5.2 backbone. Bottom set: DeepSeek V4 Pro (Preview) backbone.

	Percentile (↑)	Medal rate (%, ↑)	
	Δ	Δ	won A/B/=
GLM 5.2			
`base` vs `+S+J`	-1.43 [-3.29, +0.57]	-1.1 [-4.7, +2.7]	9/10/10
`base` vs `+S+J+N`	-0.85 [-2.88, +1.22]	-0.2 [-3.9, +3.2]	9/10/10
`base` vs `+H`	-1.93 [-3.85, -2.9e-03]*	-2.0 [-5.4, +1.3]	7/10/12
`base` vs `+D`	-1.80 [-3.82, +0.23]	-2.4 [-6.0, +1.2]	9/10/10
`base` vs `+V`	+0.68 [-1.32, +2.82]	+0.9 [-2.7, +4.7]	8/10/11
`base` vs `+S+J+N+H+D+V`	-2.80 [-4.83, -0.74]*	-5.0 [-8.4, -1.5]*	5/13/11
DeepSeek V4 Pro (Preview)			
`base` vs `+H`	-0.36 [-2.85, +2.22]	+0.5 [-3.4, +4.0]	6/5/18
`base` vs `+V`	-1.45 [-4.12, +1.05]	-0.8 [-4.8, +3.0]	5/7/17
`base` vs `+I`	-4.10 [-6.46, -1.74]*	-3.6 [-7.3, -0.2]*	3/8/18

therefore flows through the *prompt* rather than the topology: a fusion node still has exactly one parent, with the other candidates injected as reference text, so the structure is a tree with cross-branch information flow and not a genealogical graph. Three candidates execute concurrently on one GPU. It commits, greedily and with no end-of-run re-ranking, whichever submission had the best self-reported validation metric at the time it was produced — a number an LLM extracts from the candidate's own stdout. It also keeps a hybrid BM25+FAISS retrieval memory over past attempts within a run, indexing each successful node by its plan and a code summary (and each debug node by the error it was repairing) and querying it when drafting, improving and debugging; we enable it in all runs reported here.

AiScientist. (Chen et al., 2026a) AiScientist has no search structure: it is a single conversation stepping through tool calls, and what to do next is whatever tool the model names. State persists in one mutable git repository plus two append-only markdown logs, one for implementation and one for experiments; whenever the agent implements or runs an experiment, the harness greps the most recent block of the other log and prepends it. This "file-as-bus" is the whole of its memory — there is no retrieval index, and the only other summarisation fires on context overflow. Its operators are `analyze_data`, `prioritize_tasks`, `implement` (whose modes cover full/fix/explore/refine/ensemble, so debugging is a mode rather than a separate operator), `run_experiment`, `spawn_subagent` and `submit`, alongside shell primitives. Work

is strictly sequential; subagents run blocking, and the only concurrency is whatever background training the model chooses to launch inside its sandbox. Most consequentially for Section 5.1, the final choice among candidates is made by the *agent* rather than by harness code: no code ranks candidates, but the agent is required, as a step before submitting, to compare its candidates against the registry and its experiment log and copy the best to the canonical submission path. It does so substantively — where it records the decision, it cites the cross-validation or hold-out protocol it designed, per-candidate scores under that protocol, and a rationale comparing them (for instance choosing a blended ensemble at OOF F_1 0.944 over its two components at 0.942 and 0.938). Its self-selected number is therefore a genuine self-selection on its own validation signal, as for the other methods; the difference is that the mechanism is prompt-driven rather than enforced, and the structured record of it is present in only a minority of runs.

Arbor. (Jin et al., 2026) A research agent built around a Coordinator/Executor loop over a persistent "Idea Tree": the Coordinator maintains the tree of candidate ideas and their outcomes, the Executor implements and evaluates a chosen idea in an isolated git worktree per experiment, and the loop expands whichever branch of the tree looks most promising. Two run modes exist upstream: a native CLI over a swappable backbone LLM, and a keyless coding-agent skill-suite mode where the host coding agent's own model does the reasoning; this paper uses the native-CLI mode.

ScienceFlow. (Zhao et al., 2026) A config/manifest-driven long-horizon ML-research agent (`scienceflow.cli`, verbs `repl`/`parallel`/`monitor`): a run is specified as a YAML task manifest and executed by one or more parallel exploratory workers under a chosen concurrency level, orchestrating both the agent's own reasoning calls and the ML training/evaluation code the workers produce.

D.2 Detailed MLE-bench results

In Table 15, we provide more details on the headline results presented in Figure 1 of Malena, along with the different external harnesses we ran, on the `fixed30` split in MLE-bench. Additionally, Table 16 shows the paired-by-task relative performance for Malena against all other methods. We find that Malena outperforms every presented baseline at the GLM 5.2 backbone across both performance metrics; obtains strictly better percentile scores for Kimi K3, and either beats or comes up as statistically indistinguishable at the DeepSeek V4 Flash. The only backbone where Malena presents a significant regression against other baselines is with Gemma 4 31B — the weaker of all models tested — where MLEvolve outperforms according to the percentile metric, while maintaining an equivalent medal rate at our current statistical power.

Two data caveats apply across methods. First, a run with no valid submission is imputed at the worst possible percentile rather than dropped (Section 5.1's convention); this creates exact ties between two very different failure modes and gives a handful of large-magnitude gaps outsized leverage on every method's point estimate or test statistic. Second, seed counts per task can be uneven, which the per-task-mean-based methods treat as equally reliable regardless of count.

D.3 Tuning effort invested in baseline harnesses

Because Malena is our own system, a natural concern is that we invested asymmetrically more integration and tuning effort into it than into the external harnesses it is compared against. This appendix reports that effort, spanning backbone-compatibility bug fixes and targeted configuration ablations on MLEvolve, ScienceFlow, and Arbor. In every case, this work either fixed a genuine defect — adopted into every canonical batch reported elsewhere in the paper — or found no reliable improvement, in which case we kept the harness's original canonical configuration and report the negative result here rather than omitting it.

MLEvolve: reasoning effort on DeepSeek V4 Flash. For every other backbone reported above, MLEvolve's canonical batches use that backbone's own `reasoning_effort=max` (or equivalent) setting, matching every other method's convention. On DeepSeek V4 Flash, we instead adopt `reasoning_effort=medium` as MLEvolve's canonical setting, the only backbone where it departs from `max`. The reason is a length effect specific to this backbone: at `max`, DeepSeek V4 Flash emits substantially longer reasoning traces than it does at `medium`. Because MLEvolve's tree search issues many independent, short LLM calls rather than growing a single session, per-call reasoning

Table 15: **MLE-bench performance with different harnesses and backbones.** Any-medal rate (↑) and mean percentile (↑), self-selected vs. oracle (best-in-pool), macro-averaged over the full `fixed30` set (mean over seeds within a task, then mean over tasks) at 24h time budget. A run with no valid submission counts as the worst possible outcome (no medal, worst percentile) rather than being excluded from its row's average. Brackets are 95% bootstrap CIs (seed-resampling, task set fixed). Sections are grouped by backbone model; **bold**/underline/*italic* rank the 1st/2nd/3rd best value per column within a section, among rows on that section's shared task set.

	Percentile (↑)		Medal rate (↑)	
	Self-select	Oracle	Self-select	Oracle
Kimi K3				
Malena	**72.75** [70.75, 74.82]	**75.54** [73.90, 77.35]	**60.0** [55.6, 64.4]	**65.6** [60.0, 70.0]
Arbor	68.46 [66.63, 70.08]	*69.53* [67.76, 71.25]	**60.0** [56.7, 63.3]	*61.1* [57.8, 64.4]
AiScientist	*64.15* [61.26, 67.10]	65.48 [62.54, 68.32]	*52.2* [46.7, 57.8]	53.3 [47.8, 58.9]
MLEvolve	63.89 [62.00, 65.74]	70.32 [68.98, 71.58]	50.0 [46.7, 54.4]	63.3 [60.0, 66.7]
GLM 5.2				
Malena	**69.74** [67.38, 72.23]	**71.56** [69.43, 73.66]	**62.5** [58.3, 66.7]	**65.0** [61.7, 68.3]
Arbor	*59.26* [56.37, 62.13]	60.70 [57.92, 63.54]	*45.8* [41.7, 50.0]	47.5 [42.5, 52.5]
AiScientist	61.39 [58.81, 63.81]	*62.35* [59.89, 64.74]	47.1 [42.3, 52.2]	*49.9* [45.2, 54.3]
MLEvolve	54.32 [52.04, 56.61]	64.87 [62.94, 66.88]	39.2 [35.0, 42.5]	50.0 [45.8, 53.3]
ScienceFlow	47.59 [44.47, 50.41]	58.74 [56.79, 60.92]	24.4 [18.9, 30.0]	40.0 [36.7, 43.3]
DeepSeek V4 Flash				
Malena	*58.53* [54.31, 62.43]	63.21 [59.72, 66.52]	*44.4* [37.8, 51.1]	48.9 [43.3, 54.4]
Arbor	**61.23** [57.88, 64.53]	**64.00** [61.26, 66.46]	**50.0** [45.6, 54.4]	**51.1** [46.7, 55.6]
AiScientist	59.13 [55.33, 62.97]	*59.45* [55.55, 63.32]	45.6 [40.0, 51.1]	*45.6* [40.0, 51.1]
MLEvolve	42.53 [38.67, 46.42]	54.81 [51.90, 57.60]	26.7 [21.1, 32.2]	40.0 [35.6, 44.4]
ScienceFlow	49.99 [46.61, 53.41]	54.79 [51.67, 57.94]	34.4 [28.9, 40.0]	40.0 [34.4, 45.6]
Gemma 4 31B				
Malena	*32.35* [29.29, 35.62]	*35.71* [32.53, 38.98]	*17.8* [14.4, 21.1]	*22.2* [17.8, 26.7]
AiScientist	33.00 [29.61, 36.45]	36.30 [32.98, 39.50]	20.0 [15.6, 24.4]	23.3 [20.0, 26.7]
MLEvolve	**42.88** [40.24, 45.44]	**50.73** [49.16, 52.34]	**22.2** [17.8, 26.7]	**31.1** [27.8, 35.6]

length translates almost directly into wall-clock cost per search step, so under `max`'s much longer traces MLEvolve completes only a fraction of the search steps it completes at `medium` within a fixed 24h budget.

To quantify this we compare `max` and `medium` on an identical 10-task subset (same tasks, same seed range, same 24h timeout for both arms; Table 17). Cutting reasoning effort to `medium` reduces the median reasoning length per LLM call by roughly 5× (37,124 to 7,069 characters), which more than triples the mean number of completed search-tree nodes per run (18.5 to 62.8). Per-node quality is essentially unchanged between the two settings — the buggy-node rate is within noise (61% vs. 62%) — so the additional search volume at `medium` is not, on average, lower-quality than `max`'s, just far more numerous. That additional search volume alone is enough to move outcomes substantially: any-medal rate rises from 0% to 25.0% (95% CI [10.0, 30.0]) and mean percentile from 42.6 (95% CI [36.0, 48.5]) to 53.1 (95% CI [44.9, 59.8]).

Independent tool-calling failures on GLM 5.2. Before either harness's canonical GLM 5.2 batch could run, we found and fixed two unrelated bugs that both manifested as tool-calling failures specific to this backbone. MLEvolve's structured-output calls request a named (forced) `tool_choice`; GLM 5.2 served behind our vLLM deployment enters grammar-constrained decoding under a forced choice, and its FSM has no transition for the reasoning tokens the model wants to emit before its answer, so every such call failed with an internal server error. We fixed this by defaulting MLEvolve's structured-output calls to `tool_choice=auto` (configurable per stage, so backends that do handle a forced choice reliably are unaffected). Independently, ScienceFlow's LLM client always included a `tools`/`tool_choice` field in its request payload, even for

Table 16: Paired Malena vs. baseline comparison across all models, macro-averaged per task, over the `fixed30` task set. Higher percentile numbers indicate stronger average performance for Malena.

	Percentile (↑)	Medal rate (%, ↑)	
	Δ	Δ	won A/B/=
Kimi K3			
Malena vs Arbor	+4.29 [+1.54, +7.05]*	+0.0 [-5.6, +5.6]	5/5/20
Malena vs AiScientist	+8.60 [+4.91, +12.15]*	+7.8 [+0.0, +15.6]	7/2/21
Malena vs MLEvolve	+8.85 [+6.11, +11.70]*	+10.0 [+3.3, +16.7]*	7/2/21
GLM 5.2			
Malena vs Arbor	+10.48 [+6.76, +14.29]*	+16.7 [+10.0, +22.5]*	12/1/17
Malena vs AiScientist	+8.36 [+5.12, +11.90]*	+15.4 [+8.7, +21.9]*	12/1/17
Malena vs MLEvolve	+15.42 [+12.05, +18.76]*	+23.3 [+17.5, +28.4]*	14/2/14
Malena vs ScienceFlow	+22.16 [+18.43, +26.11]*	+38.1 [+31.1, +45.0]*	19/0/11
DeepSeek V4 Flash			
Malena vs Arbor	-2.71 [-8.26, +2.50]	-5.6 [-13.3, +2.2]	6/8/16
Malena vs AiScientist	-0.60 [-6.10, +5.27]	-1.1 [-10.0, +7.8]	6/5/19
Malena vs MLEvolve	+16.00 [+10.38, +21.40]*	+17.8 [+8.9, +26.7]*	14/5/11
Malena vs ScienceFlow	+8.54 [+2.85, +13.58]*	+10.0 [+1.1, +17.8]*	10/3/17
Gemma 4 31B			
Malena vs AiScientist	-0.65 [-5.23, +3.83]	-2.2 [-7.8, +3.3]	3/3/24
Malena vs MLEvolve	-10.53 [-14.72, -6.36]*	-4.4 [-10.0, +1.1]	4/6/20

Table 17: **MLEvolve on DeepSeek V4 Flash, `max` vs. `medium` reasoning effort**, restricted to an identical 10-task subset and 24h timeout for both arms. Any-medal rate and none-rate are self-selected outcomes; mean percentile is oracle (best-in-pool). 95% CIs are bootstrapped over per-task run lists (2000 resamples), matching the convention used for the paper's headline tables. Reasoning length is a per-LLM-call median pooled across all calls in each arm's runs; node counts (mean/run) and the buggy-node rate are from each run's full search-tree journal.

reasoning_effort	Any-medal rate (%) ↑ [95% CI]	None-rate (%) ↓	Mean percentile ↑ [95% CI]	Median reasoning (chars) ↓	Mean nodes/run ↑	Buggy rate (%) ↓
max	0.0 [0.0, 0.0]	13.8	42.6 [36.0, 48.5]	37124	18.5	61
medium (canonical)	25.0 [10.0, 30.0]	2.5	53.1 [44.9, 59.8]	7069	62.8	62

calls with no tools to offer (e.g., its metric-output-interpreter step); the same GLM 5.2 deployment rejects an explicit empty `tools: []` array outright, which we fixed by omitting the field entirely rather than sending an empty array. These are two independent root causes, each surfaced by validating against a live endpoint before launching the canonical batch, not two symptoms of the same underlying fix.

ScienceFlow: worker count and resource-admission policy. ScienceFlow's canonical configuration runs multiple parallel exploratory workers sharing a single GPU (2 workers for GLM 5.2; a single worker gated by an LLM-based resource-admission heuristic for DeepSeek V4 Flash). Suspecting that this concurrency could itself be a source of resource congestion, we tested two independent mitigations, one per backbone.

On GLM 5.2, we compared 1 worker against the canonical 2 workers across the full `fixed30` split (Table 18). Any-medal rate and mean percentile are statistically indistinguishable between the two worker counts, so we treat the worker-count question as settled rather than under-powered, and kept the canonical 2-worker configuration.

On DeepSeek V4 Flash, we replaced ScienceFlow's canonical `resource_smart_llm` admission mode — an LLM-based admission heuristic plus a main-agent advisory signal that decides when a worker may start a new GPU job — with the static rule-based `resource_smart_policy`, on the same canonical 10-task ablation subset used by Appendix D.3's MLEvolve ablation, to check whether the static policy was leaving performance on the table (Table 19). It was not: any-medal

Table 18: **ScienceFlow (GLM 5.2), num_workers=1 vs. the canonical num_workers=2**, full `fixed30` coverage, with no data-variant confound between the two. Any-medal is self-selected, best-of-pool across every candidate ScienceFlow harvests per run; mean percentile is oracle (best-in-pool), flipped so higher is better. 95% CIs are bootstrapped over per-task run lists (2000 resamples).

Configuration	Any-medal rate (%) ↑ [95% CI]	Mean percentile ↑ [95% CI]
2 workers (canonical)	40.0 [33.3, 46.7]	60.7 [57.4, 64.6]
1 worker	43.9 [36.7, 50.0]	60.4 [56.1, 64.3]

rate and mean percentile are both somewhat lower under `resource_smart_policy`, though the two configurations ran at different n (29 vs. 37) and the gap is within the noise we see elsewhere at this sample size. We kept `resource_smart_llm` as canonical.

Table 19: **ScienceFlow (DeepSeek V4 Flash), resource-admission ablation**, identical 10-task subset and 24h timeout as Appendix D.3's MLEvolve ablation. Any-medal is self-selected, best-of-pool across every candidate ScienceFlow harvests per run; mean percentile is oracle (best-in-pool), flipped so higher is better, matching every other percentile column in this paper. 95% CIs are bootstrapped over per-task run lists (2000 resamples), matching the convention used for the paper's headline tables (Section 5.1).

Configuration	Any-medal rate (%) ↑ [95% CI]	Gold (%)	Silver (%)	Bronze (%)	Mean percentile ↑ [95% CI]
`resource_smart_llm` (canonical)	28.3 [10.0, 50.0]	10.0	13.3	5.0	66.6 [52.9, 80.7]
`resource_smart_policy`	20.0 [0.0, 40.0]	2.5	10.0	7.5	52.2 [39.6, 65.5]

Arbor: executor timeout and convergence early-stopping (GLM 5.2). Arbor's canonical configuration for both GLM 5.2 and DeepSeek V4 Flash uses `reasoning_effort=max`, a 1M-token context window, 3 parallel executors, and a 4-hour timeout per individual experiment inside its 24h wall-clock budget; the executor timeout and the `stop_after=8` convergence early-stopping threshold ablated below are not values we chose — both are ported unmodified from the `mle_kaggle` competition-optimization plugin bundled in Arbor's own public GitHub repository, so these ablations test Arbor's own maintainers' defaults rather than a configuration we picked ourselves. We ran two targeted ablations against this configuration. First, we doubled the executor timeout to 12 hours; the medal rate is unchanged (Table 20). Second, we tried relaxing Arbor's convergence-based early stopping (default `stop_after=8` consecutive non-improving idea-tree expansions) to `stop_after=999`, effectively disabling it so a run keeps exploring until its wall-clock budget runs out rather than stopping early: medal rate is again unchanged within noise (Table 20).

Table 20: **Arbor (GLM 5.2), executor-timeout and convergence ablations** against the canonical 4-seed baseline, matched per task. Any-medal is self-selected, best-of-pool across every scored idea-tree node. 95% CIs are bootstrapped over per-task run lists (2000 resamples), matching the convention used for the paper's headline tables.

Ablation	Baseline any-medal rate (%) ↑ [95% CI]	Ablation any-medal rate (%) ↑ [95% CI]
Executor timeout: 4h → 12h	12.5 [0.0, 50.0]	12.5 [0.0, 50.0]
Convergence early-stop: `stop_after` 8 → 999	50.0 [0.0, 100.0]	43.8 [25.0, 75.0]

D.4 Trajectory and token usage with Kimi K3

Figure 4's two panels are GLM 5.2 only. Figure 10 reproduces both at Kimi K3 using the same `fixed30` task set.

The overall shape matches Figure 4: quality rises quickly in the first few hours and plateaus thereafter, with AiScientist again the slowest to converge. On cost, Malena's cache-read volume again dominates its token footprint the same way it does at GLM 5.2, but Kimi K3's much higher output pricing ($13.99/M vs. GLM 5.2's $3.123/M) makes MLEvolve's uncached, output-heavy tree search the most expensive of the four shown here despite the understated reasoning caveat above.

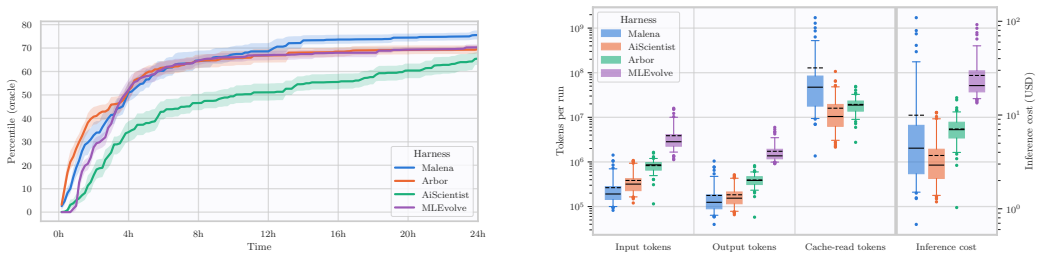

Figure 10: **Left:** oracle best-so-far vs. wall-clock time at Kimi K3 for Malena, Arbor, AiScientist, and MLEvolve. **Right:** per-run distribution of cumulative input, output(+reasoning), and modeled cache-read tokens, plus a USD-cost group at Kimi K3 pricing, for Malena, Arbor, AiScientist, and MLEvolve. MLEvolve's cache group is omitted (always exactly 0, same reason as the GLM 5.2 figure).

D.5 Quality by task difficulty

In order to better understand better the strengths of Malena against other baselines, and better investigate its improved performance, we partition the `fixed30` split in four different cluster, according to perceived difficulty among all harnesses:

1. `nomad2018`, `plant`, `spooky`, `stanford`, `us`
2. `aptos2019`, `freesound`, `h&m`, `hotel`, `hubmap`, `mlsp`, `tweet`, `whale`
3. `billion-word`, `cassava`, `imet`, `kuzushiji`, `multi-modal`, `nfl`, `petfinder`
4. `alaska2`, `champs`, `hms`, `jigsaw`, `new`, `osic`, `smartphone`, `tensorflow2`, `uw`, `ventilator`

We plot the individual oracle performance among these clusters in Figure 11. The easiest-to-solve (cluster 1) and hardest-to-solve (cluster 4) exhibit a very similar trend among all methods. We observe that, on cluster 2, MLEvolve and Malena have a slight edge over the other methods. However, Malena performs significantly better than all other baselines on the 3rd cluster, suggesting a sweet spot between tasks that are inherently borderline impossible to solve by current AI agents, and tasks that are trivial.

D.6 NatureBench: evaluation protocol

For NatureBench, we follow the benchmark's own online evaluation protocol as published (Appendix A.3): rather than a single held-out grader run once at the end of a run, as in MLE-bench, each task exposes a live `/evaluate` HTTP endpoint that an agent may query an unbounded number of times over the course of a run, with the endpoint automatically tracking the best score seen across all calls. We give every harness we compare here — Malena, AiScientist, and MLEvolve — unrestricted access to this endpoint, matching NatureBench's protocol as published.

We extend the per-run time budget to 8h, double NatureBench's own published default of 4h, to give the slower of our search-based baselines (MLEvolve, whose tree search advances through many short, sequential steps rather than one long session) a more comfortable margin within which to complete meaningful search.

Because the harness's own protocol reports the best score achieved anywhere in the run — an oracle score, as described in Appendix A.3 — rather than a self-selected final submission, this metric implicitly rewards however many independent attempts a harness manages to get graded within the time budget, on top of whatever quality gains its search process makes. Table 21 reports the number of `/evaluate` calls per run for each harness, across our full NatureBench run cohort. Malena and AiScientist call the endpoint at a broadly similar rate (median 16 and 18 calls per run, respectively); MLEvolve calls it substantially less often (median 8 calls per run). Since a benefit under an oracle-best-of-run metric scales with the number of independent scoring opportunities

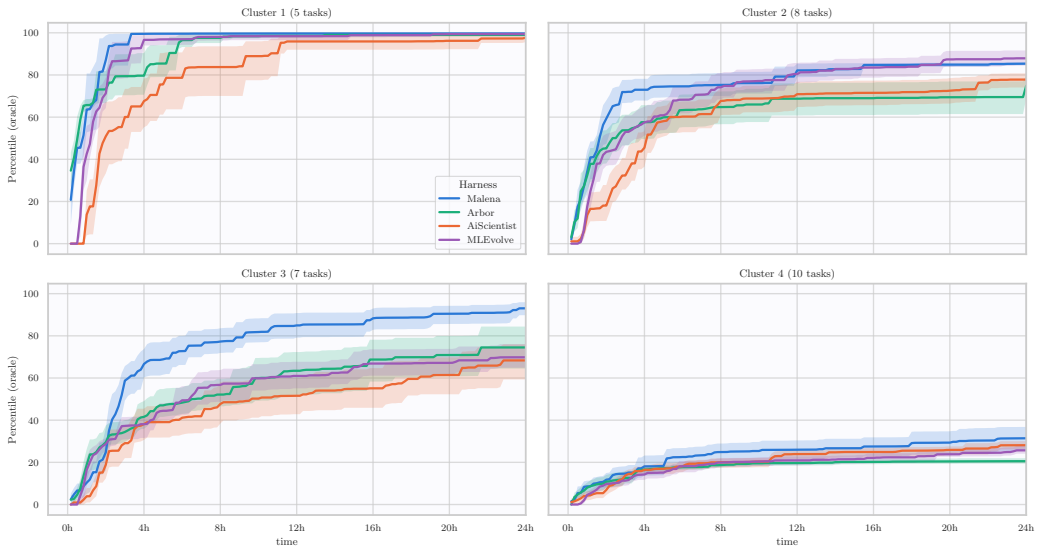

Figure 11: Oracle best-so-far vs. wall-clock time, among different task cluster. Malena strongly leads cluster 3 and gives a small edge on cluster 4, while it remains close in the other two clusters.

exercised, this asymmetry may put MLEvolve at a mechanical disadvantage on this particular metric, independent of any difference in the quality of the strategies it discovers.

Table 21: **Number of `/evaluate` calls per run on NatureBench.** 25th/50th/75th percentiles across our full NatureBench run cohort, pooling both the GLM 5.2 and Kimi K3 backbones.

	Malena	AiScientist	MLEvolve
p25	8	7	2
p50 (median)	16	18	8
p75	36	96	18

E Trace Analysis

This appendix gives the full methodology behind Section 5.2's checkpoint labeling and the technique-overlap analysis it enables, including the labeling schema, the offline deduplication/clustering procedure, and technique-name normalization we omitted from the main text for space. The checkpoint-mining and labeling steps below (through canonicalization) are the current pipeline behind Figure 6 and Figure 5. Unless stated otherwise, the labeling/judge model throughout is GLM 5.2.

E.1 Checkpoint mining

Checkpoints. We unify run traces across methods to a common checkpoint format: a snapshot of the run's code repository at specific points in time, method-dependent since each harness structures its own trajectory differently – for Malena, immediately before a long job submission, a long bash tool call, or a submission-registration tool call; for MLEvolve, before executing a candidate; for AiScientist, before a long bash tool call; for Arbor, before a job-submit call. Oneshot and Completions have no intermediate trajectory to snapshot and are labeled once, on the final state of the repository. Runs with multiple checkpoints are labeled sequentially along this chain, each job seeded with the entries already logged for its nearest earlier checkpoint so it only has to add, remove, or revise entries rather than re-deriving the whole technique set from scratch.

Labeling schema (pass 1: technique extraction). The judge (GLM 5.2) reads a checkpoint's code and logs zero or more entries, each with: a free-form `snake_case` name (no reuse-search against prior entries – naming convergence is handled later, offline); 1–3 `categories`, ordered most-to-least applicable, drawn from a fixed 13-way taxonomy (`data_preprocessing`, `data_pipeline`, `feature_engineering`, `augmentation`, `architecture`, `loss`, `regularization`, `optimization`, `training`, `ensembling`, `model_selection`, `post_processing`, `inference`, plus an `other` escape hatch (used for only 0.02% of all logged category-tag annotations across our labeled checkpoints), which we exclude from the category-share figures throughout this section); and three separate text fields instead of one blob – `mechanism` (task-agnostic, roughly 40 tokens, describing only WHAT IS COMPUTED, stripped of domain nouns and hyperparameters, the only field ever embedded for similarity/dedup), `instantiation` (task-specific detail, hyperparameters, and domain nouns, kept for search but never embedded), and `evidence_sig` (a 1–3 line canonical code excerpt for human/reviewer sanity-checking). Splitting mechanism from instantiation this way removes a specific noise source found in an earlier embedding-similarity audit, where chunk embeddings picked up code-context co-occurrence (e.g. dropout/mlp/layer_norm clustering together purely because they appear near each other in a forward pass) rather than technique identity.

Deduplication and canonicalization (offline, per task). All identity resolution – deciding whether two independently-logged entries describe the same underlying technique – happens offline, never at labeling time, and entirely within one task's own runs (never pooled across tasks, so no task's vocabulary is built from or reviewed against another task's runs). Pass-1 entries are embedded on their `mechanism` text using `Qwen3-Embeddings-8B` and grouped by complete-linkage hierarchical clustering at a fixed cosine-similarity threshold (0.70, picked by sweeping method $\times$ threshold against a hand-audited corpus for cluster-size and same-run-duplicate violations). An automated GLM 5.2 review then runs in two phases: a split-review phase flags multi-member clusters with no category shared by every member and asks GLM 5.2 whether any member should be extracted into its own cluster; a merge-review phase takes every singleton cluster and asks the judge whether it is really the same technique as one of its nearest neighbours by mechanism-embedding similarity. A final canonicalization pass has GLM 5.2 synthesize one canonical name and mechanism description per cluster that a reader would recognize as covering every member's variant, rather than mechanically picking one member's text. Pass 2 then re-labels every run a second time, this time against the finalized, canonical cluster set: the judge calls `log_known_technique(cluster_id, ...)` directly when a checkpoint's technique matches an existing cluster, and free-form entries otherwise, so a pass-2 match is never subject to a separate, error-prone post-hoc nearest-centroid assignment step.

E.2 Main-text plot methodology

This subsection gives the exact construction behind the two main-text plots built on top of the canonical technique clusters above: the category-share-decay curves (Figure 6) and the per-technique rarity distribution (Figure 5).

Category-share decay (half-life). Figure 6 tracks a method's technique-category composition as a function of *checkpoint fraction* $x \in [0, 1]$ (progress through a run, pooled over every checkpoint of every run of that method). Every technique discovered at fraction $f \leq x$ contributes weight $0.5^{(x-f)/h}$ to x's composition, for half-life h: a technique discovered exactly at x has full weight 1, one discovered h earlier has weight 0.5, and older ones decay smoothly toward (but never reach) 0. Weighted category-tag counts are normalized to shares at each x. $h \to \infty$ recovers a plain cumulative count of every technique discovered so far; $h \to 0$ recovers an immediate, local snapshot of only the most recently discovered techniques. The main text uses $h = 0.25$; Appendix E.3 below checks smaller half-lives and per-task (non-pooled) curves.

Technique rarity. Figure 5 reports, per method, the distribution of how rare the canonical technique clusters that method's runs exhibit are. Rarity is computed independently *per task*, not globally: for a fixed task, pool every run of every (method, backbone) group on that task, and for each cluster c let n_c be how many of those pooled runs exhibit it; c's rarity is $1 - n_c/n_{\text{total}}$, the fraction of that task's pooled runs that did *not* use it. Pooling within the task across every group (rather than, say, comparing a group only to one fixed baseline) means a technique that happens to be shared by two groups is not counted as rare for either of them. Figure 5 plots, per method, the resulting distribution

of these per-technique rarity values (pooled across backbones and 10 tasks) into six windows centred at rarity $0, 0.2, \ldots, 1.0$ (half-width 0.1).

E.3 Robustness of the category-share-decay curves

Figure 6 in the main text pools all labeled tasks at a single half-life (0.25). Figure 12 shows the same pooled construction at three half-lives closer to the main text's value (0.1, 0.15, 0.2), and Figures 13 and 14 show the identical three-half-life construction on two individual tasks (`aptos2019` and `nfl`) instead of pooled, to check that the shape of Malena's late-run shift toward ensembling/model_selection is not an artifact of the particular half-life or of pooling across tasks.

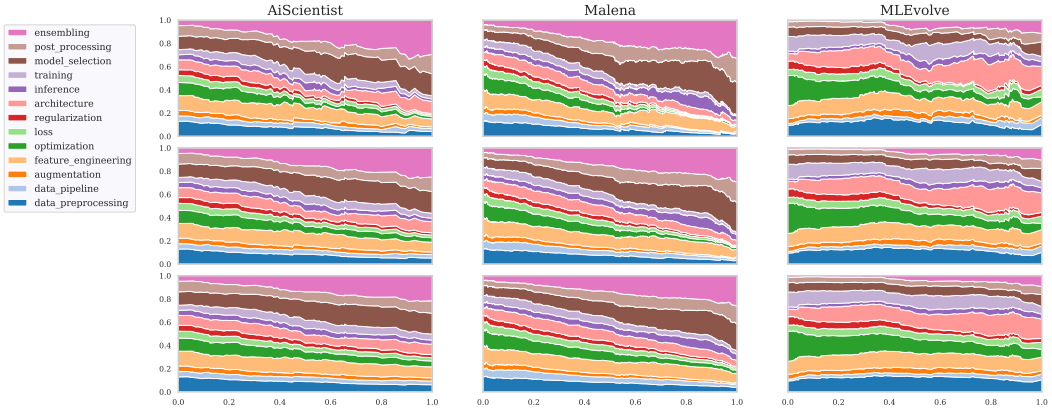

Figure 12: Pooled category-share-decay curve (all labelled tasks) at half-lives 0.1, 0.15, and 0.2 (rows, top to bottom) – smaller half-life is a more local, noisier snapshot of recently-introduced techniques. As in Figure 6, the x-axis is checkpoint fraction through the run ($\in [0, 1]$) and the y-axis is the decayed share of technique-category tags at that point.

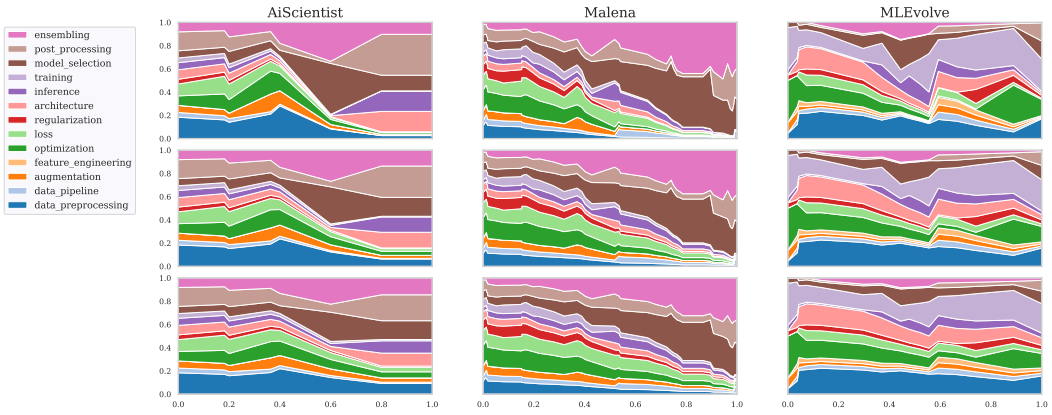

Figure 13: Category-share-decay curve on `aptos2019` alone, at half-lives 0.1, 0.15, and 0.2 (rows, top to bottom). As in Figure 6, the x-axis is checkpoint fraction through the run ($\in [0, 1]$) and the y-axis is the decayed share of technique-category tags at that point.

E.4 Per-task technique-category composition across methods

Figure 6 and Figure 5 both look at a method's technique choices pooled across all 10 tasks – respectively how its category mix shifts over the course of a run, and how rare its techniques are relative to every other group. A complementary question is whether methods reach for a different mix of technique *categories* than each other on the *same* task, i.e. whether task identity or method identity is the bigger driver of what a method tries. Here we look directly at the technique-category composition of each method's repertoire on each task independently.

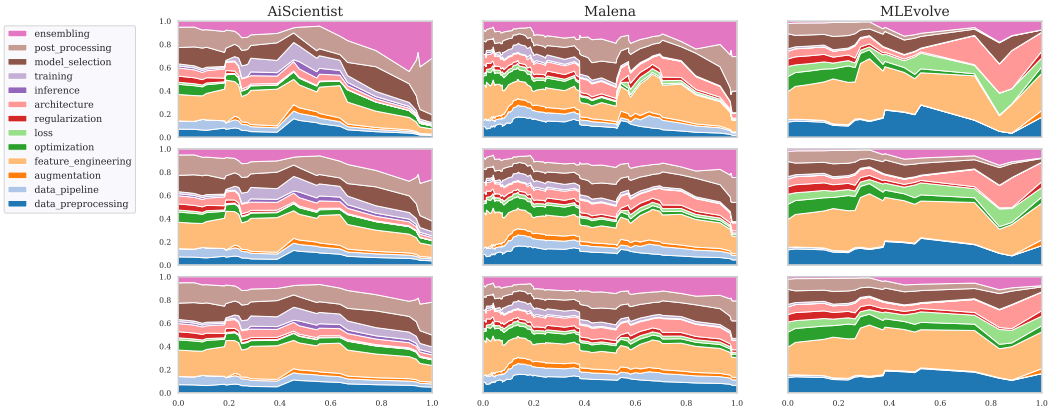

Figure 14: Category-share-decay curve on `nfl` alone, at half-lives 0.1, 0.15, and 0.2 (rows, top to bottom). As in Figure 6, the x-axis is checkpoint fraction through the run ($\in [0, 1]$) and the y-axis is the decayed share of technique-category tags at that point.

For a given (task, method) pair, we take the set of every distinct canonical technique cluster (Appendix E.1 above) any of that method's backbone/seed runs on that task exhibit, and tally, for each category tag in the taxonomy above, how many of those clusters carry it (a cluster tagged with 2 categories contributes to both – categories are not split fractionally, so the resulting bar shows a share of category-*tags*, not of techniques). This is a presence-based snapshot (did the method ever try a category-tagged technique at all over the course of the run), not usage-frequency-weighted like Figure 6's decay curves – deliberately simpler, since here we only need one static composition profile per (task, method) rather than a time series.

Figure 15 shows the resulting composition for all 10 tasks. Two things stand out. First, task identity is a strong driver on its own: `feature_engineering` dominates every method's profile on `champs`, `h&m`, and `nfl`, while `ensembling` dominates on `petfinder` and, for Malena and AiScientist specifically, on `aptos2019`, `cassava`, and `jigsaw` as well. Second, on top of that shared task-driven baseline, Malena and AiScientist's profiles track each other closely on most tasks (e.g. `aptos2019`, `cassava`, `freesound`, `nfl`), while MLEvolve is the most frequent outlier – visibly heavier on `data_preprocessing` and lighter on `ensembling` on several of those same tasks, consistent with its pooled under-representation of ensembling growth in Figure 6. The task `jigsaw` shows the largest cross-method spread: Oneshots' profile there is missing most of the `ensembling` share the other three methods have, replaced by a much larger `loss`/`optimization` block.

E.5 Illustrative session excerpts

The excerpts below are verbatim (only trimmed for length, marked with `...`) pulls from real session transcripts, given as illustrative examples of the reasoning/tool-call structure behind the checkpoint-labeling claims in Section 5.2. `[bash]` lines are literal shell commands the agent issued; `<reasoning>` blocks are the agent's own prose immediately before or after them.

Example 1: Malena, GLM 5.2, `tweet`.

```
<reasoning>
...Let me train deberta with 4 epochs using pseudo_test_v2, tag debv3pl2...
</reasoning>
[bash] python src/train.py --model microsoft/deberta-v3-large --tag debv3pl2
       --lr 1e-5 --epochs 4 --extra_train work/pseudo_test_v2.csv
-> [fold 0 epoch 3] val_jaccard=0.7211 ... best val_jaccard per fold
<reasoning>
Iterated pseudo-labels + 4 epochs: OOF = 0.7212 (vs 0.7199 for v1 pseudo + 3
epochs). +0.0013 improvement. Fold scores: fold 0: 0.7223 -> 0.7211 (slightly
down); fold 1: 0.7123 -> 0.7139 (up); fold 2: 0.7237 -> 0.7249 (up); fold 3:
0.7169 -> 0.7189 (up); fold 4: 0.7245 -> 0.7271 (up). Overall OOF 0.7199 ->
```



Figure 15: Technique-category composition of each method's technique repertoire, per task (backbones pooled, `completions` excluded). The x-axis is method (Malena, AiScientist, MLEvolve, Oneshots); the y-axis is share of category-tags. Bar segments are shares of category-tags (a technique tagged with multiple categories counts toward each), not shares of techniques.

```
0.7212. Let me compute S1 and per-sentiment for debv3pl2.
</reasoning>
[bash] python src/infer.py --model microsoft/deberta-v3-large --tag debv3pl2
       --out submission_v5_debv3pl2.csv
[submissions_register] Registered submission... Total number of submissions:
5. Best validation score so far: 0.7212.
```

Example 2: MLEvolve, GLM 5.2, `aptos2019`.

```
<reasoning>
The evolution trajectory reveals severe overfitting (train loss drops below
0.10 while validation loss plateaus > 0.60), indicating that the highly
capable ConvNeXt base model easily memorizes this small dataset. To break
this stagnation, we must apply a Tier 2 shift focusing on robust
regularization rather than just tweaking losses or backbones. By
introducing Mixup data augmentation, stochastic depth (Drop Path), and
feature-level dropout, we force the model to learn smoother, generalized
decision boundaries between severity grades, effectively combatting
single-landscape overfitting.
</reasoning>
```

Example 3: AiScientist, GLM 5.2, `aptos2019`.

```
<reasoning>
Now I have a clear picture. Let me summarize the state and plan refinements:

Current state:
- 8 models, 5-fold CV, ensemble OOF QWK = 0.9290
- Best single models: cnexts_384 (0.9236), efnb5_512 (0.9231)
- Models highly correlated (0.96-0.99) -- limited diversity gain
- Huge raw->tuned QWK gap (0.09-0.13) -- continuous scores poorly calibrated
- Training is noisy (raw_qwk bounces across epochs)
- Training takes ~130 min/model

Refinement plan (low-risk, high-impact, no broad rewrite):
1. Ensemble weighting refinement (no retraining): rank-averaging,
   z-score normalization, greedy forward selection -- cheap, could
   push 0.9290 -> 0.93+
2. Threshold tuning quality -- better optimization
3. Training stability (one refined retrain): SWA, be... [truncated in log]
```

```
</reasoning>
[bash] cd /home/code && python3 -c "
import pandas as pd, numpy as np
from utils import OptimizedRounder, qwk
from scipy.optimize import minimize

oof = pd.read_csv('artifacts/ensemble_kfold_oof.csv')
model_cols = ['score_efnb5_456','sco... [truncated in log]
-> [tool_result] result_len=1233 (stdout not recoverable in this log format)
```

This is the final record of the subagent's session (30 total records): after inspecting all 8 trained models' summaries/histories over 9 prior turns (git log, config/dataset/model/utils reads, per-model kfold summaries, training histories, GPU/torch versions), it explicitly frames the decision as "no broad rewrite" and chooses an ensemble-reweighting analysis over `artifacts/ensemble_kfold_oof.csv` rather than editing any training code (14 bash + 6 `read_file_chunk` calls, 0 `edit_file` calls across the whole subagent session).

Example 4: Malena, GLM 5.2, `whale` – the two-stage pipeline referenced in Section 5.2. Within its first 17 checkpoints, Malena settles on a two-stage pipeline and never changes its architecture again over the remaining 63 (80 total): a metric-learning embedder, followed by nearest-neighbor retrieval with a learned novelty detector.

```
<reasoning> (checkpoint 000)
...Compute embeddings for all training images and test images... find nearest
training images... Insert new_whale based on threshold... Use validation
set to find optimal threshold.
</reasoning>
```

Stage 1 (`src/train.py`) trains a CNN/ViT backbone with an ArcFace loss to produce normalized whale-fluke embeddings. Stage 2 (`src/retrieve.py`, `src/centroid_submit.py`, `src/detector.py`) computes cosine-similarity nearest neighbours against the training embeddings, ranks top-5 candidates via per-class centroids, and uses a small logistic-regression classifier over similarity-gap features (top-1/2/3 similarity, gaps, top-5 mean/std) to flag `new_whale` queries – individuals never seen in training. All 63 subsequent checkpoints are targeted patches to this fixed pipeline rather than new stages: backbone swaps (EfficientNetV2 $\rightarrow$ ConvNeXt $\rightarrow$ Swin), input-resolution bumps, novelty-threshold retuning, and ensembling. One checkpoint (`014_ms_centroid_fAB`) illustrates the granularity of these patches:

```
<reasoning> (checkpoint 014_ms_centroid_fAB)
...The gap between 0.663 and 0.722 oracle is the new-detection error. A
better new-detector could use... but let me first improve the embedding...
</reasoning>
```

That is, the agent is choosing which fixed sub-module of the pipeline to improve next, at the granularity of a targeted code change, rather than tuning a numeric hyperparameter or introducing a new pipeline stage.

Example 5: independent convergence on soft-label self-training, `aptos2019` (GLM 5.2 vs. Kimi K3). Malena's GLM 5.2 and Kimi K3 runs on `aptos2019` independently converge on the same canonical technique cluster – soft-label self-training (pseudo-labelling using an ensemble's own predictions as soft targets) – without sharing a run, then diverge in how much surrounding machinery each backbone builds around it. This cluster has rarity 0.9 (Appendix E.2): across every AiScientist, MLEvolve, and Oneshot run on this task, none use it.

The GLM 5.2 run (logged as `pseudo_label_self_training`):

```
Mechanism: Generate soft pseudo-labels by running a trained model on
unlabeled data, combine the pseudo-labeled samples with the original
labeled training set (using soft target distributions and a per-sample
weight for pseudo-labeled examples), and retrain the model on the
augmented dataset.

Instantiation: dataset.py PseudoDataset combines train images (hard
labels, soft-onehot with label smoothing) and pseudo-labeled test images
```

```
(soft probs from ensemble predictions). train_pl.py orchestrates per-fold
pseudo-label training. Pseudo-labels sourced from infer.py
--save_test_probs output.
```

The Kimi K3 run (logged independently as `self_training_pseudo_labels`), landing on the identical mechanism:

```
Instantiation: train_pseudo.py loads ensemble test probability
predictions (from compute_ensemble_test_probs.py) as soft pseudo-labels
for unlabeled test images. Trains jointly on labeled train data (one-hot
hard targets) and pseudo-labeled test data (soft 5-class probs) via soft
cross-entropy... Pseudo data influence controlled by --pseudo_repeat
(dataset repetition factor)... Supports per-fold warm-start via
--init_from_dir.

Evidence:
comb = ConcatDataset([train_ds_hard, *([train_ds_pseudo] * cfg.pseudo_repeat)])
loss = -(targets * logp)
if weight is not None: loss = loss * weight[None, :].to(device)
```

The two runs share the core mechanism but not its surroundings. Before being folded into the single canonical cluster above at the pass-1 labeling stage, the GLM 5.2 run builds substantially more machinery around it: a weighted soft-cross-entropy loss (pseudo-labeled samples down-weighted 0.2 vs. 1.0 for labeled ones), pseudo-label *sharpening* (raising soft probabilities to a power > 1 and renormalizing to peak the target distribution), iterative multi-round pseudo-labeling (retraining on a previous round's ensemble output, then keeping whichever round scored better per model on OOF), and two rounds of brute-force combinatorial search – first over which round/resolution each ensemble member should use (up to 324 combinations, each scored by out-of-fold threshold-optimized QWK), then over 7 candidate final ensemble configurations. The Kimi K3 run instead builds a leaner version of the same core loop, adding per-fold warm-starting (`--init_from_dir`) rather than the GLM 5.2 run's iterative-round search. Both runs independently rediscover the same underlying technique – reflected in Figure 5 as a rare (0.9) but recurring point in Malena's rarity distribution – while differing in how much extra engineering each backbone invests around it.

F CONTAMINATION CHECK

Every `fixed30` task is a public Kaggle competition, and MLE-bench itself ships a set of reference kernels/notebooks for a subset of its competitions (public solutions collected as part of the benchmark's own construction). We check for memorized/copied code using a vendored copy of Dolos, MLE-bench's own source-code similarity tool, against a curated corpus of 1,350 distinct real solutions spanning all 30 `fixed30` tasks.

Code similarity (Dolos). Dolos's default pairwise *similarity* score normalizes shared token k-grams by the union of both files' tokens (a Jaccard-style measure), which is a poor fit for our setting and, notably, biased in the direction of *under*-reporting contamination rather than over-reporting it: Malena's solution files are typically much smaller than the monolithic Kaggle notebooks in the real corpus, so even if Malena fully reused a small fragment of a large real file verbatim, the shared tokens would be a small fraction of the (large) union and the similarity score would stay low – the metric would fail to flag exactly the kind of partial copying we care about. We instead score every pair by *containment* – the fraction of the *smaller-file's* tokens that also appear in the other file, together with the longest matching token run – which asks "how much of this file was drawn from the other", independent of how large the other file is, and is sensitive to exactly the small-fragment-fully-reused case that raw similarity would miss. We compare Malena-vs-real containment against a real-vs-real cross-label null baseline (distinct, independently-written real solutions, pooled across the whole 30-task corpus rather than restricted to same-task pairs – see below for why). Table 22 reports the containment distribution for both, across all 924,750 Malena-vs-real pairs and 473,850 real-vs-real cross-label pairs (every Malena checkpoint / every real solution, pooled across all 30 tasks – i.e. including task-mismatched pairs, not just same-task ones – exhaustive): Malena-vs-real containment sits below real-vs-real containment at every percentile shown. A pair is flagged only if it exceeds *both* the real-vs-real containment and longest-match-run 95th percentiles; **zero of the**

924,750 pairs are flagged. This is the pattern we would expect from independently-written code, not from copied or memorized solutions. (The embedding check below restricts to same-task pairs only, to keep the $O(n^2)$ embedding-similarity cost tractable; the two pair counts are therefore not directly comparable.)

Embedding similarity (bag-of-chunks). As a second, token-independent check, we embed both corpora with `Qwen3-Embedding-8B` (Zhang et al., 2025), which is a 4096-dim embedding model, splitting each file into a sliding window of chunks (since many solution files exceed the embedding model's context window) and representing each solution as a bag of chunk embeddings rather than a single vector. We compare bags with three metrics: *mean-pooling cosine* (cosine similarity of the two bags' mean embeddings); *Chamfer similarity* (for each chunk in one bag, its max cosine similarity to any chunk in the other bag, averaged over the bag, symmetrized by averaging both directions); and *Chamfer with background subtraction* (the raw Chamfer score minus a background level – the median Chamfer score over 5,000 random Malena-vs-real chunk-bag pairs drawn from *different* tasks, 0.504 – to remove the generic code/library-idiom similarity any two unrelated files share). We compute all three over every same-task Malena-vs-real pair (62,207 pairs) and every same-task real-vs-real cross-label pair (33,465 pairs), exhaustively, across all 30 tasks. Table 22 reports the percentiles alongside the Dolos numbers above. At the median (and, for mean-pooling cosine, up to the 90th percentile), Malena-vs-real similarity is comparable to the real-vs-real baseline rather than below it; but at the percentiles where copied or memorized content would actually show up – the upper tail, p95 and p99 – Malena-vs-real similarity is consistently *below* the real-vs-real baseline for all three metrics, i.e. there is no anomalously heavy right tail. This is the same qualitative conclusion as the Dolos check, from an independent, token-free signal.

Table 22: Contamination-check similarity scores by percentile: Malena-generated code vs. real `fixed30` solutions, compared against a real-vs-real cross-label null baseline. Top: Dolos code-similarity containment. Bottom: bag-of-chunk-embedding similarity.

metric	group	p50	p90	p95	p99
Dolos containment	real-vs-real	0.008	0.077	0.143	0.453
Dolos containment	Malena-vs-real	0.002	0.009	0.013	0.020
mean-pooling cosine	real-vs-real	0.735	0.877	0.909	0.968
mean-pooling cosine	Malena-vs-real	0.773	0.880	0.896	0.919
Chamfer	real-vs-real	0.634	0.741	0.778	0.882
Chamfer	Malena-vs-real	0.638	0.713	0.726	0.746
Chamfer (bg-subtracted)	real-vs-real	0.130	0.237	0.273	0.377
Chamfer (bg-subtracted)	Malena-vs-real	0.134	0.209	0.222	0.242

Behavioral check: environment interaction matters. A separate, model-independent argument against memorization comes from comparing iteration types rather than code directly: if Malena's solutions were largely memorized from pretraining, a single completion asked to produce a full solution to a competition it has "seen" should score close to the full agentic loop, since no environment feedback would be needed to recall a memorized answer. We can clearly observe that such thing is not the case in our corpus. From Figure 2, we can observe that even Best-of-N Oneshot iterations aggregated at 6h time budgets performs significantly better than the single Chat iteration. This is the opposite of what contamination would predict, and is consistent with the agent doing real iterative problem-solving rather than recalling known answers.

G Discussion

G.1 Limitations

The primary limitation of this study is that most of our results rely on MLE-bench (Chan et al., 2025), which has known issues in its task preparation scripts and, being built from public Kaggle competitions, is a plausible target for pretraining contamination. We mitigate this in three ways. First, we identified and fixed the subset of `fixed30` tasks with broken preparation scripts (missing data or ground-truth leakage) rather than excluding them, so that our comparisons stay on the full 30-task set

used elsewhere in the paper; see Appendix A.1 and Table 3 for which tasks were fixed. Second, we directly checked for contamination via two independent methods, a code-similarity (Dolos) check and a bag-of-embeddings similarity check, both against a curated corpus of real `fixed30` solutions and both finding no anomalous similarity between Malena-generated code and real solutions (Appendix F). Third, we additionally evaluate on NatureBench (Wang et al., 2026b) (Appendix A.3), whose tasks are recent open scientific-research questions with no public solution corpus to memorize, though there we could only afford smaller models and fewer methods than on `fixed30`.

A second limitation is statistical power. Some of our results still have confidence intervals wide enough that we cannot make strong claims about the size of an effect; we believe our weaker claims of Malena being no worse than a given baseline are well supported (Appendix A.5 reports the paired-comparison methodology this relies on), but stronger claims about exactly how much better should be read with this in mind. This is a direct consequence of cost: running agents for the full 24-hour budget is expensive, so we could only afford 3–4 seeds per task at 24 hours, versus 6–10 seeds per task for our 6-hour and completion-only conditions.

A third limitation is scope on coordination between multiple agent workers. Our claim that most harness interventions add little on top of a plain coding agent (Section 4) is made almost entirely with a single worker; we do run one coordination ablation (Section 4.4) across a small set of coordination primitives on top of multiple parallel workers, but we do not otherwise invest in coordination research between multiple concurrent streams of work. We defer this to future work, since it is its own interesting problem, plausibly with a different set of useful interventions than the single-worker setting, and one that likely requires an even larger compute budget than what we use here to study properly.

A fourth limitation is possible asymmetry in tuning effort: because Malena is our own system, it is natural to worry that we invested disproportionately more debugging and configuration effort into it than into the external harnesses it is compared against, inflating its apparent advantage. We mitigate this by treating every harness's backbone port the same way we treat our own: each external harness went through its own backbone-specific verification pass before its canonical batch was launched, which surfaced and required fixing genuine incompatibility bugs in the harnesses' own code (e.g., independent tool-calling failures in both MLEvolve and ScienceFlow specific to GLM 5.2), and beyond bug-fixing we additionally ran targeted configuration ablations on ScienceFlow and Arbor (worker count, resource-admission policy, executor timeout, and early-stopping aggressiveness) to check whether their canonical settings were leaving performance on the table. Appendix D.3 reports this work in full; none of these ablations moved outcomes by more than run-to-run noise at our sample sizes, so we kept each harness's original canonical configuration, but we surface the negative results here rather than reporting only the positive tuning story we have for our own system.

G.2 Broader Outlook

Beyond the direct ablation results, our observations across backbones and prior harnesses point at two related hypotheses about why scaffolding is losing its edge, which we lay out here as motivation for future work rather than claims this paper establishes.

Post-training ate the harness. Frontier LLMs are increasingly post-trained directly inside a coding-agent loop – rewarded for planning, executing, and debugging across many turns of real tool use – rather than only on single-turn completions. We hypothesize that this kind of post-training substitutes for exactly the scaffolding that a plain coding-agent environment (Oneshot) would otherwise need on top of a chat completion (Chat): search over what to try next, recovery from a failed step, deciding when a result is good enough to stop on. If so, the Chat-vs-Oneshot gap (Section 4.1) should track how much of this coding-agent post-training a model has received, more than it tracks raw model scale. We see suggestive evidence for this in the DeepSeek V4 family: the Preview releases show close to no gap between Chat and Oneshot performance, while the final releases – built on comparable base models but with substantially more coding-agent post-training – open up a clear gap, though still narrower than the gap we measure for GLM 5.2 (Section 4.1). If this pattern holds more broadly, it reframes a common motivation behind self-evolving-harness designs, where an agent is given the ability to rewrite its own scaffolding to better match its abilities (Du et al., 2026): rather than each model needing a harness custom-fit to it at inference time, post-training may

already be doing this fitting once, in advance, for every downstream task the model is later pointed at.

Coding agents as the bitter lesson for test-time inference. A second, related reading of our results is architectural rather than about training data: for models capable enough at autonomous tool use, adding hand-designed structure on top of a plain coding-agent session – explicit search trees, planner/executor splits, multi-worker coordination – has a poor and often negative return (Section 4.2–4.4), while simply running more (and more capable) coding-agent post-training inside that same plain loop could potentially perform better. This is the same shape of argument Sutton makes about general-purpose search and learning methods outscaling hand-engineered domain structure as compute grows (Sutton, 2019), applied to test-time inference specifically: a basic coding-agent harness plus a stronger backbone tends to close the gap that a specialized harness had opened at a weaker backbone, rather than the specialized harness's advantage compounding. We see the same qualitative pattern in our own results: weaker backbones still benefit measurably from hand-designed workflow priors like MLEvolve's tree search, but as backbone agentic capability increases, Malena's plain single-session harness closes that gap and matches or exceeds MLEvolve without any of its search machinery (Table 15, Appendix D.2). Taken together with the first hypothesis, the practical implication is the same one drawn in Section 6: the harness is not where the durable advantage lives.