AGENTGA: EVOLVING CODE SOLUTIONS IN AGENT-SEED SPACE

Anonymous authors
Paper under double-blind review

ABSTRACT

We present AGENTGA, a framework that evolves autonomous code-generation runs by optimizing the *agent seed*: the task prompt plus optional parent archives that initialize a fresh workspace. The outer loop searches over reusable starting conditions rather than editing code directly. Each generation launches a fresh autonomous run in an isolated workspace, while selected parent archives provide inherited artifacts that descendants can inspect and reuse. AGENTGA couples a population-level genetic algorithm with long-horizon agents, deterministic 1:1 elite tournaments, and online operator allocation through a modified HEDGE controller. We evaluate the approach for tabular AutoML on the 16-competition Weco-Kaggle Lite benchmark, including model-matched AIDE and AutoGluon comparisons. AGENTGA with DeepSeek V4 Flash (0731) achieves 71.13% mean *Exceeds % of Human*, compared with 57.61% for model-matched AIDE and 57.89% for AutoGluon, outranking AIDE on 14/16 tasks. These descriptive results support agent-seed optimization as a practical design choice for autonomous code-search systems, without isolating the causal contribution of inheritance.

1 INTRODUCTION

Many complex problems, from algorithm discovery to data science pipelines, can be framed as search over code: find a program whose execution scores well under some evaluator. Large language models (LLMs) are increasingly used as search engines for this setting; existing methods differ in what level of the system the outer loop optimizes. In an *artifact-centric* regime, the outer loop selects a produced artifact such as a prompt, heuristic, or script. In an *agentic* regime, the model may take multiple tool-using steps, but within a predefined workflow. In an *autonomous* regime, an agent is launched into an isolated workspace and decides for itself which actions to take and when to stop. These regimes describe *what* the outer system selects and reuses, independent of *how* it searches over those units across attempts (e.g., tree search or evolution).

Prior work spans all three regimes (see Section 2 and Appendix C.1), but most of it still optimizes the produced artifact rather than the initial state of a fresh autonomous run. Search-based systems such as FunSearch, ReEvo, AlphaEvolve, and AIDE optimize prompts, heuristics, or code artifacts (Romera-Paredes et al., 2024; Ye et al., 2024; Novikov et al., 2025; Jiang et al., 2025). In AutoML, systems such as DS-Agent, AutoKaggle, and SE-Agent improve task work inside bounded scaffolds (Guo et al., 2024b; Hong et al., 2025; Trirat et al., 2025; Li et al., 2025; Fang et al., 2025; Chi et al., 2024; Liang et al., 2025; Lin et al., 2025). Karpathy's autoresearch (Karpathy, 2026) and Agent K (Grosnit et al., 2024) come closest to open-ended task solving, but each explores only a single lineage. A separate line of work (Automated Design of Agentic Systems, AFlow, AgentSquare, EvoFlow, Darwin Gödel Machine, and Group-Evolving Agents) searches over the agent system itself rather than the seed for a fresh run, and addresses a different problem from ours (Hu et al., 2025; Zhang et al., 2025b; Shang et al., 2025; Zhang et al., 2025a; 2026; Weng et al., 2026).

This leaves a specific gap: to the best of our knowledge, prior work does not combine population-based evolution with autonomous workspace agents while treating the initial state of each fresh run as the optimization object. We close that gap with AGENTGA, which evolves *agent seeds* for fresh autonomous runs and transfers knowledge through algorithmically curated parent archives rather than a single persistent workspace. The genetic algorithm decides what a child starts with; the child agent decides whether and how to use it.

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.

We instantiate AGENTGA for tabular AutoML on Weco-Kaggle Lite, where solution quality is measurable, engineering choices materially affect performance, and long-horizon agents can recover from errors that would end one-shot generation. Across the full 16-competition suite, AGENTGA outranks model-matched AIDE on 14/16 competitions. Across 759 within-method parent-child contests, descendants conditioned on parent archives win 70.7% of their contests, versus 17.8% for de novo proposals.

Contributions. Our key contributions are:

1. **Agent-seed optimization**: We formalize a search space in which the chromosome is the initial state of a fresh autonomous run rather than a code artifact or persistent agent state (Section 3.1).
2. **Evolutionary autonomous architecture**: We design a two-layer system with fresh-workspace agents, inherited parent archives, and deterministic 1:1 elite tournaments (Section 3.2).
3. **LLM-native operators with online allocation**: We replace classical crossover/mutation with six task-based operators and adapt their usage online with a modified HEDGE algorithm (Section 3.3, Section 3.4).
4. **Benchmark evidence**: On the full Weco-Kaggle Lite benchmark, AGENTGA outperforms model-matched AIDE and shows evidence that parent archives improve fresh autonomous runs on average (Section 4).

2 RELATED WORK

Artifact-centric and bounded agentic methods. Many LLM systems for code and AutoML still optimize artifacts or bounded trajectories rather than fresh autonomous runs. AlphaCode, MLCopilot, and CAAFE perform one-shot or locally iterative artifact generation (Li et al., 2022; Zhang et al., 2024; Hollmann et al., 2023). EvoPrompt, ReEvo, FunSearch, AIDE, and AlphaEvolve add orchestration, but the outer loop still stores and selects prompts, heuristics, or scripts (Guo et al., 2024a; Ye et al., 2024; Romera-Paredes et al., 2024; Jiang et al., 2025; Novikov et al., 2025). In AutoML, DS-Agent, Data Interpreter, AutoML-Agent, AutoKaggle, MLZero, SELA, I-MCTS, and SE-Agent equip LLMs with tools, planning, and iterative repair, but they optimize task work inside a fixed scaffold (Guo et al., 2024b; Hong et al., 2025; Trirat et al., 2025; Li et al., 2025; Fang et al., 2025; Chi et al., 2024; Liang et al., 2025; Lin et al., 2025).

Autonomous systems. The closest autonomous systems to AGENTGA are *single-lineage autonomous solvers*, which launch one agent into a workspace and let it run open-endedly without a population layer. Karpathy's *autoresearch* (Karpathy, 2026) commits successive edits to one fixed training script through an off-the-shelf coding CLI and logs every attempt; Agent K autonomously handles setup, debugging, and solution refinement on Kaggle tasks (Grosnit et al., 2024). AGENTGA adds a population layer to this design (Section 3).

Classical AutoML. Classical AutoML methods such as Auto-WEKA, TPOT, Auto-sklearn, and AutoGluon search over predefined pipeline spaces using Bayesian optimization, genetic programming, and ensembling (Thornton et al., 2013; Olson et al., 2016; Feurer et al., 2015; 2022; Erickson et al., 2020). They are strong baselines for structured tabular tasks, but they do not use autonomous code generation. Appendix C.1 summarizes the positioning of representative LLM-based systems.

Agent-design search. A separate line of work optimizes the agent system itself: prompts, tools, modules, workflow graphs, or agent code. Automated Design of Agentic Systems, AFlow, AgentSquare, and EvoFlow search over bounded agentic workflows using meta-agent code generation, MCTS over workflow graphs, modular evolution, and population-based crossover and mutation respectively (Hu et al., 2025; Zhang et al., 2025b; Shang et al., 2025; Zhang et al., 2025a). Darwin Gödel Machine and Group-Evolving Agents extend the same idea to autonomous coding agents, evolving the agent codebase or framework through self-modification across episodes (Zhang et al., 2026; Weng et al., 2026). These methods address a different problem from ours: improving the agent system itself, typically across heterogeneous tasks. AGENTGA instead targets the well-defined evaluator setting, in which a fixed task has a measurable fitness signal, by holding the harness fixed. We do not treat them as direct comparators.

3 METHOD

We present AGENTGA, a framework that searches in *agent-seed space*. Instead of modifying code artifacts directly, the outer loop chooses the initial state for a fresh autonomous run, while the agent decides how to act inside that workspace. The result is a two-layer system: evolution controls inheritance and selection; the agent handles long-horizon problem solving.

3.1 AGENT-SEED SPACE FORMULATION

Each generation begins from a fresh seed rather than an accumulated dialogue history. Knowledge transfer happens through parent archives, so the optimized object is the initial state c_0.

Let $\mathcal{S}$ be the space of candidate solutions and let $h : \mathcal{S} \to \mathbb{R}$ be a direction-normalized fitness function. Artifact-centric systems such as AIDE (Jiang et al., 2025) optimize directly in code space $\mathcal{S}$. We instead optimize over agent seeds $\mathcal{C}$, where each seed $c_0 \in \mathcal{C}$ launches an autonomous trajectory τ that produces a solution $s_\tau \in \mathcal{S}$:

$$c_0^* = \arg\max_{c_0 \in \mathcal{C}} \mathbb{E}_{\tau \sim \text{Agent}(c_0)}[h(s_\tau)] \tag{1}$$

The expectation matters because the same seed can induce different trajectories. A seed contains a task template plus zero or more parent archives:

$$c_0 = \left(\textit{prompt}(\textit{task_type}, \textit{context}),\ \{\textit{parent_archive}_j\}_{j=1}^{n_p}\right) \tag{2}$$

Each child is launched in an isolated workspace containing the execution environment, data, and any inherited artifacts. The seed specifies what is *available* to the agent, not what must be injected into context. Parent material can therefore be inspected selectively, used late, or ignored entirely. Evolution operates on a stochastic seed-to-solution map rather than a deterministic artifact-edit path. This is an empirical inductive bias rather than a guarantee: stronger inherited archives need only improve the descendant distribution on average, not deterministically for every run. Once an agent produces a concrete solution artifact, the downstream workflow evaluates that fixed artifact deterministically with a fixed-seed evaluation harness, so stochasticity lies in generation rather than evaluation.

3.2 TWO-LAYER ARCHITECTURE

AGENTGA separates evolutionary coordination from agent execution (Figure 1). The evolution layer never continues a workspace in place and never specifies the agent's internal reasoning; it only chooses the seed for each new run. The agent layer executes that run in our general-purpose, goal-driven agent harness, which was not built or adapted specifically for this task. The harness is implemented as a LangGraph-based extended ReAct loop (Yao et al., 2023) with six functional nodes: Planning, Agent, Tools, Compression, Verification, and Finalize. Planning maintains a persistent execution plan, and Compression combines windowing with summarize/truncate/drop passes, backed by a durable memory file, so long-horizon trajectories remain within context limits (Appendix A.5). The tool suite covers file inspection, bash execution, and code editing; Appendix A.2 lists the full interface. In the reported configuration, the agent uses DeepSeek V4 Flash (0731).

Within each run, the agent constructs its own context through a sequence of messages, tool calls, and tool results. It determines which files to inspect, which experiments to execute, and which failures to record; the solution emerges through this trajectory as the agent writes, evaluates, and revises code. The harness compresses the conversation as it approaches the context limit, and the run terminates when the agent judges the task complete. Appendix A.4 details this conversation-chain formulation, and Appendix A.5 specifies the compression procedure.

Appendix F illustrates this process through a Bike Sharing lineage and the recorded message sequence and activities of its final agent invocation.

The evolutionary layer maintains an elite pool of size n. At each iteration, every population slot spawns one child from a sampled task operator and selected parent archives. The child then competes only against the elite occupying the same slot from the previous iteration, with the comparison direction determined by the task metric. This deterministic 1:1 elite tournament preserves parallel lineages, avoids eliminating a lineage through unlucky pairing, and gives a direct measurement of whether the child improves on the incumbent elite.

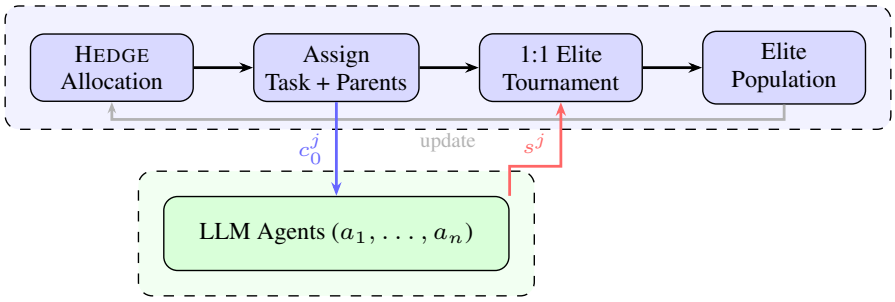


Figure 1: Two-layer architecture. Evolution assigns a task type and parent archives to each fresh autonomous run; the resulting child competes only against its corresponding elite parent.

For the AutoML instantiation, archived solutions are structured as leakage-safe executable workflows with outcome-independent feature engineering, outcome-dependent feature engineering, model definitions, experiment logs, and summary artifacts. A domain-specific workflow layer handles evaluation, artifact management, and submission generation. Child agents receive parent materials through `Previous Experiments/`: a copy of each selected parent's workspace containing its code, recorded attempts together with the code that produced them, notes, and version history, plus a generated summary. Virtual environments, caches, the read-only data mount, and the parent's own `Previous Experiments/` directory are excluded, so inheritance is exactly one generation deep.

3.3 Task-Based Genetic Operators

Traditional crossover and mutation are ill-suited to text and code. We therefore use task-based operators that specify how a fresh child should use inherited context. Each operator launches one autonomous run that may itself execute multiple experiments; in the reported configuration, prompts instantiate a 3-experiment budget per invocation:

1. **Initial** ($n_p = 0$): no parents; start from scratch.
2. **Continue** ($n_p \geq 1$): inspect parent solutions and refine what already works.
3. **Ablation** ($n_p = 1$): add or remove one component at a time to probe individual component contributions.
4. **Merge** (normally $n_p = 2$): combine components from two parents, typically building from one parent while borrowing components or ideas from the other; fall back to one parent if no second elite is available.
5. **EDA** ($n_p = 1$): analyze data first, then propose features or modeling changes guided by that analysis.
6. **Jumpstart** ($n_p = 1$): use inherited artifacts as raw material for a substantially different solution, implemented but inactive in the runs reported here.

The budget is an instruction in the prompt rather than an enforced limit. Operators with parents first re-run the best inherited configuration once, outside the budget, and a child's fitness is the best score among all attempts it recorded. Appendices A and B provide the tool inventory, prompt structure, and compression pipeline of the harness, and task-template excerpts and configuration details of the evolution layer.

3.4 Adaptive Allocation via Modified Hedge

Standard HEDGE (Freund & Schapire, 1997) is designed for settings with many rounds and cleaner reward signals than we have here. Our version updates operator probabilities from direction-aware parent-child improvements:

$$\Delta_j = \begin{cases} s_j - s_{p_j} & \text{if the metric is higher-is-better} \\ s_{p_j} - s_j & \text{otherwise} \end{cases} \tag{3}$$

Positive Δ_j therefore always means "child outperformed parent." Compared with vanilla HEDGE, our variant is tuned for small, noisy gains: it skips updates when fewer than two operators are observed, converts operator means to ranks to suppress metric-scale effects, clips importance weighting at $\kappa = 4.0$, updates log-weights with $\eta = 0.15$, and enforces configured probability floors and ceilings after the softmax. These modifications are designed to stabilize allocation when operator usefulness changes over time. The exact reward and update equations, together with pseudocode and bound-enforcement details, are deferred to Appendix B.

3.5 STOPPING POLICY

The framework uses a configurable budget-and-patience stopping rule. Evolution terminates when either the maximum iteration budget is reached or the best-so-far elite score fails to strictly improve for the configured patience window. In the reported experiments, we set the maximum to 20 iterations and the patience window to 3 iterations.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Benchmark. We evaluate on Weco-Kaggle Lite (Jiang et al., 2025), a 16-competition benchmark of Kaggle tabular machine learning tasks. It was introduced by AIDE and enables direct comparison against human Kaggle leaderboards across regression and classification problems. Its competitions had between 1,174 and 4,978 teams, and Appendix C.2 lists the full benchmark.

Evaluation protocol. Within a run, each scored experiment is assigned the mean holdout score over 5 fixed shuffled train/validation splits, computed by a fixed evaluation harness. We call this the *development score*. It decides every tournament, and Kaggle is never queried during the search. Only after all runs were complete were the test predictions of each run's selected solution submitted to Kaggle. Every reported benchmark number is the score and rank returned by Kaggle on the competition's official leaderboard. We use the recorded Kaggle scores and fixed-leaderboard ranks without rerunning models. We report *Exceeds % of Human* as $100(1-r/N)$, where r is the recorded rank and N is the reporting denominator, the number of teams on the leaderboard, and we average it across the competitions. Because it is rank-based, small absolute score differences can produce large rank changes when many teams are clustered (Appendix G.2).

Baselines. **AIDE (DS)** is AIDE (Jiang et al., 2025) with the same model as AGENTGA and otherwise the default configuration of its repository. It is the closest matched baseline, because both systems target Kaggle-style ML engineering problems, but AIDE searches over code artifacts whereas AGENTGA searches over autonomous agent seeds. **AutoGluon** (Erickson et al., 2020) is a non-LLM AutoML baseline, used in version 1.6.1 with its `best_quality` preset. As a reference, we also report the results published by the AIDE authors for their own GPT-4 Turbo (GPT-4T) runs, **AIDE (GPT-4T)**. We did not run these. **AGENTGA 1×1** is a single-agent control with one population slot and one iteration: a single agent receives the nominal allowance of a full AGENTGA run, 180 attempts ($3 \times 20 \times 3$), and everything else is unchanged (Section 4.4).

Configuration. The AGENTGA and model-matched AIDE arms use DeepSeek V4 Flash (0731), as does AGENTGA 1×1. **AGENTGA** maintains 3 population slots ($n = 3$), runs for at most 20 iterations, and stops early after 3 consecutive iterations without improvement of the best elite (Section 3.5). Each agent that inherits a parent is asked to first reproduce the parent's best configuration, and every agent is asked for 3 new scored experiments (Section 3.3). Appendix B.5 lists the full configuration, and Section 4.6 reports the resources used by AGENTGA and AIDE (DS).

4.2 BENCHMARK RESULTS

Table 1 reports every competition. AGENTGA averages 71.13% *Exceeds % of Human*, against 57.61% for AIDE (DS) and 57.89% for AutoGluon. AGENTGA 1×1 averages 61.20%, and AIDE (GPT-4T) 51.38%.

Table 1: *Exceeds % of Human* on Weco-Kaggle Lite (higher is better; the best of the arms we ran is in bold). AGENTGA, AGENTGA 1×1, and AIDE (DS) use DeepSeek V4 Flash (0731). The AIDE (GPT-4T) column is reported by the AIDE authors, with GPT-4 Turbo, and was not run by us.

Competition	AIDE (GPT-4T)	AIDE (DS)	AutoGluon	AGENTGA 1×1	AGENTGA
Bike Sharing Demand	91.92	92.04	9.07	10.92	**93.34**
Cat in the Dat	46.76	43.40	56.23	54.81	**58.69**
House Prices	72.74	79.99	**96.97**	83.15	96.50
NYC Taxi Fare	44.85	70.37	69.02	81.21	**89.76**
Playground S3E14	52.21	80.98	66.76	76.77	**84.82**
Playground S3E16	51.57	63.59	73.38	**77.92**	76.17
Playground S3E19	36.80	9.45	**55.11**	43.02	43.10
Playground S3E22	25.99	77.38	65.39	72.20	**80.04**
Playground S3E24	65.71	77.12	66.28	72.88	**99.06**
Playground S3E25	41.95	64.91	**78.44**	63.99	63.99
TPS Aug 2022	79.25	65.43	74.22	77.29	**92.17**
TPS Feb 2021	61.02	67.43	67.64	62.48	**82.29**
TPS Feb 2022	43.68	43.28	**49.40**	42.96	39.38
TPS Jan 2022	44.35	2.07	3.45	31.09	**44.35**
TPS Jul 2021	12.98	25.73	29.83	**67.85**	25.97
TMDB Box Office	50.39	58.57	65.02	60.65	**68.46**
Mean	51.38	57.61	57.89	61.20	**71.13**

Against AIDE (DS), the model-matched baseline, AGENTGA ranks higher on 14 of 16 competitions (one-sided sign test, $p = 0.0021$), with a mean difference of +13.52 points (95% bootstrap CI [7.58, 20.04]). The advantage does not depend on any single competition. Omitting any one of them leaves a mean difference between +11.60 and +14.68 points.

Against AutoGluon, AGENTGA ranks higher on 11 competitions and lower on 5, but leads by +13.24 points on average (CI [2.74, 26.05]), because AutoGluon is strong on several competitions and weak on a few.

AIDE (DS) outperforms the published AIDE (GPT-4T) results, against which AGENTGA's record is 14–1–1 (wins–losses–ties), so AIDE (DS) is the stronger baseline. For further context, the AIDE authors also report 41.17% for a human working with ChatGPT, 35.34% for H2O AutoML, and 32.34% for AutoGPT (Table 8).

The reported rankings are determined by Kaggle's official leaderboard rather than the internal validation score. The outer loop must produce executable workflows that can be evaluated across splits, retrained on full data, and submitted to Kaggle under the competition metric, and the gain over AIDE remains after that end-to-end procedure. These are descriptive system comparisons, not controlled causal evidence for inheritance. Appendix D.1 reports every comparison in full, and the following subsections examine inheritance and search behavior within the recorded runs.

4.3 INHERITANCE AND LINEAGE

We analyze within-run tournaments to compare the outcomes of archive-conditioned and de novo proposals. From iteration 2 onward, every child, whatever its operator, must beat the elite of its own population slot under the same model, agent harness, attempt budget, and scoring protocol. **Initial** children start from scratch, which makes them a de novo control for the operators that inherit a parent archive.

Tournament outcomes by inheritance. The 16 runs contain 759 decided tournaments. Children that inherit an archive win 505 of 714 (70.7%), whereas **Initial** children win only 8 of 45 (17.8%, Fisher's exact test, $p < 10^{-10}$), and Table 9 in Appendix D.2 shows the same pattern for every inheriting operator. The difference also appears within the 41 iterations in which the same run launched both kinds of child against their respective slot elites: the win rates are 17.8% and 71.8%, respectively. Win rates also change over the course of the search. As incumbent development scores

improve, the win rate of **Initial** children falls from 37.5% in iterations 2–5 to 0.0% in iterations 16–20, while inheriting children still win 66.0% (Figure 6, Appendix D.2).

An inheriting child records a worse development score than the reference elite in 6.4% of tournaments, compared with 82.2% for children that start from scratch (**Initial**). Ties count as losses, so every win is a strict improvement. Appendix D.2 presents the controls behind these comparisons, and Appendix G.1 discusses selection on the development splits, to which both kinds of child are subject.

How children use what they inherit. The seed only makes the archive available. The agents' tool logs show that every inheriting agent that ran had detected archive accesses, with a median of 25 tool calls (14.5% of their calls), the first of which occurs 1.8% of the way into the run. All 717 had calls accessing the parent's summary and recorded attempts; 91.8% also accessed solution code and 40.7% version history. Access includes reading, listing, copying, and executing inherited material, as defined in Appendix D.2. As the search proceeds, the median inherited archive grows from 105 to 308 text files, yet agents access it with about the same number of calls, 21 at iteration 2 and 26 in iterations 16–20 (Table 11, Appendix D.2).

Exploration and parallel lineages. Within-run experiments include both improvements and regressions. Of 2,357 new scored experiments, 48.9% beat the inherited parent, 48.5% were worse, and the remaining 2.5% matched it exactly. Each agent keeps its best experiment, and the tournament discards the rest. The 1:1 tournament keeps 3 lineages alive instead of truncating to the global best. The lead changed between population slots in 13 of 16 runs. In 8 of the 14 runs with at least 10 iterations, the slot that finished best was not leading at the midpoint, and in 8 of 16 runs, the final best elite descends from two or more slots. Appendix D.3 describes the search dynamics in full.

4.4 Single-Agent Control

We compare AGENTGA with AGENTGA 1×1 to assess the effect of the outer loop under the same nominal attempt allowance. The control uses the same system with one population slot and one iteration, and in all 16 competitions its task prompt differs from that of an iteration-1 AGENTGA agent in **exactly one line**, shown below.

	Population × iterations	The one line that differs
AGENTGA	3×20	`Run 3 attempts. For each one:`
AGENTGA 1×1	$\mathbf{1 \times 1}$	`Run 180 attempts. For each one:`

The model, harness, tools, and evaluation are identical.

Given this allowance in a single conversation, 14 of 16 agents stopped of their own accord before using 90% of it, after a median of 22.5 scored experiments (12.5% of the allowance).

All 14 early-stopping agents explicitly refer to the allowance while planning, 13 note the unspent allowance in their own final audit, and 11 of these attribute it to convergence or noise. On Playground S3E24, for example, the agent wrote, "The task text said 'Run 180 attempts'; the agent ran 29 (budget not consumed). This is a justified early stop."

Of the 14 agents that stopped early, 11 recorded their best development score in the final fifth of their experiments. In 11 of the 14 competitions, AGENTGA went on to reach a better development score than the one at which the single agent stopped.

Both kinds of conversation grow with every call. The single agent's grew from 5,316 to 120,576 tokens over a median of 209 model calls, about 60% of the 200,000-token context limit, and an AGENTGA agent's grows from 5,564 to 70,597 tokens over 126 calls, after which the next generation starts a new one. Each AGENTGA agent is asked for only 3 new experiments, 98.2% deliver at least that many, and relaunching agents from fresh seeds carries a run to a median of 240 scored experiments. Appendix D.5 reports every competition and the conversation lengths in detail.

On the leaderboard, AGENTGA ranks higher than AGENTGA 1×1 on 12 competitions, lower on 3, and equal on 1 (one-sided sign test, $p = 0.018$). The mean difference is +9.93 percentage points, although its 95% bootstrap confidence interval includes zero $[-0.97, 22.36]$ (Appendix D.1). Two of

the three losses, including the largest, are on TPS Jul 2021 and TPS Feb 2022, the two competitions whose verifier does not reflect the test set (Section 4.5). The comparison is matched in allowance, not in compute. AGENTGA ran 4.7× as many scored experiments at 20.7× the LLM cost.

4.5 VERIFIER–LEADERBOARD ALIGNMENT

AGENTGA uses the development score over 5 fixed random splits as its verifier, guiding selection without querying Kaggle during search. We examine whether improvements on this verifier translate into better performance on the unseen test set, as measured by the Kaggle leaderboard. We traced the reported member of four 20-iteration runs, the two highest- and the two lowest-ranked, back through its ancestors to the iteration-1 **Initial** member, and submitted every generation of each line to Kaggle without rerunning any model. Including additional elite checkpoints, we scored 77 distinct files (Appendix E). Figure 8 in Appendix E shows paired development-score and leaderboard trajectories over successive members of each selected lineage. Along every line development performance improved, by +0.26% to +10.03%, because an elite is replaced only by a strict improvement.

In the two highest-ranked traced runs, development-score improvements were accompanied by leaderboard improvements. Playground S3E24 rose from *Exceeds* 70.8 to 99.1 over 17 generations, from rank 557 to rank 18, and Bike Sharing Demand rose from 83.2 to 93.3. Across these two lines, 20 of 31 generation steps moved up the leaderboard and 6 moved down.

In the two lowest-ranked traced runs, we observed data properties that may contribute to the mismatch between development and leaderboard performance. TPS Jul 2021 has a temporal train–test split: its test period begins where its training period ends (2011-01-01) and extends to 2011-04-04, whereas the development verifier uses randomly held-out training rows. In TPS Feb 2022, 51% of a random holdout has an exact duplicate among the training rows. These observations suggest possible limitations of random-split evaluation on these tasks, but do not establish the cause of the observed mismatch.

Along the TPS Jul 2021 line development performance improved by +10.0% while *Exceeds* went from 31.5 to 26.0, and the TPS Feb 2022 line stayed nearly flat, from 37.7 to 39.4. These are also the two competitions on which AGENTGA trails the single agent (Section 4.4).

4.6 COMPUTATIONAL COST

The 16 AGENTGA runs comprise 272 iterations, 816 agent invocations, and 3,286 scored experiments (16,383 model fits), and consumed 5.39B tokens over 109,437 LLM calls. Because 71.6% of prompt tokens were served from the cache, the total cost was about $155, or $9.69 per competition. Model training and scoring dominate wall-clock time, and waiting for the LLM accounts for only 14.1% of agent time. AIDE (DS) ran at its default budget, which uses far less compute, so the comparison in Section 4.2 matches the model rather than the compute (Appendix G.1). Section 4.4 shows that, within our harness, giving a single agent a larger allowance does not by itself close the gap to AGENTGA. Appendix C.3 gives per-competition accounting.

5 DISCUSSION

Agent-Seed Space. The agent-seed formulation mirrors how a human data scientist might be briefed. A new task can begin either from scratch or from prior working solutions; the seed is exactly that reusable starting point, not a deterministic recipe for the final answer. This is useful when prior context matters but carrying forward one growing workspace is undesirable. By resetting each run to a compact initial condition while preserving parent archives in the workspace, AGENTGA separates what is inherited from how the next agent chooses to use it. Mechanically, it replaces one ever-growing conversation with a sequence of short conversations that pass their state on through files. The single-agent control provides a practical reason for this reset (Section 4.4). Given the whole allowance in one conversation, most agents declared the task complete after using a small fraction of it, whereas short, bounded tasks in fresh conversations, with the archive carrying state between them, sustained the search. This is an observation about one model and one harness, not a general law. We conjecture that this reflects how current models are trained: on long-horizon

tasks, but rarely on horizons of hundreds of experiments, so that an agent concludes too early that it is done. We have not tested this conjecture, and the behavior may change with future models.

Self-Directed Context Construction. Unlike retrieval-style systems, AGENTGA does not inject pre-selected snippets into prompt context. Parent archives remain available in the workspace for the agent to inspect, ignore, or reinterpret while pursuing the task. In our runs, every inheriting agent that executed had detected archive accesses, including the parent's summary and recorded attempts, while code access was selective (Section 4.3). The effective working context is therefore constructed by the trajectory itself: file reads, experiments, failures, and replanning decisions. In that sense, the search object is the reusable starting condition, while the concrete solution remains a downstream product of the run.

Role of the Verifier. The verifier determines which solutions survive and what subsequent agents inherit. For this search to improve performance on unseen data, the development score must be a useful proxy for the target test distribution: otherwise, the system can steadily improve its measured fitness without improving generalization. Averaging over multiple splits reduces reliance on any one split, but cannot correct a systematic mismatch between the development protocol and the test set. Section 4.5 illustrates this distinction: development performance improved along all four traced lineages, whereas leaderboard performance improved only in the two highest-ranked runs. The temporal structure and duplicate overlap observed in the other two runs suggest that choosing a development protocol that reflects the intended evaluation setting is essential to making evolutionary progress meaningful beyond the verifier itself.

Limitations. *Compute cost.* Long-horizon agents consume far more tokens and wall-clock time than artifact-centric generation (Section 4.6), and multi-split evaluation adds further CPU and wall-clock evaluation cost rather than additional LLM token cost. Our comparison with AIDE matches the model, not the compute.

The verifier. The alignment analysis in Section 4.5 motivates rerunning the affected competitions with time-ordered or duplicate-aware splits. These alternatives would better reflect the temporal structure or reduce duplicate overlap, respectively, but we have not tested whether they would improve leaderboard performance.

Workflow and capability. The method requires a bounded, automatable fitness signal and a domain-specific workflow layer that makes candidate solutions comparable, and its performance is limited by the base model, that layer, and the search space exposed to the agent. Slow evaluations and increasingly complex descendants reduce the number of experiments a fixed budget allows, and a new domain requires engineering how artifacts are structured, inherited, and evaluated.

Component isolation. The present experiments evaluate AGENTGA as an end-to-end system rather than a full factorial ablation of every design choice. In particular, we do not separately isolate the marginal contribution of HEDGE allocation, population size, archive inheritance, task-template design, and the domain-specific workflow layer under matched compute. The single-agent control removes the outer loop under a matched attempt allowance (Section 4.4), and the parent-child tournament analysis provides direct evidence that archive-conditioned descendants are more competitive than de novo **Initial** proposals, but all results come from one base model, one agent harness, and one benchmark domain, and a complete controlled ablation remains future work.

6 CONCLUSION

We introduced AGENTGA, a framework that searches in agent-seed space by launching fresh autonomous runs with inherited parent archives. Deterministic 1:1 elite tournaments and adaptive HEDGE allocation coordinate population-level search. On Weco-Kaggle Lite, the DeepSeek V4 Flash (0731) implementation outranks model-matched AIDE on 14/16 tasks, with a mean Exceeds advantage of 13.52 percentage points, and exceeds a single long-lived agent given the same attempt allowance by a mean of 9.93 points. These system-level results motivate further study of agent-seed optimization as a way to convert budget into sustained search; controlled attribution to inheritance, compute-matched baselines, ablations of adaptive allocation and population size, and domains beyond tabular AutoML remain open.

AI USE STATEMENT

Large language models are the object of study in this work: the agents that AGENTGA evolves are LLM agents (Section 3), the models are named in Section 4, and every solution an agent writes is scored by a fixed evaluation harness that involves no LLM. The rest of this statement is about the authors' own use of generative AI tools.

In this work, we used generative AI tools for research execution: to implement the analysis of the archived runs (scripts that extract, deduplicate and reformat the run records and compute the reported statistics and tables from them), and to support the analysis and interpretation of those records, including auditing the numbers in the manuscript against them. We also used generative AI tools as a sounding board, to suggest and discuss alternatives for the analyses, the experimental choices, and the presentation, which the authors then accepted, adapted, or rejected. We have not used generative AI tools to develop the conceptual framework or to propose the research hypotheses. The central idea, evolving the starting conditions of long-horizon autonomous agent runs with a genetic algorithm, as well as the method and the final experimental design, are the authors' own. Generating synthetic data sets, formulating mathematical claims, writing proofs and translation are not applicable to this work. Additionally, we used generative AI tools (LLM coding and writing assistants, including Claude, Anthropic) to create and edit software code and figures, to draft and edit parts of the text for readability, and to prepare the LaTeX formatting.

We have reviewed all AI-assisted work. AI-assisted code and text were checked by two of the authors. Every reported number is produced by a script that reads the archived run records rather than typed by hand, and the tournament tallies recomputed by those scripts were checked against the summaries that the search itself wrote at run time. We take responsibility for the final content of this work, including text, claims and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

AGENTGA could broaden access to complex optimization tasks by automating parts of the search for executable solutions. Potential harms include misuse for harmful code generation, displacement of some specialized engineering work, and increased environmental cost from long autonomous trajectories. Practical deployment should therefore pair the system with domain restrictions, monitoring, and compute controls.

Because AGENTGA searches over long-horizon autonomous runs, deployment should use sandboxed tools, audit logs, explicit budget caps, and task-specific rules that prevent unsafe multi-step behavior and wasteful leaderboard chasing.

REPRODUCIBILITY STATEMENT

Section 4 summarizes the configuration used in this manuscript, and Appendix B.5 lists the full experiment settings. These values correspond to the experiment environment configuration used for the reported runs rather than fallback library defaults. LLM generation remains stochastic, but evaluation of a fixed archived solution is deterministic under the fixed seeds of the workflow harness. The experimental records used for this manuscript include all reported benchmark runs together with all their archived iterations, agent chains, and code artifacts. Upon acceptance, we will release the code for AGENTGA: the genetic-algorithm evolution layer and the tabular AutoML instantiation used in this work.

We distinguish reproducibility of *search* from reproducibility of *evaluation*: the full search is stochastic, but the system preserves every iteration, run, and agent chain together with all archives and code, providing a complete audit trail for the reported tournaments and plots.

REFERENCES

Yizhou Chi, Yizhang Lin, Sirui Hong, Duyi Pan, Yaying Fei, Guanghao Mei, Bangbang Liu, Tianqi Pang, Jacky Kwok, Ceyao Zhang, Bang Liu, and Chenglin Wu. SELA: Tree-Search Enhanced LLM Agents for Automated Machine Learning, October 2024. URL http://arxiv.org/abs/2410.17238. arXiv:2410.17238 [cs].

Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data, March 2020. URL http://arxiv.org/abs/2003.06505. arXiv:2003.06505 [stat].

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. In *Advances in Neural Information Processing Systems 38 (NeurIPS)*, 2025. doi: 10.48550/arXiv.2505.13941. URL http://arxiv.org/abs/2505.13941.

Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and Robust Automated Machine Learning. In *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. URL https://papers.nips.cc/paper/2015/hash/11d0e6287202fced83f79975ec59a3a6-Abstract.html.

Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. Auto-Sklearn 2.0: Hands-free AutoML via Meta-Learning. *Journal of Machine Learning Research*, 23 (261):1–61, 2022. ISSN 1533-7928. URL http://jmlr.org/papers/v23/21-0992.html.

Yoav Freund and Robert E Schapire. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *Journal of Computer and System Sciences*, 55(1):119–139, August 1997. ISSN 0022-0000. doi: 10.1006/jcss.1997.1504. URL https://www.sciencedirect.com/science/article/pii/S002200009791504X.

Antoine Grosnit, Alexandre Maraval, Refinath S. N, Zichao Zhao, James Doran, Giuseppe Paolo, Albert Thomas, Jonas Gonzalez, Abhineet Kumar, Khyati Khandelwal, Abdelhakim Benechehab, Hamza Cherkaoui, Youssef Attia El-Hili, Kun Shao, Jianye Hao, Jun Yao, Balázs Kégl, Haitham Bou-Ammar, and Jun Wang. Kolb-Based Experiential Learning for Generalist Agents with Human-Level Kaggle Data Science Performance, November 2024. URL http://arxiv.org/abs/2411.03562. arXiv:2411.03562 [cs].

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. EvoPrompt: Connecting LLMs with Evolutionary Algorithms Yields Powerful Prompt Optimizers. In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024a. doi: 10.48550/arXiv.2309.08532. URL http://arxiv.org/abs/2309.08532.

Siyuan Guo, Cheng Deng, Ying Wen, Hechang Chen, Yi Chang, and Jun Wang. DS-Agent: Automated Data Science by Empowering Large Language Models with Case-Based Reasoning. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, pp. 16813–16848, 2024b. doi: 10.48550/arXiv.2402.17453. URL http://arxiv.org/abs/2402.17453.

Noah Hollmann, Samuel Müller, and Frank Hutter. Large Language Models for Automated Data Science: Introducing CAAFE for Context-Aware Automated Feature Engineering. In *Advances in Neural Information Processing Systems 36 (NeurIPS)*, 2023. doi: 10.48550/arXiv.2305.03403. URL http://arxiv.org/abs/2305.03403.

Sirui Hong, Yizhang Lin, Bang Liu, Bangbang Liu, Binhao Wu, Ceyao Zhang, Chenxing Wei, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, Li Zhang, Lingyao Zhang, Min Yang, Mingchen Zhuge, Taicheng Guo, Tuo Zhou, Wei Tao, Xiangru Tang, Xiangtao Lu, Xiawu Zheng, Xinbing Liang, Yaying Fei, Yuheng Cheng, Zhibin Gou, Zongze Xu, and Chenglin Wu. Data Interpreter: An LLM Agent For Data Science. In *Findings of the Association for Computational Linguistics: ACL 2025*, 2025. doi: 10.48550/arXiv.2402.18679. URL http://arxiv.org/abs/2402.18679.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems, 2025. URL https://arxiv.org/abs/2408.08435.

Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. AIDE: AI-Driven Exploration in the Space of Code, February 2025. URL http://arxiv.org/abs/2502.13138. arXiv:2502.13138 [cs].

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

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, December 2022. doi: 10.1126/science.abq1158. URL https://www.science.org/doi/10.1126/science.abq1158.

Ziming Li, Qianbo Zang, David Ma, Jiawei Guo, Tuney Zheng, Minghao Liu, Xinyao Niu, Yue Wang, Jian Yang, Jiaheng Liu, Wanjun Zhong, Wangchunshu Zhou, Wenhao Huang, and Ge Zhang. AutoKaggle: A Multi-Agent Framework for Autonomous Data Science Competitions. In *Proceedings of the 13th International Conference on Learning Representations (ICLR)*, 2025. doi: 10.48550/arXiv.2410.20424. URL http://arxiv.org/abs/2410.20424.

Zujie Liang, Feng Wei, Wujiang Xu, Lin Chen, Yuxi Qian, and Xinhui Wu. I-MCTS: Enhancing Agentic AutoML via Introspective Monte Carlo Tree Search, February 2025. URL http://arxiv.org/abs/2502.14693. arXiv:2502.14693 [cs].

Jiaye Lin, Yifu Guo, Yuzhen Han, Sen Hu, Ziyi Ni, Licheng Wang, Mingguang Chen, Hongzhang Liu, Ronghao Chen, Yangfan He, Daxin Jiang, Binxing Jiao, Chen Hu, and Huacan Wang. Se-agent: Self-evolution trajectory optimization in multi-step reasoning with llm-based agents, 2025. URL https://arxiv.org/abs/2508.02085.

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

Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pp. 485–492, New York, NY, USA, July 2016. Association for Computing Machinery. ISBN 978-1-4503-4206-3. doi: 10.1145/2908812.2908918. URL https://dl.acm.org/doi/10.1145/2908812.2908918.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, January 2024. ISSN 1476-4687. doi: 10.1038/s41586-023-06924-6. URL https://www.nature.com/articles/s41586-023-06924-6.

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. Agentsquare: Automatic llm agent search in modular design space, 2025. URL https://arxiv.org/abs/2410.06153.

Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '13, pp. 847–855, New York, NY, USA, August 2013. Association for Computing Machinery. ISBN 978-1-4503-2174-7. doi: 10.1145/2487575.2487629. URL https://dl.acm.org/doi/10.1145/2487575.2487629.

Patara Trirat, Wonyong Jeong, and Sung Ju Hwang. AutoML-Agent: A Multi-Agent LLM Framework for Full-Pipeline AutoML. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025. doi: 10.48550/arXiv.2410.02958. URL http://arxiv.org/abs/2410.02958.

Zhaotian Weng, Antonis Antoniades, Deepak Nathani, Zhen Zhang, Xiao Pu, and Xin Eric Wang. Group-evolving agents: Open-ended self-improvement via experience sharing, 2026. URL https://arxiv.org/abs/2602.04837.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *Proceedings of the 11th International Conference on Learning Representations (ICLR)*, 2023. URL http://arxiv.org/abs/2210.03629.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. ReEvo: Large Language Models as Hyper-Heuristics with Reflective Evolution. In *Advances in Neural Information Processing Systems 37 (NeurIPS)*, 2024. doi: 10.48550/arXiv.2402.01145. URL http://arxiv.org/abs/2402.01145.

Guibin Zhang, Kaijie Chen, Guancheng Wan, Heng Chang, Hong Cheng, Kun Wang, Shuyue Hu, and Lei Bai. Evoflow: Evolving diverse agentic workflows on the fly, 2025a. URL https://arxiv.org/abs/2502.07373.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin godel machine: Open-ended evolution of self-improving agents, 2026. URL https://arxiv.org/abs/2505.22954.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. Aflow: Automating agentic workflow generation, 2025b. URL https://arxiv.org/abs/2410.10762.

Lei Zhang, Yuge Zhang, Kan Ren, Dongsheng Li, and Yuqing Yang. MLCopilot: Unleashing the Power of Large Language Models in Solving Machine Learning Tasks. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, pp. 2931–2959, 2024. doi: 10.48550/arXiv.2304.14979. URL http://arxiv.org/abs/2304.14979.

APPENDIX

A THE AGENT HARNESS

This appendix summarizes the general-purpose agent harness used to execute each autonomous run, its available tools, and its interface to externally supplied task instructions. It then describes how the agent's conversation develops within a run and how information is transferred between runs. The harness provides the execution infrastructure; Appendix B describes the evolutionary orchestration that constructs agent seeds and selects the resulting solutions.

A.1 STRUCTURE

The agent layer uses our general-purpose, goal-driven agent harness, which was not built or adapted specifically for tabular AutoML. Domain-specific instructions and evaluation commands are supplied through the workspace.

The harness is implemented as a LangGraph-based extended ReAct loop with planning, tool execution, durable memory, context compression, and completion checks. A persistent plan and memory record support execution across model turns, while checkpointing allows interrupted runs to resume. The reported configuration uses DeepSeek V4 Flash (0731) for all LLM roles.

The agent requests termination when it judges the task complete. Plan-completion checks and a final validation audit can return control to the agent before finalization. These mechanisms govern task execution rather than fitness: after the run, the evolution layer reads the recorded score from the workspace and applies the tournament and retry rules of Appendix B.2.

A.2 TOOL SUITE

Table 2 lists the general-purpose tools available in the reported configuration. They support workspace inspection, file editing, command execution, visual inspection, and management of the agent's plan and memory. Domain-specific training and evaluation are accessed through the workspace's command-line interface rather than specialized harness tools; for example, a scored attempt is launched with `uv run python run.py main-training`. Web search, sub-agent dispatch, skills, and plugins are disabled in the reported runs.

Table 2: Tools available to AGENTGA agents in the reported configuration.

Category	**Tools**
File and shell	`read_file`, `write_file`, `edit_file_search_replace`, `run_shell`
Vision	`view_image`, `file_to_image`
Plan	`check_off_plan_items`, `expand_plan`, `modify_plan`, `revise_task_and_plan` (for user redirects; not applicable to unattended runs)
Memory	`update_memory`
Control	`exit_session`

A.3 TASK INPUT INTERFACE

Each run receives a task brief and a prepared workspace, following the familiar goal-directed interaction pattern of coding assistants such as Codex and Claude Code. The brief specifies what the agent should accomplish, while accompanying workspace instructions describe the available artifacts, permitted changes, and evaluation commands. The agent determines the sequence of analyses, code edits, and tool calls used to carry out the task.

In AGENTGA, the evolution layer constructs these starting conditions programmatically: it selects the task operator, supplies the brief and shared execution instructions, and makes the selected parent archives available in the workspace (Appendix B.3). The domain package supplies the problem-specific instructions and evaluation interface (Appendix B). These externally supplied inputs are

distinct from the harness's internal prompts for execution, planning, and memory. Agent-seed search therefore changes the starting conditions of each run rather than the internal design of the agent.

A.4 The Conversation Chain

Within a run, the agent builds its own context. Like any chat-based agent, it works on a chain of atomic chunks, one per model turn, and a turn can contain a message, tool calls with their results, or both. Let S be the system prompt and A the first message. The first call sees $S + A$, the second sees these together with the first turn and its tool results, $S + A + B$, the third sees $S + A + B + C$, and so on, because every call re-reads the chain so far, so early turns shape all later ones (Figure 4). The chain therefore only grows, until it approaches the context limit. Schematically, the harness compresses the accumulated context $S + A + B + C + D$ into S', which represents the retained instructions and compressed conversational state. The system prompt, task, and recent turns are preserved as described in Appendix A.5. The context shrinks, and the chain grows again from there, to $S' + E + F + G$ (Figure 3).

What enters the chain is decided by the agent itself, namely which files it reads, which data it inspects, which experiments it runs, and which failures it records. The solution is not generated in a single step from a prompt but emerges from this chain. At some point a turn, say G, issues a tool call that writes or edits the solution files, the next turn H runs the scoring command, and that code becomes the running solution, which later turns of the same chain continue to revise (Figure 2). The chain ends when the agent itself judges that its task is complete and exits the run. AGENTGA never edits the solution a chain produces. It chooses only how each new chain begins, which is the agent seed of Section 3.1. What earlier chains learned reaches the child only as files under `Previous Experiments/`, which it opens when it chooses (Section 4.3).

This distinguishes AGENTGA from two alternatives. A single long-lived agent, as in autoresearch (Karpathy, 2026), carries one chain, compressed whenever it approaches the limit, through the entire search. In our single-agent control, the search therefore ends whenever that one agent decides it is done (Section 4.4). Artifact-centric systems such as AIDE keep no such persistent chain. Each solution is produced from a prompt assembled by the scaffold, and the search operates on the resulting artifacts.

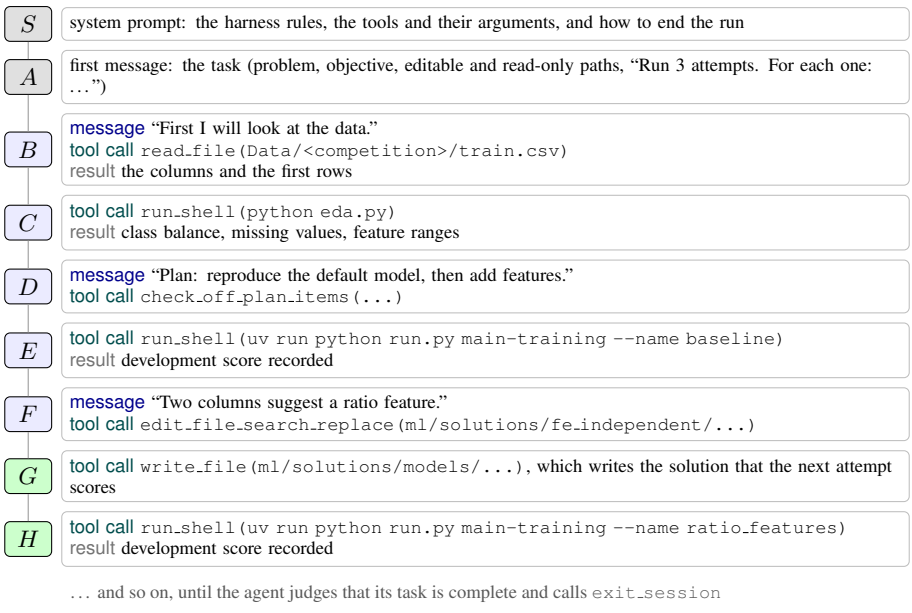


Figure 2: What the chunks of one chain contain, illustrated (not taken from a specific run). Each box is one model turn, with its message, its tool calls and their results; every call re-reads all boxes above it. Figure 3 shows how such a chain grows and is compressed.

A.5 Context Compression

Figure 3 illustrates the compression process described in Appendix A.4: the chain grows until it approaches the context limit, is rewritten into a compact state S', and then grows again. The remainder of this subsection specifies the compression rules.

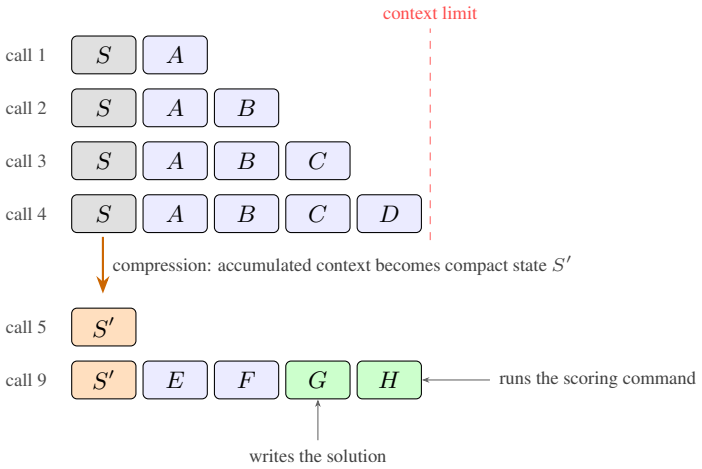


Figure 3: How the chain grows and is compressed (schematic). Near the context limit the chain is rewritten into a compact state S' and then grows again. In the harness itself, the system prompt, the task, and the most recent turns are kept verbatim.

Token counting uses `tiktoken` with `cl100k_base` encoding. The unit of context management is a *chunk*: one model turn together with the tool results that answer it, so that a tool call is never separated from its result. Under the harness defaults, compression is triggered when the prompt context exceeds 200,000 tokens or after every 100 agent steps, and it targets a context of 40,000 tokens. The first chunk (the task) and the last three chunks are protected. The remaining chunks are escalated, oldest first, through the following stages until the target is met:

1. **Compress.** An LLM summarizes message content and reasoning to about 10% of their length. Tool-call arguments and tool results are instead truncated to 200 tokens per value.
2. **Truncate.** Every part of the chunk is clipped to a leading and a trailing span of tokens.
3. **Drop.** The chunk is removed from the working context.

A chunk moves to a new stage only if that stage shortens it. Before a new block of chunks is dropped, the memory subgraph rewrites `MEMORY.md` from the span that is about to leave the context, so that information in the dropped chunks survives in condensed form. The agent can also request this rewrite through the `update_memory` tool. Compression calls run sequentially, and the full transcript is retained on disk.

A.6 Path Dependence Within a Chain

Because every call receives the retained working context, early turns shape all later ones. Figure 4 illustrates the consequence: two chains that start from the same $S + A$ and differ only slightly in their first turns can end up searching in quite different directions, and a single chain tends to keep refining the direction it started in.

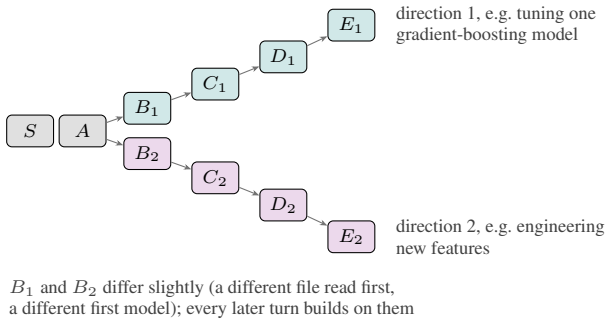


Figure 4: Why a chain drifts (schematic, not measured). A small difference in the first turns is carried into every later turn.

B EVOLUTIONARY ORCHESTRATION

This appendix describes the evolution layer: how workspaces are passed between generations, the outer loop and its tournament, the operators, the HEDGE allocation, and the configuration of the reported runs. Unless otherwise stated, concrete values refer to the experiment environment configuration reported in the manuscript rather than fallback library defaults.

The implementation intentionally separates stochastic generation from a fixed evaluation protocol. LLM calls are not seeded, so the same agent seed can produce divergent autonomous trajectories and solution artifacts. The evolution layer's random generator is not seeded either, so a run cannot be replayed from a random seed. Inherited archives are exposed as optional workspace resources rather than forced prompt context, so two runs from the same seed may draw different amounts from the workspace before diverging. The evolution layer never re-executes a solution. A member's fitness is the score that the fixed evaluation harness recorded when the agent ran the solution, namely the mean over 5 fixed seeds (42–46), each with its own 20% holdout. The evaluation code is mounted read-only, and the protocol files are hashed before and after the run. A member whose protocol files changed is archived but cannot win a tournament.

Domain interface and evaluation records. The domain package supplies the workspace's `AGENTS.md`, specifying the evaluation command, editable components, archive-adoption procedure, and leakage constraints. The evaluation package and data are mounted read-only, except for the editable `ml/solutions/` directories. The agent may reorganize its solution within these directories; the evaluation interface remains fixed. Outcome-independent feature engineering is fitted across both frames of a split, whereas outcome-dependent feature engineering is fitted only on non-holdout rows. Each scored attempt records per-seed results, predictions, and a frozen solution. The evolution layer reads the score from `attempts.json`, which is writable and is not itself hashed; protocol-file hashes determine tournament eligibility. The prompts require scoring through the designated command, and the recorded artifacts permit subsequent auditing without re-executing the solution.

B.1 WORKSPACE INHERITANCE ACROSS GENERATIONS

AGENTGA addresses the drift of Appendix A.6 by requesting a small number of experiments per invocation and starting several fresh chains per generation (Figure 5). Each chain works in a workspace. In generation 1, every chain starts from scratch, from the same template workspace $W_{0,0}$, and the chain of population slot j edits it into $W_{1,j}$. In subsequent generations, each child starts in a fresh workspace containing the parent archives selected by its operator, and its chain produces $W_{g,j}$. A de novo `Initial` child receives no archives. The tournament keeps the best workspace per population slot, making it available for inheritance in the next generation. Each child's chain starts again from $S + A$ and reads any inherited archives as files when it chooses. The direction of the search is therefore re-drawn at every generation, while what earlier chains built is carried forward.

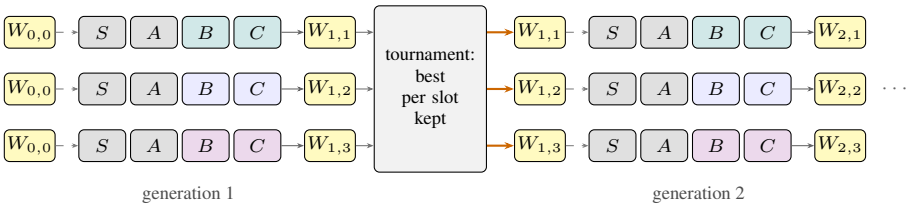


Figure 5: How workspaces are inherited across generations (single-parent schematic). $W_{g,j}$ is the workspace produced by population slot j in generation g. Generation 1 starts from the template workspace ($W_{0,0}$). Later children receive the selected parent's kept workspace as an archive in a fresh workspace; dashed arrows indicate access to the template or inherited archive, not continuation of the parent's conversation.

B.2 Algorithm Summary

Algorithm 1 summarizes the outer-loop procedure used in the reported experiments.

Algorithm 1 AGENTGA outer loop

Input: dataset $\mathcal{D}$, population size n, hard cap T, patience window w
Initialize elite pool $E \leftarrow \emptyset$ and bounded HEDGE sampler from configured base probabilities
best_so_far $\leftarrow \varnothing$
stagnation_count $\leftarrow 0$
for $t = 1$ **to** T **do**
  $E^- \leftarrow E$ {previous elite snapshot}
  for $j = 1$ **to** n **in parallel do**
    if $t = 1$ **then**
      $k_j \leftarrow$ `Initial`, $p_j \leftarrow \varnothing$, $P_j \leftarrow \emptyset$
    else
      $k_j \leftarrow$ HEDGESAMPLE()
      $(p_j, P_j) \leftarrow$ SELECTPARENTS(k_j, E^-, j)
    end if
    $c_0^j \leftarrow (\text{task}_{k_j}, \text{archives}[P_j])$
    $(\pi_j, s_j) \leftarrow$ AGENT$(c_0^j, \mathcal{D})$
  end for
  for $j = 1$ **to** n **do**
    if not ELIGIBLE(j) **then**
      Carry forward the reference elite if present; otherwise leave slot j empty
    else if $t = 1$ **or** BETTER$(s_j, E^-[p_j].\text{score})$ **then**
      $E[j] \leftarrow (\pi_j, s_j)$
    else
      $E[j] \leftarrow E^-[p_j]$
    end if
  end for
  if $t > 1$ **then**
    Compute direction-aware improvements Δ_j for scored children against elite parents in E^-
    Update bounded HEDGE weights from observed (k_j, Δ_j) pairs
  end if
  $b_t \leftarrow$ BEST(E).score
  if $t = 1$ **or** BETTER$(b_t, \textit{best_so_far})$ **then**
    best_so_far $\leftarrow b_t$
    stagnation_count $\leftarrow 0$
  else
    stagnation_count $\leftarrow$ *stagnation_count* $+ 1$
  end if
  if *stagnation_count* $\geq w$ **then**
    break
  end if
end for
Output: BEST(E)

The pseudocode summarizes selection for iterations with an available elite; retry handling is described below. If no elite is available for the next iteration, the implementation launches only `Initial` members, and an eligible child can fill an empty slot without a comparison. ELIGIBLE requires a recorded score and an intact evaluation protocol. Here BETTER and BEST apply the configured metric direction, *best_so_far* stores the best elite score seen so far, and $n = 3$, $T = 20$, and $w = 3$ in the reported configuration. BETTER is a strict inequality with no tolerance, so a tie retains the incumbent.

Fitness, eligibility, and retries. The score s_j is the best value among all attempts that member j recorded in its own workspace, including its reproduction of an inherited configuration. The attempts of its parents are not counted. A child is eligible for the tournament only if it recorded a score under an unmodified evaluation protocol. Otherwise, the reference elite is carried forward if present; an ineligible first-generation member leaves its slot empty. A member that records no score is relaunched as a new member with the same operator and parents, up to 3 times per slot and

iteration. If the slot still has no score after these retries, the run terminates once the bookkeeping for that iteration is complete. An interrupted run resumes its members from their harness checkpoints. No separate pass re-validates a winner.

The SELECTPARENTS subroutine called in Algorithm 1 is operationalized in Algorithm 2. The previous-iteration elite snapshot E^- used as input is the elite pool taken at the start of iteration t, before any of this iteration's children have been evaluated, matching the assignment $E^- \leftarrow E$ at the top of the loop in Algorithm 1. Thus $E^-[j]$ denotes slot j's incumbent elite carried in from iteration $t-1$.

SELECTPARENTS returns a pair (p, P) that play distinct roles in the outer loop. The first component p is the slot's *tournament reference*: the incumbent elite that the new child must beat for promotion (the tournament step of Algorithm 1). For every operator, including `Initial`, $p = j$ is the slot's own elite in E^-, so a fresh `Initial` child sampled by HEDGE in iteration $t > 1$ is still tournament-compared against that elite. The second component P is the *archive set copied into the child's workspace* as inheritable context; P varies by operator and is summarized in Algorithm 2. If slot j holds no elite because its iteration-1 member was not eligible, a random elite from another slot serves as p. If the child loses, that elite is copied into slot j.

Algorithm 2 SELECTPARENTS subroutine called in Algorithm 1

Input: operator k, previous elite snapshot E^-, slot index j
$p \leftarrow j$ if slot j holds an elite, otherwise a uniform random slot that does {tournament reference}
if $k =$ `Initial` **then**
  return $(p, \emptyset)$ {no archives in workspace; tournament still runs}
else if $k \in \{$`Ablation`, `EDA`, `Jumpstart`$\}$ **then**
  return $(p, \{p\})$
else if $k =$ `Continue` **then**
  $A \leftarrow$ sample $n_p - 1$ additional parents without replacement from $E^- \setminus \{p\}$
  return $(p, \{p\} \cup A)$ {$n_p = 1$ in the reported configuration}
else if $k =$ `Merge` **then**
  if $E^- \setminus \{p\} = \emptyset$ **then**
    return $(p, \{p\})$ {no other elite exists}
  end if
  $q \leftarrow$ uniform random from $E^- \setminus \{p\}$
  return $(p, \{p, q\})$
end if

B.3 TASK TYPES AND OPERATOR BEHAVIOR

The code and its configuration keys refer to operators as *tasks*, and the two terms are used interchangeably below.

Task and execution instructions. The evolution layer constructs the task brief that forms the prompt component of each agent seed. It specifies the problem and objective, the selected operator, inherited archive locations, the requested experiment allowance, and the expected experimental records and outputs. The evolution layer also supplies shared execution instructions for using the designated scoring command, recording attempts, and committing experimental changes. These instructions are inputs to the general-purpose harness, not part of its internal prompting. Domain-specific commands and workspace constraints are supplied by the domain package (Appendix B).

Each member receives a single task prompt, which is composed for its operator from one fragment file (`agentga_ga/tasks/brief.md`). All operators share the header (problem, objective, direction, and "Nothing else is measured."), the list of inherited paths, the setup command, the editable and read-only paths, the reproduction rule, and the attempt loop (change, run the scoring command, and commit). Only the stance, any work before the loop, the first loop step, and the requested content of the closing `RESULTS.md` differ between operators. An operator specifies what a member inherits, not which modeling method it should use.

Inheritance. The number of parents (n_p) varies by operator. Each child receives a complete copy of the elite workspace of every selected parent under `Previous Experiments/Iteration NNNN Population NNNN/`. The copy contains the parent's code, every attempt together with the frozen code that produced it, its notes, and its git history (as `parent.git`), plus a generated `PARENT.md`. Only virtual environments, caches, the read-only data mount, and the parent's own `Previous Experiments/` folder are excluded. Inheritance is therefore one generation deep, avoiding recursive copying of ancestral workspaces. The prompt merely lists these paths and asks the agent to read `PARENT.md` first. The archive contents are not injected into the context.

Attempt budget. Each prompt requests 3 scored attempts. Operators with parents first re-run the best inherited configuration once, and the prompt excludes this reproduction from the budget. `Initial` instead first runs the untouched defaults. The budget is a prompt instruction rather than an enforced limit. A member's fitness is the best score among all attempts it recorded, including the reproduction. The shared fragments read as follows, rendered for the reported configuration, where "[...]" marks omitted text:

```
Start from the best inherited configuration: install it, then run it
once to confirm it reproduces. Every measurement below is against that
reproduction.

**The reproduction does not count as one of your attempts.** It is the
starting line, not a result.
[...]
Run 3 attempts. For each one:
```

For each operator, we quote the stance and the first loop step ("1.").

Initial ($n_p = 0$). The member has no access to previous experiments and starts from the default pipeline.

```
No ancestors. Start from scratch.
[...]
Run the untouched defaults first, so you have a number to compare
against.
[...]
1. Make one change you can name.
```

Ablation ($n_p = 1$). The member adds and removes single components of the elite parent's configuration.

```
Add or remove one component at a time, to find out what the score
depends on.
[...]
1. Add or remove exactly one component. Two changes in one attempt
   measures nothing.
```

Merge (normally $n_p = 2$). The child sees the slot's elite parent and one other elite chosen at random, or only the slot's elite if no other elite is available. The intended base is the better-performing parent under the configured metric direction; the choice of what to transfer is left to the agent.

```
Combine components from two ancestors: build from one, borrow from the
other.
[...]
Diff the two ancestors and find where they differ. Take the
higher-scoring one as your base.
[...]
1. Bring across one component from the other ancestor.
```

Continue ($n_p \geq 1$, configurable). The member refines the inherited configuration. The number of parents is configurable (default $n_p = 1$ in our experiments); the elite parent is always included, with additional parents sampled randomly from the elite pool.

```
Refine what already works.
[...]
1. Make one change you can name. Build on what worked.
```

EDA ($n_p = 1$). The member first analyzes the data with disposable scripts and ties every change to a recorded observation.

```
Let the data decide.
[...]
Investigate before you change anything.
[...]
1. Make a change that follows from a specific observation, and name the
   observation in the notes.
[...]
Leave enough budget to act on what you find.
```

Jumpstart ($n_p = 1$). The member uses the parent's artifacts as raw material for a substantially different solution. No external reference material is provided. The operator is implemented for completeness, but it was inactive in the reported experiments (`GA_PROB_JUMPSTART=0`).

```
Use everything the ancestor left you as raw material, and build
something that beats it.
```

B.4 HEDGE BOUNDS ENFORCEMENT

The bounded HEDGE update has three stages: (i) compute per-operator rewards from parent-child gains, (ii) update weights in log space, and (iii) enforce floor/ceiling bounds with iterative rebalancing. The implementation aggregates parent-child gains by observed operator and computes the operator-level mean improvement $\bar{\Delta}_k$. If fewer than two operators are observed in an iteration, the HEDGE update is skipped. Gains are measured against the pre-tournament elite scores for every child that recorded a score, including a child whose protocol files changed. A child without a score contributes no observation.

Rank-based rewards. Rewards are derived from ranks rather than from raw operator means:

$$r_k = \frac{2 \cdot \text{rank}(k)}{|\mathcal{K}_{\text{obs}}| - 1} - 1 \tag{4}$$

where $\text{rank}(k) \in \{0, \ldots, |\mathcal{K}_{\text{obs}}| - 1\}$ orders observed operators by $\bar{\Delta}_k$ (ties are broken by sort order). The worst operator receives $r = -1$ and the best receives $r = +1$, which reduces sensitivity to metric scale and outliers.

Clipped importance weighting. To avoid exploding updates for low-probability operators, we clip the importance factor:

$$\tilde{r}_k = r_k \cdot \min\left(\frac{1}{p_k}, \kappa\right) \tag{5}$$

where p_k is the current bounded sampling probability and $\kappa = 4.0$ in the reported configuration. Operators not sampled in an iteration receive no weight update.

Log-space updates. Weights are updated in log space for numerical stability:

$$\log w_k^{(t+1)} = \log w_k^{(t)} + \eta \cdot \tilde{r}_k \tag{6}$$

with learning rate $\eta = 0.15$ in the reported configuration.

Iterative 2-pass algorithm (max 10 iterations):

1. **Pass 1**: Enforce ceilings. Redistribute excess to below-ceiling tasks proportionally.
2. **Pass 2**: Enforce floors. Take deficit from above-floor tasks proportionally.

Repeat until stable (no changes in an iteration), then renormalize. These bounds operate on the post-softmax probabilities used both for task sampling and for the importance-weighting term in the HEDGE update. Task types with base probability 0 are excluded from the active set before the bounds pass runs. Each slot samples its operator independently, so at most n operators are observed per iteration. An iteration in which all slots draw the same operator therefore produces no update.

B.5 HYPERPARAMETERS

The reported runs share the following search hyperparameters and configured evaluation protocol. Execution-resource settings are distinguished from these algorithmic settings below.

- Population size: $n = 3$. The members of an iteration run concurrently, each in its own container. Container resource settings are reported separately below.
- Continue parents: min $= 1$, max $= 1$
- Data: the dataset is bind-mounted read-only into every container from a pinned, checksummed local copy, and the framework downloads nothing at run time.
- Base task probabilities: Initial $= 0.1$, Continue $= 0.2$, Ablation $= 0.1$, Merge $= 0.1$, Jumpstart $= 0$, EDA $= 0.5$
- HEDGE learning rate $\eta = 0.15$, importance-weighting clip $\kappa = 4.0$
- Configured HEDGE floors: Continue $= 0.10$, Ablation $= 0.05$, Merge $= 0.05$, Initial $= 0.05$, EDA $= 0.05$
- Active HEDGE tasks in these experiments: Initial, Continue, Ablation, Merge, EDA. Jumpstart is disabled by base probability 0 and therefore excluded from updates and sampling.
- HEDGE ceiling: Merge $= 0.30$ (other active tasks uncapped)
- Stopping policy: 20-iteration cap, patience 3 (terminate when best-so-far elite score fails to strictly improve for 3 consecutive iterations)
- Attempt budget: 3 requested attempts per agent, giving a nominal allowance of $3 \times 20 \times 3 = 180$ counted attempts for a full run. Inheriting agents additionally reproduce the best inherited configuration without counting that reproduction against the allowance. The budget is a prompt instruction, not an enforced limit.
- Retries: a member without a score is relaunched up to 3 times per slot and iteration.
- Evaluation protocol: 5 seeds (42–46), each with a 20% holdout (stratified for classification) and a further 20% validation split for early stopping. The configured development score is the mean over these splits; actual model-fit counts are reported separately in Appendix C.3.
- Competitions: the problem type, metric, direction, and submission format come from a per-competition manifest. The search hyperparameters and configured split protocol above are shared across the 16 competitions.
- Model: DeepSeek V4 Flash (0731).
- Disabled capabilities: web tools, sub-agents, skills, and plugins (Appendix A.2).
- Recorded container settings: the saved `runner_memory` field is unset in 10 runs, `36g` in 5, and `64g` in 1. These are recorded container configuration values, not measurements of host RAM or realized memory use; an unset field does not imply unlimited physical memory.
- Harness defaults: context compression as described in Appendix A.5, replanning every 50 agent responses, an LLM request timeout of 900 s, and a shell timeout of 60 s unless the agent passes a longer one. No wall-clock limit is set, and the default turn cap (about 10^7) is not a practical limit.
- AGENTGA 1×1: the same model, agent harness, and configured evaluation protocol, with a population size of one, a single iteration, and a prompt that requests 180 attempts. With a single `Initial` member, there is no parent, no tournament, and no HEDGE update.

C Positioning, Benchmark, and Resource Details

C.1 Representative Positioning

Table 3 places AGENTGA relative to representative prior systems by primary optimized unit and whether a higher-level orchestration algorithm coordinates multiple runs.

Table 3: Representative positioning of AGENTGA. **Primary Unit** denotes what the outer system selects and reuses: an artifact, a bounded agentic execution, or an autonomous run. **Orchestrated** indicates whether a higher-level search coordinates multiple runs. **Agent-seed** indicates whether the optimized object is the initial state that launches a fresh autonomous run.

Method	Primary Unit	Orchestrated	Agent-seed
EvoPrompt (Guo et al., 2024a)	Artifact	✓	✗
ReEvo (Ye et al., 2024)	Artifact	✓	✗
FunSearch (Romera-Paredes et al., 2024)	Artifact	✓	✗
AIDE (Jiang et al., 2025)	Artifact	✓	✗
AlphaEvolve (Novikov et al., 2025)	Artifact	✓	✗
AFlow (Zhang et al., 2025b)	Artifact	✓	✗
DS-Agent (Guo et al., 2024b)	Agentic	✗	✗
Data Interpreter (Hong et al., 2025)	Agentic	✗	✗
AutoML-Agent (Trirat et al., 2025)	Agentic	✗	✗
AutoKaggle (Li et al., 2025)	Agentic	✗	✗
MLZero (Fang et al., 2025)	Agentic	✗	✗
SELA (Chi et al., 2024)	Agentic	✓	✗
I-MCTS (Liang et al., 2025)	Agentic	✓	✗
SE-Agent (Lin et al., 2025)	Agentic	✓	✗
Agent K (Grosnit et al., 2024)	Autonomous	✗	✗
Darwin Gödel Machine (Zhang et al., 2026)	Autonomous	✓	✗
Group-Evolving Agents (Weng et al., 2026)	Autonomous	✓	✗
AGENTGA (Ours)	**Autonomous**	✓	✓

C.2 Benchmark Context

Table 4 lists all 16 Weco-Kaggle Lite competitions used in the reported experiments.

Table 4: Weco-Kaggle Lite benchmark: 16 Kaggle tabular ML competitions from Jiang et al. (2025).

Competition	URL	Total Teams
playground-series-s3e14	kaggle.com/c/playground-series-s3e14	1,877
playground-series-s3e16	kaggle.com/c/playground-series-s3e16	1,431
playground-series-s3e19	kaggle.com/c/playground-series-s3e19	1,174
playground-series-s3e22	kaggle.com/c/playground-series-s3e22	1,543
playground-series-s3e24	kaggle.com/c/playground-series-s3e24	1,910
playground-series-s3e25	kaggle.com/c/playground-series-s3e25	1,633
tabular-playground-series-aug-2022	kaggle.com/c/tabular-playground-series-aug-2022	1,889
tabular-playground-series-feb-2021	kaggle.com/c/tabular-playground-series-feb-2021	1,434
tabular-playground-series-feb-2022	kaggle.com/c/tabular-playground-series-feb-2022	1,257
tabular-playground-series-jan-2022	kaggle.com/c/tabular-playground-series-jan-2022	1,592
tabular-playground-series-jul-2021	kaggle.com/c/tabular-playground-series-jul-2021	1,294
tmdb-box-office-prediction	kaggle.com/c/tmdb-box-office-prediction	1,395
bike-sharing-demand	kaggle.com/c/bike-sharing-demand	3,243
cat-in-the-dat	kaggle.com/c/cat-in-the-dat	1,341
house-prices-advanced-regression-techniques	kaggle.com/c/house-prices-advanced-regression-techniques	4,978
new-york-city-taxi-fare-prediction	kaggle.com/c/new-york-city-taxi-fare-prediction	1,485

C.3 Function Evaluations per Competition

Table 5 reports, for each competition, the number of outer-loop iterations, agent invocations, and scored experiments, together with the tokens consumed. All counts are read from the lineage and usage records of the runs rather than derived from the configuration. An invocation records a mean of 4.0 scored experiments (the 3 requested attempts plus, for inheriting children, the reproduction of the parent), which gives a mean of 12.1 scored experiments per iteration with 3 population slots. Each scored experiment is evaluated with up to 5 models, one per split (a few recorded fewer); the refit on all labeled rows that produces the submission file is not included in this count. Iteration counts range from 4 to 20 (mean 17.0), because a run stops either at the 20-iteration cap or after 3 iterations without improvement of the best development score, and scored experiments per competition range from 52 to 253. Over all 16 competitions, the AGENTGA 1×1 control used 694 scored experiments and $7.49 of LLM calls.

Table 5: Per-competition compute resources for the reported AGENTGA runs, counted from the run records. Tokens include cached prompt tokens. Cached is the share of prompt tokens served from the provider's cache, which are billed at a fraction of the normal price, so most of the token volume costs little.

Competition	Iterations	Invocations	Scored experiments	Tokens (M)	Cached (%)
Bike Sharing Demand	20	60	250	258	82.9
Cat in the Dat	18	54	222	256	85.2
House Prices	11	33	159	199	78.5
NYC Taxi Fare	14	42	122	467	58.2
Playground S3E14	20	60	240	315	79.5
Playground S3E16	20	60	244	395	77.7
Playground S3E19	20	60	253	365	80.5
Playground S3E22	5	15	67	43	82.4
Playground S3E24	20	60	244	318	74.7
Playground S3E25	4	12	52	45	87.6
TPS Aug 2022	20	60	244	371	84.6
TPS Feb 2021	20	60	232	319	78.4
TPS Feb 2022	20	60	224	669	51.6
TPS Jan 2022	20	60	241	352	79.6
TPS Jul 2021	20	60	249	662	53.1
TMDB Box Office	20	60	243	351	83.3
Total	**272**	**816**	**3,286**	**5,385**	**71.6**

Hardware. The compute reported in Table 5 was distributed across 4 identical workstations, with 4 competitions per workstation run sequentially; in practice, approximately 3 of the 4 workstations were active concurrently rather than all 4. Within each active competition the GA dispatches one Docker container per population member per iteration (3 containers per iteration in the reported configuration), so each workstation hosts up to 3 concurrent training containers. Each workstation has an AMD Ryzen Threadripper 7960X CPU (24 cores / 48 threads), 128 GB DDR5-5600 ECC RAM, and 4 TB NVMe SSD plus 16 TB SATA HDD storage. No GPU is mapped into the workflow containers, so all training and scored evaluation in this work is CPU-only. Local hardware runs the Docker workflow image and the GA orchestrator; LLM inference for every model role is provider-hosted (DeepSeek V4 Flash (0731) via API) and is therefore not part of the local-compute footprint.

C.4 Licenses for Existing Assets

Table 6 enumerates the existing assets used in this work, with the version or accession point and the license or terms under which we accessed them. We use each asset within the scope of its stated terms; no scraped data is redistributed, and no asset is re-released as part of this submission.

Table 6: Existing assets used in this paper, with versions and license terms. URLs and identifiers are provided where available; Kaggle competition data is governed by the per-competition rules accepted at download time.

Asset	Version / source	License / terms
Weco-Kaggle Lite benchmark (Jiang et al., 2025)	WecoAI/aideml repository	MIT License
Kaggle competition datasets (16 tasks listed in Table 4)	Per-competition data downloads on kaggle.com	Per-competition rules and Kaggle Terms of Service; not redistributed
AIDE reference results (Jiang et al., 2025)	WecoAI/aideml repository	MIT License (figures used as published baselines, with citation)
AIDE code used for the AIDE (DS) runs (Jiang et al., 2025)	WecoAI/aideml repository, commit `7d2fc95` (v0.2.2 and later upstream fixes)	MIT License
AutoGluon (Erickson et al., 2020)	`autogluon.tabular` 1.6.1 (autogluon/autogluon repository)	Apache License 2.0
LLM used by AGENTGA, AGENTGA 1×1 and AIDE (DS) (DeepSeek V4 Flash (0731))	Provider-hosted API	Used under the respective provider Terms of Service; model weights are not redistributed

D EXTENDED EXPERIMENTAL RESULTS

This appendix expands Section 4 and follows its order: the full leaderboard comparisons, the inheritance evidence, the dynamics of the search, the behavior of the HEDGE controller, and the single-agent control, competition by competition. All numbers are computed from the archived run records by the same scripts as in the main text.

D.1 LEADERBOARD COMPARISONS IN FULL

Table 7 compares AGENTGA with every other arm. For each, it reports wins, losses, and ties by rank; the mean difference in *Exceeds % of Human* with a bootstrap confidence interval (10,000 resamples of the per-competition differences); a one-sided sign test on the wins; and the range of the mean difference when any single competition is omitted. The comparison with AIDE (DS) is model-matched. The comparison with the published AIDE (GPT-4T) results serves as a reference, and the comparison with AGENTGA 1×1 is matched in allowance rather than in compute (Section 4.4). Among the four arms we ran, AGENTGA is best on 10 competitions (AutoGluon on 4, AGENTGA 1×1 on 2), and it reaches the top decile of the human leaderboard on 4 competitions, compared with 1 for AIDE (DS), 1 for AutoGluon, and 0 for AGENTGA 1×1.

Table 7: AGENTGA compared with each other arm over the 16 competitions. The leave-one-out range gives the lowest and highest mean difference obtained when any one competition is omitted.

AGENTGA against	Wins	Losses	Ties	Mean Δ Exceeds [95% CI]	Sign test p (one-sided)	Leave-one-out range
AGENTGA 1×1	12	3	1	+9.93 [-0.97, 22.36]	0.0176	+5.10 to +13.39
AIDE (DS)	14	2	0	+13.52 [7.58, 20.04]	0.0021	+11.60 to +14.68
AutoGluon	11	5	0	+13.24 [2.74, 26.05]	0.1051	+8.51 to +15.09
AIDE (GPT-4T)	14	1	1	+19.75 [12.44, 27.69]	0.0005	n/a

Table 8 places the four arms we ran beside every system whose mean the AIDE authors report on the same benchmark. The published numbers were obtained with a different model, harness, and date, so they are a reference rather than a controlled comparison.

Table 8: Mean *Exceeds % of Human* over the 16 competitions of Weco-Kaggle Lite: the arms we ran, and the systems reported by the AIDE authors (their Table 1).

System	Model	Mean *Exceeds % of Human*
Run by us (this paper)		
AGENTGA	DeepSeek V4 Flash (0731)	71.13
AGENTGA 1×1	DeepSeek V4 Flash (0731)	61.20
AIDE (DS)	DeepSeek V4 Flash (0731)	57.61
AutoGluon	N/A	57.89
Published by the AIDE authors (Jiang et al., 2025), not run by us		
AIDE	GPT-4 Turbo	51.38
Human + ChatGPT	GPT-4 Turbo	41.17
H2O AutoML	N/A	35.34
AutoGPT (LangChain)	GPT-4 Turbo	32.34

D.2 INHERITANCE EVIDENCE IN DEPTH

Section 4.3 compares children that inherit a parent archive with **Initial** children that start from scratch. Table 9 breaks the 759 decided tournaments down by operator, and Figure 6 by phase of the search. Three descriptive comparisons supplement the main contrast. First, within the 41 iterations in which the same run launched both kinds of child against their respective slot elites, **Initial** children won 17.8% of their tournaments and inheriting children 71.8%, so the gap also appears within this restricted comparison. Second, against a fixed bar, the best development score of iteration 1, only 31.1% of later **Initial** proposals clear it, compared with 87.3% of inheriting children. Third, we compare tie and regression rates. An inheriting child ties with the parent in 22.4% of tournaments and ends worse in only 6.4%, whereas 82.2% of **Initial** children end worse than the elite they challenge.

Table 9: Parent-child tournaments by operator over the 16 reported runs (iteration 2 onward). A win is a strict improvement over the elite of the same population slot. Median gain is the relative change of the development score against that elite, signed so that positive is better.

Operator	Parents	Tournaments	Wins	Win rate % [95% CI]	Median gain %	Median gain % when won
Initial	0	45	8	17.8 [9.3, 31.3]	-0.939	+0.399
Continue	≥1	138	93	67.4 [59.2, 74.6]	+0.028	+0.097
Ablation	1	62	29	46.8 [34.9, 59.0]	+0.000	+0.032
Merge	2	66	51	77.3 [65.8, 85.7]	+0.025	+0.069
EDA	1	448	332	74.1 [69.9, 77.9]	+0.024	+0.076
All inheriting operators	≥1	714	505	70.7 [67.3, 73.9]	+0.020	+0.077

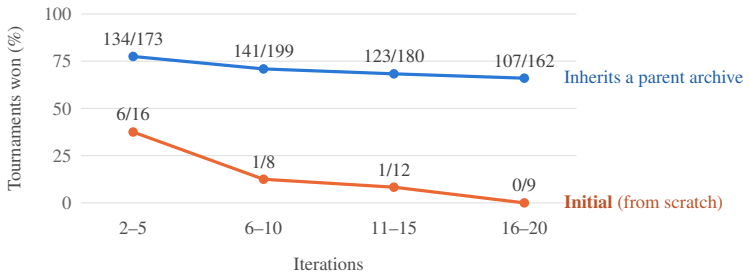


Figure 6: Tournament win rate by phase of the search. De novo **Initial** proposals lose ground as elites improve, whereas children that inherit a parent archive continue to win about two tournaments in three.

Detected archive accesses. The seed only makes the archive available; the agents' tool logs show how it is used. All 717 inheriting agents that ran had detected accesses to the inherited archive (6 further launches failed before their first model call), with a median of 25 tool calls (14.5% of their calls), the first of which occurs 1.8% of the way into the run. Table 10 lists the accessed categories: all had calls accessing the parent's summary and recorded attempts, most also accessed solution code, and fewer accessed version history. Table 11 shows that detected access counts remain approximately stable as archives grow with lineage depth. Between iteration 2 and iterations 16–20, the median inherited archive grows from 105 to 308 text files (from 0.33 to 1.41 MB of text, and from 3.2 to 41.0 MB in total), yet the number of calls that access it stays nearly constant, and the median peak context rises only from 59,845 to 90,748 tokens against a limit of 200,000 tokens.

How access is measured. The measurement uses each agent's event log, which records every tool call together with its arguments. A tool call counts as opening the inherited archive when its arguments contain the path `Previous Experiments/`, whatever the tool: reading or listing a file, or a shell command that copies or runs something from the folder. The path that follows the parent's folder decides what was opened, with the rules applied in this order. `PARENT.md` is the parent's summary. Other Markdown files and notes count as notes and documentation, paths under `attempts/` as recorded attempts, Python files and the solution folders as solution code, and `.git` as git history, as does any `git` shell command that names the folder. JSON, CSV, and other result files count as result files and artifacts, and anything else, such as listing a folder, as a folder listing. A shell command can also change into a parent's folder and name a file afterwards, so for shell commands that mention the folder the whole command is scanned for the same names, and `run.py adopt`, which installs the frozen code of the parent's best recorded attempt, counts as accessing the recorded attempts. A call that names several paths can count in several categories, so a file inside a recorded attempt's frozen `solutions/` counts both as a recorded attempt and as solution code. The position of the first touch is the index of the first such call divided by the agent's total number of tool calls. Uses of files after an agent has copied them out of the folder are not counted, and the proxy does not directly measure which contents the model read.

Table 10: Parts of the inherited archive accessed by inheriting children, over the 717 inheriting agents that ran.

What the child opens in the inherited archive	Invocations touching it (%)	Median calls
the parent's summary (`PARENT.md`)	100.0	2
notes and documentation	98.5	2
solution code	91.8	5
recorded attempts and their results	100.0	10
git history	40.7	0
result files and artifacts	59.4	1
folder listings	99.6	9

Table 11: Size of the inherited archive and the number of detected accesses by each child, by iteration of the search (medians over all inheriting invocations with event logs, including those whose child recorded no score, which is why the counts slightly exceed the number of decided tournaments).

Iterations	Invocations	Text files	Text (MB)	Total (MB)	Calls touching it	Peak context (tokens)
2	41	105	0.33	3.2	21	59,845
3–5	132	128	0.40	5.3	23	60,770
6–10	199	182	0.57	12.7	25	74,378
11–15	183	240	0.83	23.2	26	89,633
16–20	162	308	1.41	41.0	26	90,748

D.3 Search Dynamics

The best development score improved after iteration 1 in 15 of the 16 runs, and 75.0% of all iterations improved the global best. The final best elite lies a median of 12.5 inheriting generations

deep (at most 17), so the reported solutions are the product of long lineages rather than of a single fortunate agent. Agents explore rather than climb monotonically. Of 2,357 new scored experiments (excluding reproductions of the parent), 48.9% beat the inherited parent, 48.5% were worse, and the remaining 60 experiments matched it exactly. 35.6% of invocations contain both improvements and regressions, 243 invocations produced no experiment that beat the parent, and 240 produced only improvements. Each agent keeps its best experiment and the tournament discards the rest, which is why so few inheriting children end worse than their parent.

Parallel lineages. The 1:1 tournament keeps 3 lineages alive instead of truncating to the global best. Two or more population slots held the lead at some point in 13 of the 16 runs, and the lead changed between slots in 13 runs (median 3 changes). In 8 of the 14 runs with at least 10 iterations, the slot that finished best was not leading at the midpoint. The three final elites are distinct solutions in 15 runs, with a median gap of 0.62% between the best and the worst slot. Lineages also exchange material through Merge. The final best elite descends from two or more slots in 8 runs (from all three in 7), its ancestry contains a Merge in 8 runs, and it has a median of 16.5 ancestor invocations, 44.2% of which were never the global best themselves. Intermediate solutions that were never submitted thus shaped the one that was. Patience matters for the same reason: two-iteration plateaus were later broken 12 times, and a patience of 2 instead of 3 would have ended 9 runs prematurely.

D.4 Operator Allocation

Changes in HEDGE probabilities are positively associated with observed operator win rates in most runs. In 13 of the 14 runs in which two or more operators can be compared, the change in an operator's probability correlates positively with its realized win rate (median Spearman $\rho = 0.40$). Table 12 gives the prior and final probability of each operator together with its pooled tournament win rate. **EDA** rises from its prior of 0.5 to a mean final probability of 0.62, while **Initial** falls from 0.1 to 0.06. Reweighting pooled observed operator win rates by the realized and prior mixes gives 67.7% and 64.7%, respectively. This descriptive calculation does not estimate the effect of running a fixed-prior policy. The usefulness of operators does change over the search (Figure 6 shows this for the two kinds of child), which is the setting the bounded update was designed for. Because we did not run a fixed-prior control, however, the marginal value of adaptive allocation is not isolated.

Table 12: HEDGE probabilities per operator: the configured prior and the probability at the end of the run (mean, minimum, and maximum over the 16 runs), together with the operator's pooled tournament win rate.

Operator	Prior	Mean final	Min final	Max final	Pooled win rate (%)
Initial	0.10	0.06	0.05	0.18	17.8
Continue	0.20	0.16	0.10	0.48	67.4
Ablation	0.10	0.07	0.05	0.12	46.8
Merge	0.10	0.09	0.05	0.30	77.3
EDA	0.50	0.62	0.25	0.75	74.1

D.5 The Single-Agent Control in Depth

Table 13 expands the matched-allowance single-agent comparison in Section 4.4, listing the experiments, stopping behavior, and development-score comparison for every competition. 14 of the 16 agents stopped of their own accord before using 90% of the allowance, after a median of 22.5 scored experiments (12.5% of the allowance), and a median of 2 experiments after their last improvement. 11 of them set their best score in the last fifth of their experiments. They did not overlook the instruction. All 14 refer to the allowance while planning, 13 note the unspent allowance in their own final audit, 11 of these attribute it to convergence, a performance ceiling, or noise, and 8 mention a plateau or optimum in their final message. Among the competitions of the early-stopping agents, the two arms share a scoring protocol in 14; in 11 of these, AGENTGA finished with a better development score than the one at which the single agent stopped (last column of the table).

Table 13: The single-agent control, competition by competition: the number of scored experiments and their share of the 180-attempt allowance; the position in the sequence of the last improvement; the context size at the stop; the relative margin by which AGENTGA's final development score exceeds the single agent's best (positive is better); and the agent's own stated reason for stopping, as classified from its final audit.

Competition	Scored exp.	Share (%)	Last gain at (%)	Context at stop	AGENTGA beyond (%)	Own account of stopping
Bike Sharing Demand	30	16.7	79	101,893	+78.14	ceiling reached, converged, within noise
Cat in the Dat	27	15.0	87	123,539	+0.25	not stated
House Prices	182	101.1	99	137,766	-0.98	converged
NYC Taxi Fare	18	10.0	95	109,512	-4.03	converged
Playground S3E14	21	11.7	96	117,612	+0.87	within noise, plateau
Playground S3E16	30	16.7	97	104,181	+1.07	diminishing returns, plateau
Playground S3E19	188	104.4	100	124,492	+4.21	plateau
Playground S3E22	26	14.4	21	70,608	+0.00	plateau, saturated
Playground S3E24	29	16.1	81	95,387	+0.85	converged, within noise, plateau
Playground S3E25	21	11.7	77	74,977	+0.00	not stated
TPS Aug 2022	19	10.6	96	198,042	+1.34	not stated
TPS Feb 2021	14	7.8	100	78,030	+0.16	converged, within noise, plateau
TPS Feb 2022	16	8.9	89	137,229	+0.28	diminishing returns, within noise
TPS Jan 2022	31	17.2	97	151,857	+1.87	converged, within noise
TPS Jul 2021	18	10.0	90	126,509	+37.08	converged, plateau
TMDB Box Office	24	13.3	85	143,823	+4.68	diminishing returns

The conversation as a chain. Section 3.2 describes an agent's context as a chain that every model call re-reads. Figure 7 illustrates this on Playground S3E24: the single agent's chain is one ramp that ends when the agent stops, whereas the line of the reported AGENTGA member is a sequence of 17 short chains, one per generation, each starting from its task prompt. Across all runs, the median single agent's chain grows from 5,316 to 120,576 tokens over 209 model calls, re-reading 8.7M tokens in total. Only 1 of the 16 single agents ever reached the harness's compression threshold, and at the stop the median chain held 60% of the 200,000-token limit, so the agents did not stop because their context was full. The median AGENTGA agent's chain grows from 5,564 to 70,597 tokens over 126 calls, re-reading 3.2M tokens, after which the next generation starts a new one; 12 of the 807 AGENTGA agents whose event logs record context size reached the compression threshold. A complete 20-iteration run starts 60 such chains and makes 7,604 model calls in total. AIDE, by contrast, has no chain: each of its 320 solutions is written from a prompt assembled by its scaffold, with 2.1 LLM calls per solution and 3,228 input tokens per call.

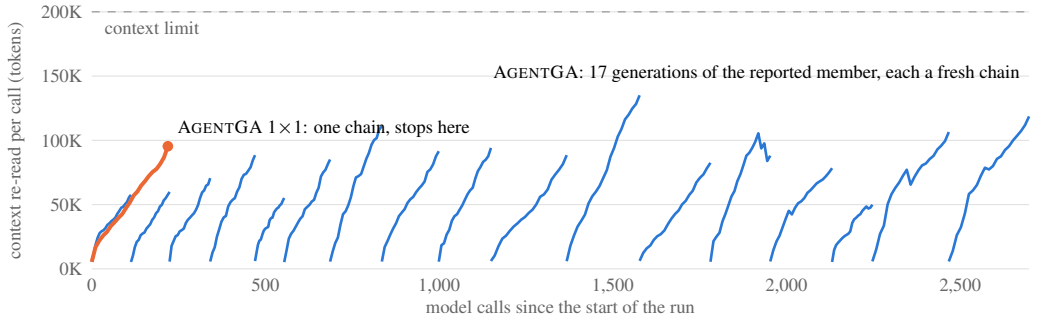


Figure 7: How the conversation grows, on Playground S3E24. Every model call receives the retained working context, so a chain grows until the harness compresses it. AGENTGA 1×1 keeps a single chain until it stops; AGENTGA starts every generation from a fresh chain and passes state on through files.

E LINEAGE TRACING AND VERIFIER ALIGNMENT

This appendix details the lineage tracing and retrospective scoring used to assess verifier–leaderboard alignment in Section 4.5. Among reported runs that completed all 20 iterations, we examine the two with the highest and the two with the lowest *Exceeds % of Human*. For each run, we trace the population member that produced the reported submission back to its iteration-1 **Initial** ancestor. For a Merge child, we follow the parent from the child's own population slot. Each subse-

quent member in the traced lineage replaced its slot's elite, so development performance improves strictly along the lineage.

For each lineage member, we select its saved submission using the same rule as the reported results: order its attempts from best to worst development score and take the first with a submission file. We submit these files unchanged as late submissions to Kaggle and obtain their scores and ranks on the fixed private leaderboard, following the protocol of Table 1. No models are retrained or predictions altered, and each lineage's final submission reproduces the reported result exactly. In total, 77 distinct files were scored, covering the traced lineages and the two leading elites of each run after iterations 1, 5, 10, 15, and 20. These additional checkpoints are included in the rank correlations in Table 14.

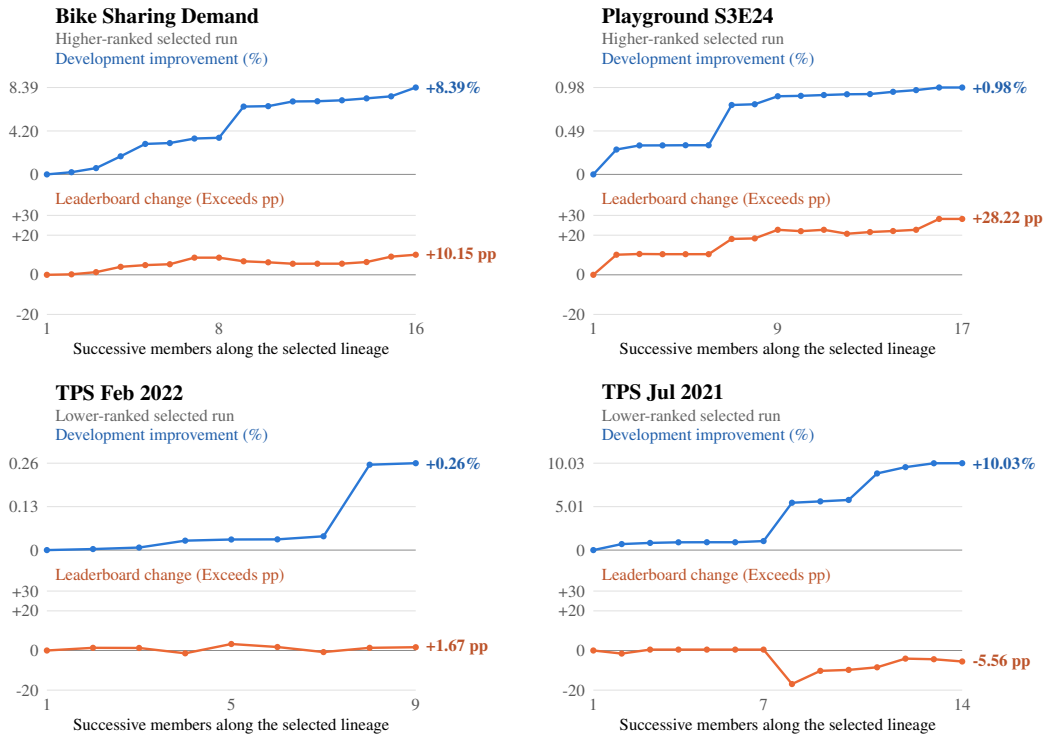


Figure 8: **Development-score improvement does not guarantee leaderboard improvement.** Trajectories of the reported member's lineage in the two highest-ranked and two lowest-ranked completed 20-iteration runs. Each competition has two aligned traces over successive members of its selected lineage: direction-aware relative development-score improvement above (blue), and change in *Exceeds % of Human* below (orange), both relative to the initial ancestor. Development panels use task-specific scales; leaderboard panels share a percentage-point scale. Although development scores improve along all four lineages, leaderboard performance improves in two and remains approximately flat or declines in the other two. Endpoint labels give the total change in each measure.

Table 14 summarizes each line, and Table 15 lists every generation. The training-data column of Table 14 is measured on the competition files: whether the test set lies after the training period, and how much of a random holdout has an exact copy among the training rows.

Table 14: The reported member's line of descent in each competition. Gen.: number of generations in the line, all of which were scored. Verifier: relative change of the development score from the first to the last generation (positive is better). Exceeds: *Exceeds % of Human* of the first and the last generation. Steps up / down: generation steps in which *Exceeds* rose or fell. ρ: Spearman correlation between development score and private score over every scored file of the run.

Competition	Group	Training data against the test set	Gen.	Verifier	Exceeds	Steps up / down	ρ
Bike Sharing Demand	highest	test set inside the training period	16	+8.39%	83.2 → 93.3	10 / 3	+0.46
Playground S3E24	highest	no time order, no duplicate rows	17	+0.98%	70.8 → 99.1	10 / 3	+0.94
TPS Feb 2022	lowest	51% of a holdout has an exact copy in training	9	+0.26%	37.7 → 39.4	4 / 4	-0.18
TPS Jul 2021	lowest	test set after the training period	14	+10.03%	31.5 → 26.0	5 / 4	-0.65

Table 15: Every generation of the reported member's line of descent: the iteration in which it was produced, its population slot, its operator, its development score, and its Kaggle private score, fixed-leaderboard rank, and *Exceeds % of Human*. The last row of each competition is the reported result.

Competition	It.	Pop.	Operator	Development	Private	Rank	Exceeds
Bike Sharing Demand	1	1	Initial	0.27081	0.42880	545	83.19
Bike Sharing Demand	2	1	EDA	0.27024	0.42798	537	83.44
Bike Sharing Demand	3	1	Continue	0.26917	0.42418	502	84.52
Bike Sharing Demand	4	1	EDA	0.26607	0.41720	414	87.23
Bike Sharing Demand	6	1	EDA	0.26283	0.41539	386	88.10
Bike Sharing Demand	7	1	EDA	0.26260	0.41338	372	88.53
Bike Sharing Demand	8	1	Continue	0.26143	0.40502	265	91.83
Bike Sharing Demand	9	1	Continue	0.26123	0.40510	265	91.83
Bike Sharing Demand	10	1	EDA	0.25307	0.40853	324	90.01
Bike Sharing Demand	11	1	Continue	0.25296	0.41034	341	89.49
Bike Sharing Demand	12	1	EDA	0.25173	0.41228	364	88.78
Bike Sharing Demand	13	1	EDA	0.25167	0.41215	363	88.81
Bike Sharing Demand	14	1	EDA	0.25142	0.41205	363	88.81
Bike Sharing Demand	15	1	EDA	0.25093	0.40983	336	89.64
Bike Sharing Demand	17	1	EDA	0.25038	0.40192	248	92.35
Bike Sharing Demand	19	1	EDA	0.24808	0.39848	216	93.34
Playground S3E24	1	1	Initial	0.86952	0.87139	557	70.84
Playground S3E24	2	1	EDA	0.87199	0.87401	363	80.99
Playground S3E24	3	1	EDA	0.87240	0.87417	356	81.36
Playground S3E24	4	1	EDA	0.87241	0.87414	358	81.26
Playground S3E24	5	1	Continue	0.87242	0.87415	358	81.26
Playground S3E24	7	1	Continue	0.87242	0.87415	358	81.26
Playground S3E24	8	1	EDA	0.87643	0.87744	211	88.95
Playground S3E24	9	1	Merge	0.87650	0.87782	206	89.21
Playground S3E24	10	1	Continue	0.87730	0.87867	123	93.56
Playground S3E24	11	1	EDA	0.87734	0.87853	136	92.88
Playground S3E24	12	1	EDA	0.87742	0.87863	123	93.56
Playground S3E24	13	1	EDA	0.87749	0.87829	161	91.57
Playground S3E24	14	1	EDA	0.87751	0.87841	145	92.41
Playground S3E24	15	1	EDA	0.87774	0.87854	135	92.93
Playground S3E24	18	1	EDA	0.87791	0.87862	123	93.56
Playground S3E24	19	1	EDA	0.87816	0.87898	18	99.06
Playground S3E24	20	1	Ablation	0.87816	0.87899	18	99.06
TPS Feb 2022	1	3	Initial	0.99317	0.94888	783	37.71
TPS Feb 2022	2	3	Merge	0.99321	0.95008	766	39.06
TPS Feb 2022	3	3	EDA	0.99325	0.94996	767	38.98
TPS Feb 2022	13	3	Continue	0.99345	0.94707	801	36.28
TPS Feb 2022	16	3	EDA	0.99349	0.95238	742	40.97
TPS Feb 2022	17	3	EDA	0.99350	0.95044	761	39.46
TPS Feb 2022	18	3	EDA	0.99359	0.94795	793	36.91
TPS Feb 2022	19	3	Merge	0.99572	0.95004	766	39.06
TPS Feb 2022	20	3	EDA	0.99576	0.95023	762	39.38
TPS Jul 2021	1	1	Initial	0.12417	0.25170	886	31.53
TPS Jul 2021	2	1	Continue	0.12331	0.25937	906	29.98
TPS Jul 2021	3	1	Ablation	0.12315	0.24845	880	31.99
TPS Jul 2021	4	1	Continue	0.12305	0.25016	880	31.99
TPS Jul 2021	5	1	EDA	0.12305	0.25063	880	31.99
TPS Jul 2021	7	1	EDA	0.12305	0.24968	880	31.99
TPS Jul 2021	9	1	EDA	0.12289	0.24775	880	31.99
TPS Jul 2021	10	1	Merge	0.11739	0.34397	1,105	14.61
TPS Jul 2021	11	1	Continue	0.11720	0.32317	1,019	21.25
TPS Jul 2021	13	1	EDA	0.11698	0.31892	1,013	21.72
TPS Jul 2021	14	1	EDA	0.11319	0.31024	996	23.03
TPS Jul 2021	15	1	EDA	0.11229	0.28274	939	27.43
TPS Jul 2021	19	1	Merge	0.11174	0.28398	943	27.13
TPS Jul 2021	20	1	EDA	0.11172	0.29207	958	25.97

F AGENT TRAJECTORY SHOWCASE: BIKE SHARING DEMAND

This appendix follows the reported Bike Sharing Demand solution from its initial ancestor to its final descendant. We select it as a successful illustration with substantial relative reductions in both development and leaderboard error and a measured increase in *Exceeds % of Human*; it is not a representative or randomly selected run. The gains below accumulate across the lineage, from iteration 1 to iteration 19, rather than arising from a single invocation.

F.1 Search Progress

Table 16 compares the initial ancestor of the reported solution with its final descendant, not the globally best population member at the first iteration. Both scores are RMSLE, for which lower is better. Leaderboard values come from the saved submissions scored retrospectively in Appendix E; they were not available to the agent during search.

Table 16: Bike Sharing Demand: measured ancestor-to-descendant gains. Exceeds improvement is in percentage points; error reductions are relative to the initial ancestor.

Metric	Initial ancestor	Final descendant	Improvement
Development RMSLE	0.27081	0.24808	8.39% reduction
Leaderboard RMSLE	0.42880	0.39848	7.07% reduction
Exceeds % of Human	83.19%	93.34%	+10.15 points
Leaderboard rank	545	216	329 places

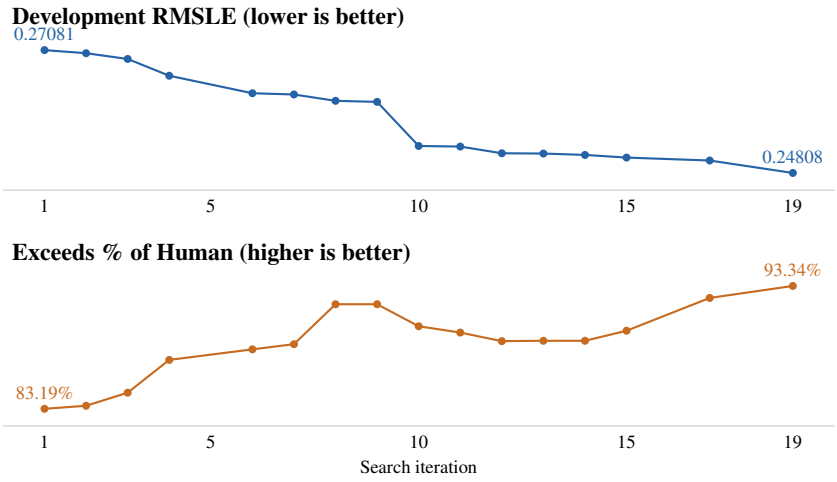


Figure 9: The reported Bike Sharing lineage. Every measured generation is retained, including leaderboard setbacks. Development RMSLE decreases monotonically, but leaderboard performance does not. The panels have separate vertical scales.

The complete checkpoints for this lineage appear in Table 15.

F.2 How the Solution Evolved

The initial solution already used a log-target model with hour-aware features. Subsequent descendants explored ensemble composition, refitting and calibration, and additional model members. Table 17 shows selected recorded attempt notes. They describe the agents' motivations for changes, rather than isolating each change's causal contribution.

Table 17: Selected agent-recorded actions along the Bike Sharing lineage. Notes are quoted, shortened for space.

Iter.	Attempt	Recorded note (truncated)
4	`gbm_ensemble_4member`	3-member ensemble cut rmsle to 0.26656 (seed-noise lever); add a 4th structurally distinct GBM (low subsample/high colsample) to decorrelate errors further
8	`ens6b`	model: ens5 + 2nd low-variance regularized specialist G (slow/shallow, low colsample); continues winning direction after deep member F regressed
10	`ens_refit_scale17`	Refine iteration-scale lever: raise REFIT_ITER_SCALE 1.5-¿1.7. Observation: offline faithful replica shows the 'more trees on the larger refit set' scaling plateau is flat 1.5-1.8 with shallow optimum 1.6-1.7 (1.7 best across ...
17	`a3_mlp2`	obs: a2 showed algorithmic-diversity axis delivers with strong non-tree MLP; add a 2nd orthogonal-sized (128,64) MLP — offline 5-seed holdout a2 0.25063-¿0.25038 (5/5). 12 trees + 2 MLPs.
19	`b3_cat14`	obs: CatBoost new-family axis near-saturated at 14 members — offline on 8-cat bag, +6 more cats 0.248274-¿0.248083 (4/5 seeds), +8 regresses. Take the 14-cat sweet spot.

F.3 What a Long Agent Trajectory Looks Like

We now zoom into the invocation that produced the final solution. Figures 10 and 11 show what its trajectory looks like in practice, using 200 messages retained in a saved LLM request from Bike Sharing Demand (iteration 19, population 1, EDA operator). The first view shows the message sequence and tool interactions; the second groups it into activities, from inspecting inherited work to developing and testing candidates. Table 18 lists the scored attempts in this invocation; these gains are distinct from the full-lineage gains above.

Table 18: Recorded development RMSLE in the showcased Bike Sharing invocation.

Stage	Recorded attempt	RMSLE
Inherited reproduction	`repro`	0.250384
Candidate 1: LightGBM diversity	`b1_lgb`	0.250148
Candidate 2: eight CatBoost members	`b2_cat8`	0.248274
Candidate 3: fourteen CatBoost members	`b3_cat14`	0.248083

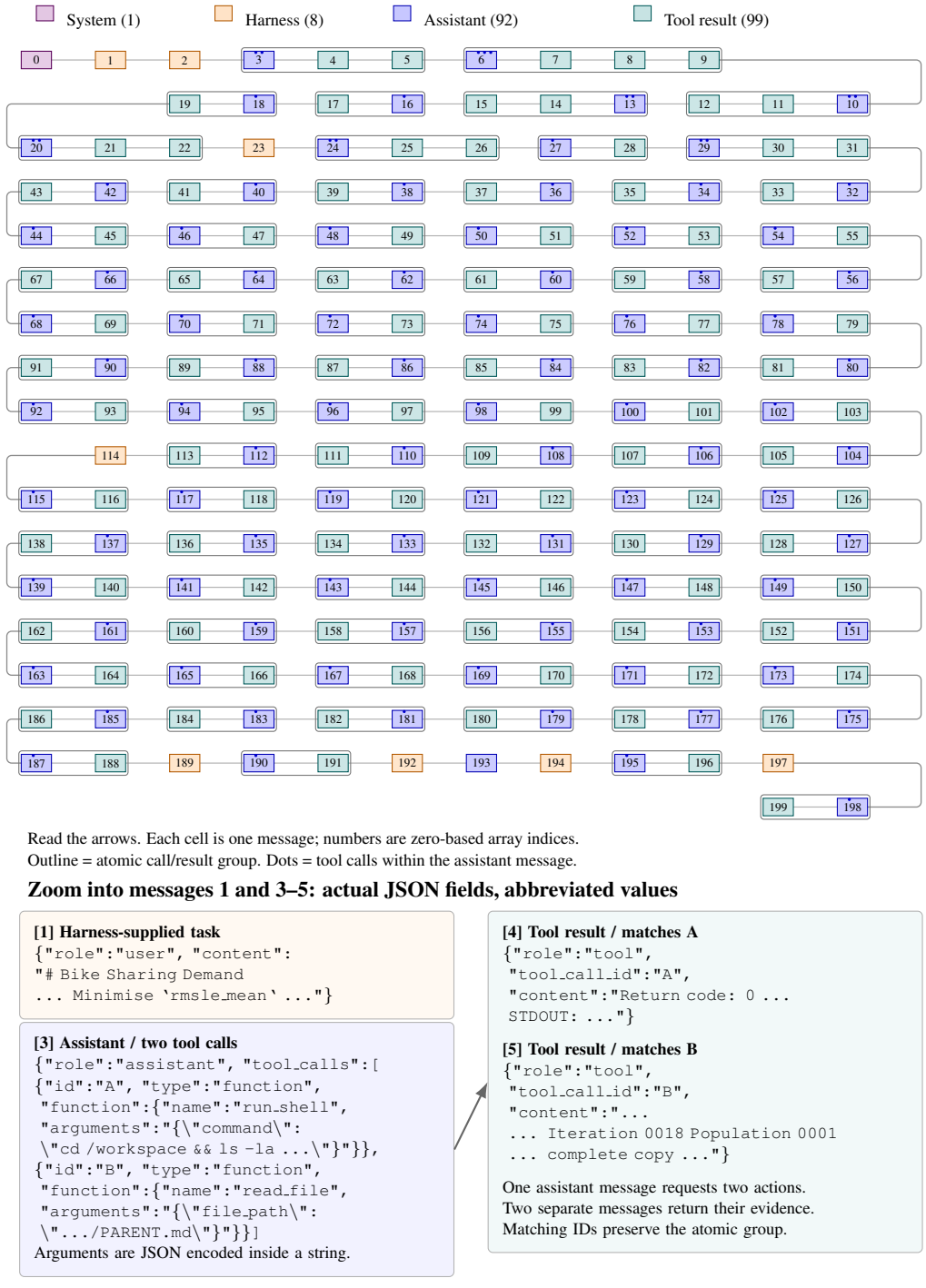


Figure 10: **A recorded agent conversation: Bike Sharing Demand, iteration 19, population 1 (EDA).** All 200 messages in the saved request appear in order. Harness messages use the API's `user` role, not human input. Outlines keep tool calls and matching results together; dots count calls. Close-ups show the task and a two-call exchange, with IDs aliased as A/B and values abbreviated. The figure shows one retained request, not the entire invocation.



Figure 11: **The same conversation grouped by agent activity.** The agent reproduces the inherited solution, evaluates three candidates, and records the final winning submission (Table 18). Bands are author-reviewed annotations of the recorded commands, outputs, and attempts; each describes the dominant activity, including interleaved monitoring and harness handoffs.

G LIMITATIONS AND THREATS TO VALIDITY

This appendix expands on the limitations of the system comparisons, leaderboard metric, and public benchmark.

G.1 END-TO-END SYSTEM COMPARISON VS. CONTROLLED CAUSAL CLAIM

The comparison with AIDE (DS) in Table 1 matches the model but neither the harness nor the compute. Both systems use DeepSeek V4 Flash (0731) for every LLM call, which we verified from the AIDE run configurations. AIDE, however, searches over code artifacts with a tree-search outer loop and localized code generation, whereas AGENTGA launches autonomous agents that manage their own context, follow task instructions, and complete a task over a long-horizon trajectory. We ran AIDE at its default budget of 20 steps, at which it used three orders of magnitude fewer tokens than AGENTGA. Because we did not run AIDE with a larger step budget, Table 1 does not show how AIDE would scale. The published AIDE (GPT-4T) column is a reference obtained with a different model (GPT-4 Turbo), harness, and date, not a controlled arm.

The AGENTGA 1×1 control removes the outer loop while holding the model, agent harness, tools, and workflow layer fixed. It is matched in the attempt allowance offered to the agent, not in the compute consumed. Because the single agent stopped early on most competitions (Section 4.4), the comparison shows that a larger allowance for one agent is not converted into search; it does not show that the two arms are equally expensive. We did not run a fixed-prior control for HEDGE, a population-size ablation, or a variant that continues a single workspace in place, so the marginal contributions of adaptive allocation, population size, and the fresh-workspace reset are not isolated.

The tournament analysis compares descendants conditioned on inherited parent archives against de novo **Initial** proposals *within* AGENTGA runs (i.e., with the model, scaffold, and nominal attempt allowance held constant, with children challenging their respective slot elites). Two caveats apply. First, a child keeps the best of its scored attempts and is compared with a fixed parent score, so absolute win rates include selection on the development splits; both kinds of child are subject to this rule, but its differential effects are not isolated. Second, operator sampling probabilities are adapted by HEDGE rather than held fixed, which is why Section 4.3 also reports the comparison restricted to iterations that launched both kinds of child. Table 1 asks whether the system is competitive on a public benchmark, while the tournament analysis asks whether the inheritance mechanism contributes within the system. We do not claim that Table 1 alone establishes the causal effect of agent-seed evolution.

G.2 LEADERBOARD-RANK METRIC LIMITATIONS

Exceeds % of Human is defined in Section 4 as $100(1 - r/N)$ for official rank r among N teams on the fixed Kaggle leaderboard. Two consequences follow from this definition that are relevant to interpreting the reported numbers.

First, because the metric is rank-based, small absolute differences in the underlying competition score can produce large rank changes when many teams are clustered near the same score, and conversely large absolute score differences can produce small rank changes when teams are sparse. *Exceeds % of Human* should therefore be read as a measure of external leaderboard position relative to the historical Kaggle field, not as a calibrated effect size in the underlying competition metric. We report private-leaderboard *ranks* alongside *Exceeds % of Human* for the traced lineages (Section 4.5).

Second, "*Exceeds % of Human*" refers to the fraction of the Kaggle leaderboard surpassed by the submission, not to a measurement of true human ability on the underlying task. The leaderboard population includes a mixture of serious competitors, casual entries, and ensembled solutions that are themselves the product of automated tooling, and its composition varies across competitions. Cross-paper comparisons that use different reference populations (e.g., different submission cutoffs or filtered subsets of teams) should therefore be interpreted cautiously.

G.3 Benchmark Contamination and Public Kaggle Risk

Weco-Kaggle Lite is built from public Kaggle competitions, several of which have public leaderboards, public discussion forums, and publicly archived top solutions, in some cases predating the training cutoffs of the LLMs used in this work and in baselines. The base models used by AGENTGA and by AIDE may have encountered task descriptions, leakage discussions, feature-engineering recipes, or full solution code for some of these competitions during pretraining. This is a property of LLM-based Kaggle-style benchmarks generally, not of AGENTGA specifically: it affects the AIDE baselines and any other agentic system evaluated on the same suite under the same protocol.

Concretely, this means that the *Exceeds % of Human* numbers reported in Table 1 should be read as system performance under the benchmark's stated protocol, given whatever public prior knowledge the underlying models bring, rather than as a measurement of solving each task from scratch. The shared-model comparisons provide a common source of potential prior exposure. Both the model-matched benchmark comparison and the within-run inheritance analysis use the same base model within each comparison. However, systems and operators may exploit prior knowledge differently, and its effect on relative performance and inheritance comparisons is unmeasured.

We do not attempt to fully decontaminate the benchmark or to construct a held-out competition set, both of which are outside the scope of this paper. A benchmark of more recent or harder-to-search competitions, paired with the same evaluation protocol, would be a useful complement to the present results.

H Code and Reproducibility

The implementation separates the evolution layer (`agentga-ga`) from the tabular domain package (`agentga-kaggle_tabular_ml`). The former coordinates agent seeds, selection, and lineage recording; the latter supplies competition manifests, workspace instructions, and the evaluation interface. An agent backend receives a task brief and a prepared workspace. The reported runs use our harness image; a goal-directed agent such as Codex can instead be integrated through an appropriate launcher. Package layouts, execution commands, and output formats are documented in the repository rather than repeated here.

Release plan. Upon acceptance, we will release the evolution layer and domain package at the version used for the reported experiments, including run scripts, configuration templates, operator prompts and workspace instructions, competition manifests and data checksums, lineage and summary records with recorded Kaggle scores, and the analysis and plot-generation scripts. The release will also include adaptations for other agent harnesses. Provider credentials, model weights, prebuilt images, and the full archive of intermediate experimental artifacts will not be redistributed; users supply their chosen agent backend and competition data.

The recorded artifacts support inspection of the reported experiments, while rerunning the search produces a new stochastic realization rather than an exact replay: neither the evolution layer's random generator nor LLM sampling is seeded. Using another backend likewise changes the execution configuration rather than reproducing the reported harness.