Notebook as World Model

EE290/194 Final Report

Group 4

May 16, 2026

Abstract

We introduce NotEvolve, a training-free agentic framework that uses a Jupyter notebook as both the execution environment and the persistent world model for an LLM agent. Unlike program-centric approaches such as AlphaEvolve, which evolve a single program artifact, NotEvolve evolves an entire *notebook state*—comprising code cells, markdown annotations, execution outputs, and live kernel memory—across iterative rounds. This design unifies LLM-accessible memory (text summaries, prior code, structured plans) with non-LLM memory (kernel variables, cached data structures, intermediate computations) in a single, executable, multimodal document. We demonstrate competitive or superior performance to AlphaEvolve on open mathematical optimization problems, achieve 100% pass rate on select Terminal Bench tasks – substantially outperforming the same model without the notebook harness, show competitive performance to a Claude Code-inspired harness and other SoTA approaches on real-world Kaggle tasks, and superior performance, wall-time, and token consumption on Cuda kernel design benchmarks. Code is available at https://github.com/hengyuf/jupyter-researcher. [1]

1 Introduction

Large language models (LLMs) have demonstrated strong, and in some domains superhuman, capabilities in reasoning, coding, and scientific problem solving. Turning these capabilities into reliable agents, however, remains difficult on complex multi-step tasks: the agent must plan, execute code, inspect intermediate results, preserve useful state, recover from errors, and decide which partial solutions deserve further computation. Prior work has made substantial progress on individual pieces of this problem. Tool-using and code-acting agents interleave reasoning with external actions [16, 14, 15]; memory-based agents retain feedback, skills, or long-term context across interactions [11, 13, 8]; and evolutionary or test-time search systems improve candidate programs through repeated generation and evaluation [9, 6, 10, 4, 1].

Despite this progress, memory, execution, and evolution are often treated as separate design choices. Tool-use systems may execute commands without a durable representation of the evolving workspace; memory systems may store textual summaries without preserving executable computational state; and evolutionary systems typically optimize isolated programs or search policies rather than a full record of experimentation. This separation is especially limiting for scientific and machine-learning workloads, where progress depends not only on the final code artifact but also on intermediate data, plots, diagnostics, failed attempts, and live variables that shape later decisions.

Our goal is to design an agentic system in which memory, execution environment, and evolution state are unified. We propose NotEvolve, a training-free agent framework that uses a **Jupyter notebook as the agent's world model**. The notebook is not merely a user interface: it is the central state object through which the agent perceives, acts, and remembers. It stores text, code, outputs, plots, tables, and execution history in a structured document, while the live kernel preserves

[1] An interactive blog post accompanying this report is available at https://hengyuf.github.io/notevolve-blog/.

non-text state such as Python objects, loaded datasets, fitted models, cached intermediate results, and imported libraries.

This design gives the agent a persistent workspace that is both human-readable and executable. Unlike program-evolution systems that maintain a population of isolated code artifacts, NotEvolve evolves a full notebook trajectory: helper functions, failed attempts, diagnostics, visualizations, and live variables all remain available for later rounds. Unlike text-memory agents, the notebook can re-execute and inspect prior work rather than merely summarize it. The resulting state can serve simultaneously as memory for the LLM, an execution substrate for tools, and the object being refined by test-time search.

Our evaluation studies this claim across mathematical optimization, Terminal-Bench tasks [5], and MLE-bench-style machine-learning engineering [2]. In mathematical optimization, NotEvolve matches or exceeds AlphaEvolve/OpenEvolve-style baselines on the problems tested while using only a small number of notebook sessions. On Terminal-Bench, the same local Nemotron backend improves substantially when placed inside the notebook harness, showing that persistent state and structured execution can compensate for weaker base-model capability. On MLE-bench tasks, the results further suggest that the choice of memory substrate and orchestration policy is as important as the choice of LLM backend.

2 Related Work

Tool-using and code-acting agents. ReAct [16] established the now-standard pattern of interleaving reasoning, tool actions, and observations. Later systems extend this idea with richer action spaces: CodeAct [14] lets agents act through executable code, and SWE-agent [15] shows that the agent-computer interface can strongly affect software-engineering performance. NotEvolve follows this line of work, but shifts the interface from isolated commands or scripts to a persistent notebook whose cells can be edited, executed, inspected, and reused across rounds. This makes the interface itself part of the agent's problem representation: instead of treating execution as a transient side effect, the notebook records the evolving state of the task in a form that is both executable and legible to the model.

Agent memory. Long-horizon agents need memory beyond the current prompt. Reflexion [11] stores verbal feedback from prior attempts, Voyager [13] accumulates reusable skills, and MemGPT [8] frames context management as a memory hierarchy. These systems mostly store text or discrete artifacts. NotEvolve instead uses the notebook as a mixed memory substrate: prompt-visible cells provide LLM-readable memory, while the live kernel preserves computational memory that need not fit in the context window. In this sense, our memory mechanism is not only a retrieval layer but an executable external state, allowing later decisions to condition on prior computations, intermediate datasets, plots, and failed attempts without re-deriving them from scratch.

Program evolution and discovery. FunSearch [9], AlphaEvolve [6], and OpenEvolve [10] demonstrate that LLM-generated programs can be improved through automated evaluation and selection. More recent systems such as EvoX [4] and AdaEvolve [1] further study how the search process itself should adapt over time, for example by evolving search strategies or allocating compute according to improvement signals. These methods motivate our view of agentic work as a test-time optimization process, but their evolving object is typically a candidate solution, program, or search policy. NotEvolve generalizes this view from program evolution to state evolution: the object being improved is an entire executable research workspace, including code, analysis, outputs, intermediate

variables, and failed attempts.

3 Proposed Approach: the Notebook as World Model

The central design choice in NotEvolve is to treat the Jupyter notebook not as a passive transcript, but as the agent's *world model*: the state that is observed, acted on, scored, selected, and evolved. This choice is natural because a notebook combines three roles that are usually separated across different agent systems.

First, a notebook is an evolvable state. Program-evolution systems such as AlphaEvolve [6] evolve code artifacts and store scored programs in an evolutionary database. NotEvolve generalizes the artifact being evolved from a program to a full notebook state: code cells, markdown notes, execution outputs, plots, diagnostics, and round summaries. A branch is therefore not just a candidate program; it is an entire research trajectory that can contain multiple hypotheses, helper functions, failed attempts, and the current best construction. The outer loop can select, clone, merge, or perturb these notebook states in the same spirit that evolutionary systems select programs.

Second, a notebook is a multimodal memory management system. Pure text memory, such as conversation history or a markdown scratchpad, is limited to what can be sent back to the LLM. A notebook provides a dual-layer memory:

- **LLM memory** (text-based): Cell source code, markdown summaries, execution output strings, and structured round-end summaries—all visible to the LLM in the notebook rendering.
- **Non-LLM memory** (kernel state): Python variables, loaded datasets, trained models, cached intermediate results, and imported libraries persist in the Jupyter kernel across rounds. This memory can be reused without being serialized into the LLM context.

Notebook cells also support multiple modalities—code, Markdown/LaTeX, tables, plots, images, and rich display objects—which makes the notebook a more faithful state representation for scientific and data-centric workflows than a plain text log.

Third, a notebook is an executable environment. Unlike a static memory document, each code cell can be executed and re-executed against a persistent kernel. The agent can define a helper in one cell, use it in later cells, revise it, and re-run stale downstream computations. This makes previous work operationally reusable: expensive data loading, search state, cached tables, and trained models can remain alive in kernel memory while the LLM only sees compact summaries of them.

3.1 Framework Architecture

NotEvolve is organized as an outer–inner loop. The **outer loop** maintains a population of notebook branches, where $\mathcal{N}_t^{(i)}$ denotes the i-th notebook branch at round t and $s_t^{(i)}$ denotes its score. After each round, every branch produces an updated notebook state together with logs and evaluator scores. An evolution algorithm then maps the scored branch set to the branch set for the next round. This evolution algorithm is deliberately modular: it can be as simple as best-of-n selection, tournament selection, or MAP-Elites-style archive updates. The key abstraction is that the evolutionary unit is a scored notebook state, not a single function or file.

The **inner loop** is one notebook-agent round executed independently on each branch. It has three phases:

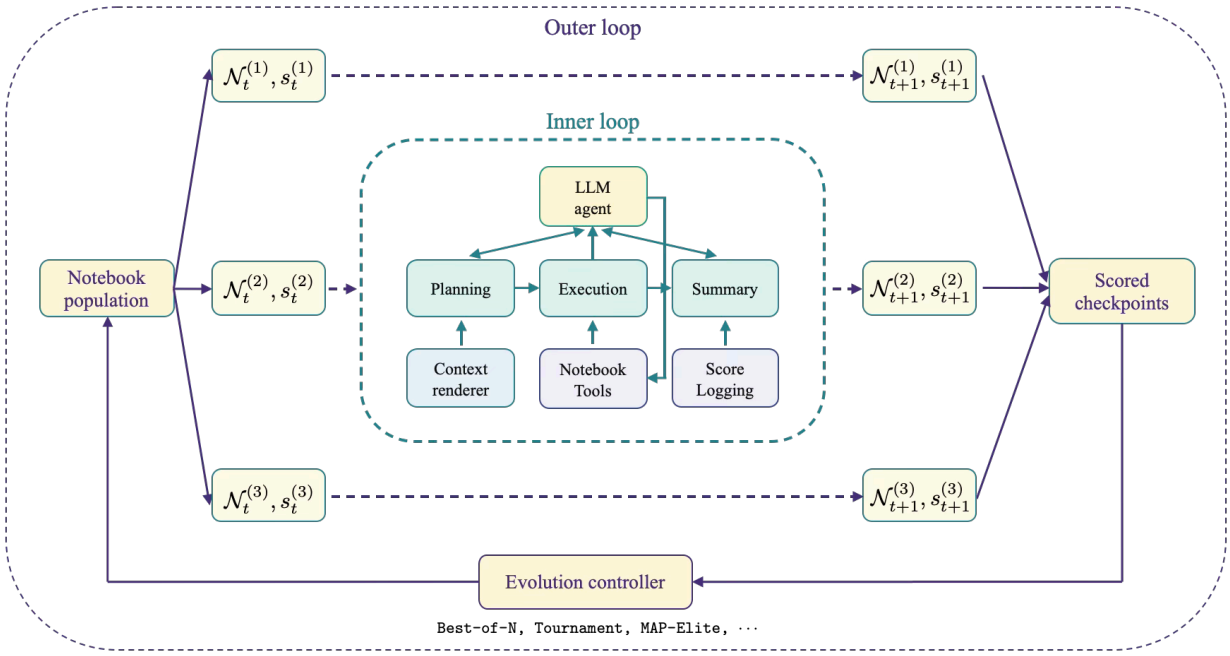


Figure 1: NotEvolve outer–inner architecture. The outer loop maintains a population of notebook branches. At round t, branch i is represented by a full notebook state $\mathcal{N}_t^{(i)}$ and score $s_t^{(i)}$. The `Evolution Controller` fans this population out into branch-specific notebook states, runs a notebook-agent round on each branch, collects updated notebook states $\mathcal{N}_{t+1}^{(i)}$ with scores $s_{t+1}^{(i)}$, and then selects or evolves the next population. This outer update can instantiate simple best-of-n selection, tournament selection, or MAP-Elites-style archive updates. The inner loop is expanded for one branch: `Context_renderer` renders the notebook into compact context, the `LLM Agent` acts through the `Notebook Tool Pool` and `Jupyter Kernel`, the `Evaluator` records scores, and the `Cleaner` summarizes and cleans the notebook before checkpointing the next state.

- **Planning / beginning**: the current notebook state is rendered into a compact text context; the LLM receives the task, scaffold prompt, notebook snapshot, and available tool schemas; its plan is saved back into the notebook.
- **Execution**: the LLM calls notebook tools to add, edit, run, read, fold, unfold, and summarize cells. Code execution goes through a live Jupyter kernel, while evaluator tools log real scores that can later drive the outer evolution loop.
- **Summary / cleanup**: the LLM emits cleanup instructions; the system folds or unfolds cells, deletes obsolete current-round cells, inserts a round summary, and saves a checkpoint.

Three components interact throughout the inner loop: the `Notebook State` as context, the `LLM Agent`, and the `Notebook Tool Pool`. The notebook state is rendered for perception; the LLM decides actions; the tool pool mutates the notebook, executes code, manages context, and reports outputs and scores. Figure 1 illustrates the branch-level architecture, while Algorithm 1 gives the corresponding outer–inner loop pseudocode. In the pseudocode, C_{sys} denotes the fixed system context formed by the task description, scaffold prompt, and tool-use prompt/tool schemas.

Algorithm 1 NotEvolve notebook evolution framework pseudocode.

Input: seed population $\mathcal{P}_0 = \{(\mathcal{N}_0^{(i)}, s_0^{(i)})\}_{i=1}^{m_0}$, round budget T, max tool calls n.
$C_{\text{sys}} \leftarrow$ task description + scaffold prompt + tool-use prompt/tool schemas.
for $t = 0, \ldots, T-1$ **do**
  for each $(\mathcal{N}_t^{(i)}, s_t^{(i)}) \in \mathcal{P}_t$ **do**
    Planning / beginning:
    $C_t^{(i)} \leftarrow \texttt{Context\_renderer}(\mathcal{N}_t^{(i)}) + C_{\text{sys}}$ *render notebooks into compact context*
    $(\mathcal{N}_t^{(i)}, C_t^{(i)}) \leftarrow \texttt{Planner}(\mathcal{N}_t^{(i)}, C_t^{(i)}, \texttt{LLM})$ *make plans for execution*
    Execution:
    for $k = 1, \ldots, n$ **do**
      $\tau_{t,k}^{(i)} \leftarrow \texttt{Propose\_tool}(C_t^{(i)}, \texttt{LLM})$ *propose one tool call*
      $(\mathcal{N}_t^{(i)}, C_t^{(i)}) \leftarrow \texttt{Call\_tool}(\tau_{t,k}^{(i)}, \mathcal{N}_t^{(i)}, C_t^{(i)})$
      Typical tools: add/edit/run/read/summarize/fold/unfold cells.
    Summary / cleanup:
    $\mathcal{N}_{t+1}^{(i)} \leftarrow \texttt{Cleaner}(\mathcal{N}_t^{(i)}, C_t^{(i)}, \texttt{LLM})$ *summarize; delete/fold/unfold cells*
    $s_{t+1}^{(i)} \leftarrow \texttt{Evaluator}(\mathcal{N}_{t+1}^{(i)})$ *log scores*
  $\mathcal{S}_{t+1} \leftarrow \{(\mathcal{N}_{t+1}^{(i)}, s_{t+1}^{(i)})\}_{i=1}^{m_t}$
  $\mathcal{P}_{t+1} \leftarrow \texttt{Evolution\_alg}(\mathcal{S}_{t+1})$ *next population*
return $\arg\max_{(\mathcal{N},s) \in \bigcup_{u=1}^{T} \mathcal{S}_u} s$.

3.2 Context Management: Rendering, Folding, and Cleanup

A raw Jupyter notebook is a JSON document containing cell identifiers, execution counters, metadata, MIME bundles, and full outputs. Sending this file directly to an LLM wastes context on serialization details and makes long-running notebook histories quickly unusable. NotEvolve therefore introduces a renderer and several context-management tools that make the notebook usable as LLM memory.

Rendering to compact text. At the start of a round, the notebook is converted into a structured text view. Each cell is shown with its index, type, execution state, short description, and output summary. Recent or explicitly unfolded cells show full source and clipped outputs; older or explicitly folded cells are shown as one-line summaries. This preserves the notebook's sequential structure while removing JSON boilerplate.

Context tools during execution. The tool pool includes output summarization, cell folding/unfolding, single-cell reading, and full-output expansion tools: `summarize_cell`, `fold_cell`, `unfold_cell`, `read_cell`, and `expand_output`. After each execution, the agent must call `summarize_cell` to convert raw output into a short semantic memory. Long outputs are clipped in tool results and rendered contexts, with explicit pointers to `expand_output` when full details are needed. This lets the agent preserve the information value of an experiment without repeatedly paying for every printed line or traceback.

Round-end compression. At the end of each round, the agent emits cleanup instructions. Obsolete or superseded cells from the current round may be deleted; successful but no-longer-active experiments can be folded; cells that define the current best method or important state can remain unfolded. A structured round summary is inserted as a markdown cell. Thus context compaction is not a one-time preprocessing step, but a continual part of the agent's interaction protocol.

Figure 2 illustrates this context lifecycle across the beginning, execution, and final cleanup phases.

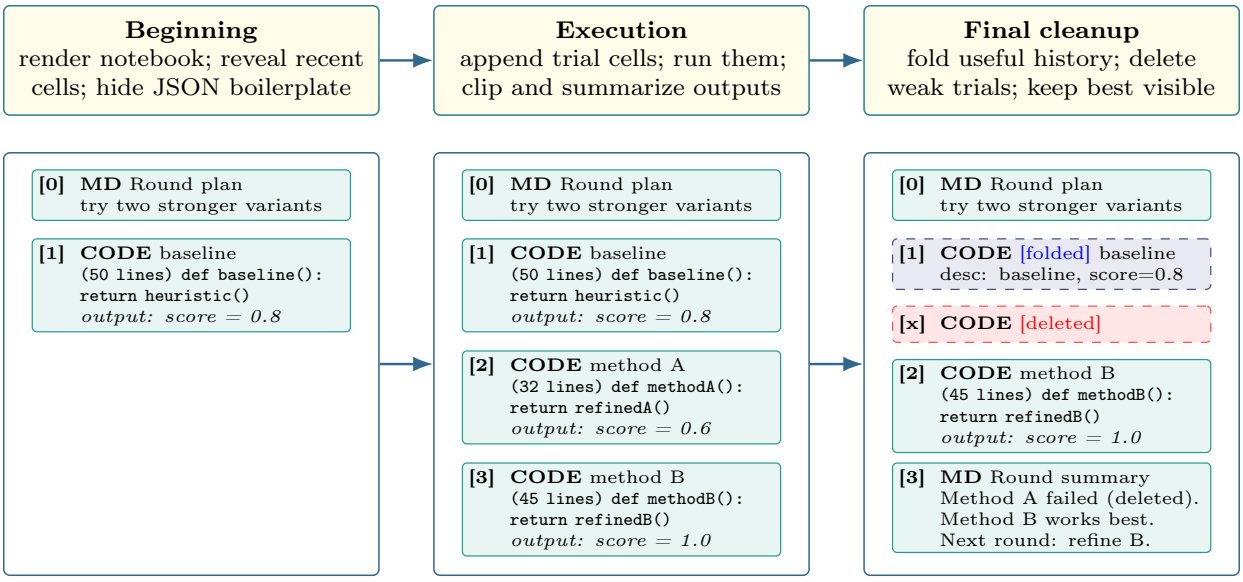


Figure 2: Context management in NotEvolve shown as a two-row, three-phase pipeline. The top row names the context operation performed in each phase; the bottom row shows a concrete rendered-notebook example. Green cells are unfolded and contribute source code plus clipped output to the LLM context. In the execution phase, the agent appends two unfolded trial cells: method A scores worse than the baseline, while method B improves the score. During final cleanup, the baseline is folded into a short description, the low-scoring trial is deleted from the next-round context, and the best trial remains unfolded so the next round can directly inspect and build on it.

4 Results

We evaluate NotEvolve on a few complementary benchmarks: (1) mathematical optimization problems from the AlphaEvolve suite, testing iterative refinement over many rounds, and (2) Terminal Bench tasks, testing single-shot task completion with a locally-served open-weight model, and (3) MLEBench, which tests performance on Kaggle challenges, gauging real-world multi-step coding ability.

4.1 Mathematical Optimization (AlphaEvolve-type Problems)

We compare NotEvolve against the published AlphaEvolve results [6] and OpenEvolve[2], an open-source evolutionary coding agent inspired by AlphaEvolve. All OpenEvolve and NotEvolve runs in this comparison use Gemini 3 Flash as the base model. The benchmark suite contains four mathematical optimization problems where the objective is directly evaluable: circle packing, the Erdős minimum-overlap problem, the Heilbronn triangle problem, and the 3D min–max distance-distribution problem.

For these experiments, NotEvolve uses a lightweight multi-branch inspiration algorithm. We maintain four long-lived notebook branches with independent kernels. After each branch round, the branch writes a scored notebook checkpoint to a shared snapshot database. Before the next round, a branch receives a compact, code-only inspiration block sampled from the best sibling-branch checkpoints. Thus the unit of evolution is a full notebook state, while cross-branch information flow

[2] https://github.com/algorithmicsuperintelligence/openevolve

Table 1: Mathematical optimization results with Gemini 3 Flash. ↑ means higher is better and ↓ means lower is better. **Bold** marks the best or tied-best score. For NotEvolve, we create 4 branches of jupyter books and set the number of rounds to be 4 with 25 maximal tool calls each round. For OpenEvolve, we set the number of evolutionary iterations to be 100 with a maximal program population size being 60.

Problem	**AlphaEvolve**	**OpenEvolve**	**NotEvolve**
Circle Packing ↑	2.635	2.4672	**2.635983**
Erdős Min-Overlap ↓	0.380923	0.4334	**0.38089**
Heilbronn Triangle ↑	**0.03653**	0.0349	**0.03653**
Min–Max Distrib. ↑	0.24004706	0.2300	**0.24005088**

is controlled through short code excerpts rather than copying entire notebooks. Appendix A gives the detailed pseudocode.

With the same base model, NotEvolve obtains the best score on three of the four problems and ties AlphaEvolve on Heilbronn, while OpenEvolve trails on all four objective values. These results suggest that evolving notebook states can be competitive with program-only evolutionary systems on objective-driven mathematical search.

Why Does the Notebook Help? To understand why NotEvolve performs well, we inspect a high-performing Circle Packing[3] branch notebook. Figure 3 shows that the notebook is more than a code transcript: it stores *plots of the current packing*, *persistent variables* such as `best_c`, `best_r`, and `best_s`, reusable solver functions, clipped outputs, and round summaries. These artifacts let later cells diagnose geometric gaps, warm-start from the best known layout, and avoid repeating failed directions.

In this branch, the solver evolves from grid-search baselines to a penalty formulation with LP-based radius evaluation, then to shake/refine and gap-relocation operators, and finally to SLSQP refinement plus inflation-relaxation polishing. The x-axis in Figure 3 is therefore indexed by round and cell rather than by method name: the solver is an evolving sequence of notebook states.

4.2 Terminal Bench (Pass/Fail Agent Tasks)

To evaluate NotEvolve on practical software engineering tasks, we adapt 10 tasks from Terminal Bench [5] and run them with a locally-served NVIDIA Nemotron-3-Nano-30B model (3.5B active parameters via Mixture-of-Experts) served through vLLM. This tests whether the notebook harness can compensate for a smaller, less capable model on tasks that require multi-step reasoning, tool use, and error recovery. Table 2 compares NotEvolve (single trial) against the bare Nemotron model called directly via the Terminal Bench harness (8 trials per task).

The bare Nemotron model achieves only 33.8% pass rate across 80 trials, failing entirely on 5 out of 9 tested tasks. With the NotEvolve harness, the *same model* achieves 100% pass rate on all 11 tasks in a single trial each. This indicates both the capability of weaker models to leverage the the notebook harness effectively, and that weaker model performance can be augmented at test-time on coding tasks by utilizing our notebook-based approach while oftentimes *lowering wall-clock runtime*.

[3]Circle Packing asks the agent to place 26 non-overlapping circles inside the unit square and maximize the sum of their radii. A valid score is therefore the total radius sum after checking boundary and overlap constraints; the AlphaEvolve-level target used in this experiment is 2.635.

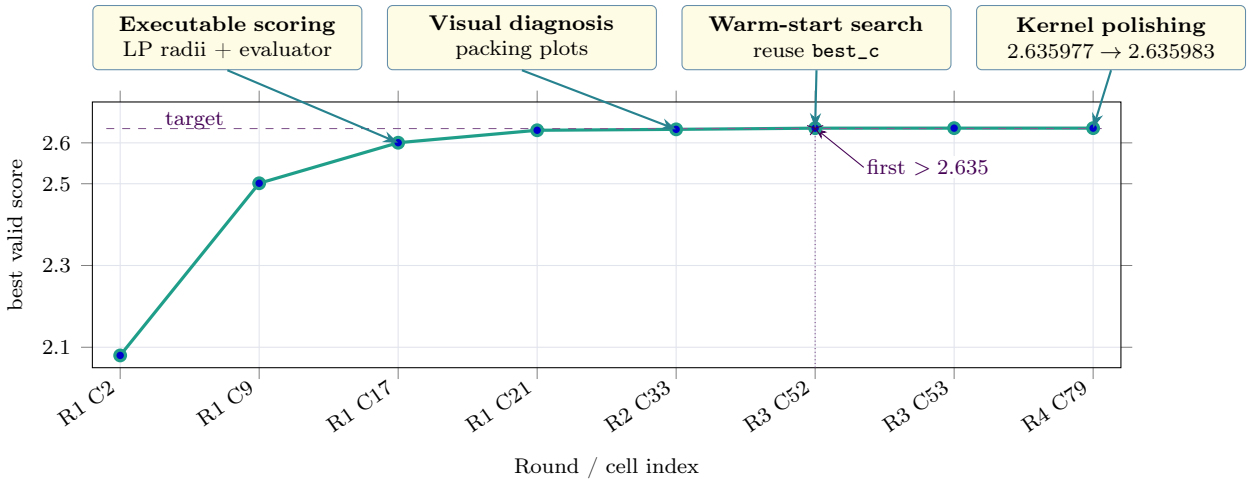


Figure 3: Qualitative mechanism observed in the Circle Packing notebook branch. The x-axis indexes selected notebook rounds and cells where the best valid score improved; solver names are described in the main text rather than used as tick labels. The dashed line marks the AlphaEvolve-level target of 2.635, and the star marks the first cell that exceeds it. The notebook accumulates more than source code: it stores solver definitions, validated best variables such as `best_c`, visualization outputs, failed attempts, and round summaries. Later cells reuse this state as warm starts for increasingly specialized search operators, culminating in a small but measurable final improvement from kernel-state polishing.

Table 2: Terminal Bench results: NotEvolve (single trial) vs. bare Nemotron-3-Nano-30B (8 trials). Time in seconds.

	Bare Nemotron		**Ours (Notebook + Nemotron)**	
Task	**Pass Rate**	**Avg Time**	**Pass Rate**	**Time**
hello-world	8/8	41s	**1/1**	12s
csv-to-parquet	8/8	156s	**1/1**	17s
fix-permissions	8/8	45s	**1/1**	25s
extract-safely	2/8	55s	**1/1**	40s
simple-web-scraper	1/8	163s	**1/1**	51s
download-youtube	0/8	311s	**1/1**	149s
vim-terminal-task	0/8	303s	**1/1**	151s
count-dataset-tokens	0/8	358s	**1/1**	254s
oom	0/8	250s	**1/1**	26s
train-fasttext	0/8	639s	**1/1**	82s

4.3 MLEBench

We next evaluate NotEvolve on MLEBench [2], a benchmark of real-world machine-learning engineering tasks drawn from public Kaggle competitions. These tasks require agents to inspect data, design and debug training pipelines, run experiments, and submit predictions under task-specific metrics. We run all methods with the same locally hosted Nemotron-120B-Super model [7], served on 8×H100 GPUs, and compare four harnesses: NotEvolve (NotEvolve), AIDE with population size one [3], CheetahClaw [12], and OpenEvolve [10].

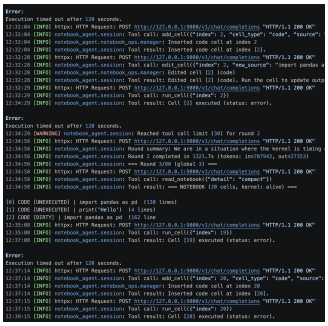
This comparison is intended to probe how different methods to retain memory over test-time optimization affect long-horizon coding performance. NotEvolve stores intermediate development

state in a Jupyter notebook: previous cells, execution outputs, kernel state, diagnostics, and partial results remain available as the agent iterates. AIDE is the closest evolutionary baseline, but represents state as a single evolving code file; setting its population size to one makes the comparison closer to NotEvolve while removing cross-individual search effects. CheetahClaw follows a Claude-Code-style development loop in which the agent edits a directory of files over time, providing a strong baseline for realistic software-engineering workflows. OpenEvolve instead maintains a population of separately evolving code files, allowing multiple solution islands to develop in parallel and exchange progress through selection.

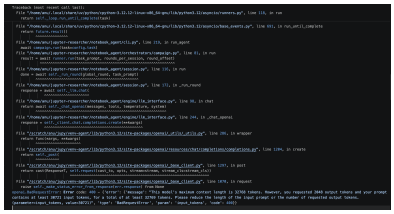
Table 6 shows that NotEvolve is competitive with CheetahClaw, the strongest realistic code-development harness in our comparison. While CheetahClaw is often much faster, NotEvolve consistently produces solutions near the best task score and is best or tied-best on several tasks, including `random-acts-of-pizza`, `tps-dec-2021`, and `dog-breed`. However, NotEvolve generally uses the largest token budget and has high wall-clock time. The wall-clock cost is dominated by CPU-side execution rather than LLM querying, which we empirically found was due to notebook agent proposing more complex or compute-intensive pipelines. The larger token budget, however, does reflect the cost of exposing rich notebook state to the model, and motivates future work on more sophisticated context compaction for notebook representations.

Failure Cases and Mitigations The MLEBench runs exposed several practical failure modes of notebook-based agents that are not captured by final task score alone. These failures were most visible in long-running Kaggle tasks where the notebook history became large, cells occasionally reached the execution timeout, and the model had to decide when to stop. We treat these primarily as orchestration failures rather than fundamental limitations of the notebook representation: in many cases the agent had already produced a valid predictive submission, but the surrounding control policy failed to terminate cleanly, recover efficiently from runtime errors, or protect the final artifact from further edits. Table 4 summarizes common failures and the steps we took to mitigate them.

(a) Repeated endings and duplicate completion cells.



(b) Repeated execution of cells that had already timed out.



(c) Context-limit failure after notebook state growth.

Figure 4: Qualitative failure cases observed in early MLEBench notebook-agent runs. These screenshots illustrate redundant finalization behavior, repeated timeout loops, and model-context saturation.

Prompt and harness changes. The updated MLEBench prompt preserves the original benchmark objective but makes the run-control contract more explicit. It adds: (i) a 120s timeout warning and a required kernel restart after timeouts, (ii) an explicit instruction to verify the final submission exactly once, (iii) an explicit instruction to stop by returning plain text rather than another tool call, and (iv) a stronger persistent-state hint encouraging the agent to reuse loaded datasets, fitted preprocessors, and

trained models. In addition to the prompt changes, we increased the model context length and reduced the minimum run requirement so successful runs can terminate as soon as a real trained model, local validation metric, and correctly formatted submission are present.

Some failure modes remain. In particular, agents still occasionally attempt to regenerate the submission after completion or emit repeated ending statements instead of cleanly exiting the harness. These behaviors point to the limits of prompt-only control. Future work should train or tune the agent on harness interaction traces, including supervised fine-tuning or reinforcement learning from successful stop decisions; structure the prompt so the stop condition is a separate state in the task protocol rather than a final paragraph; and add stricter supervision, potentially from a lightweight checker model or rule-based controller, that tracks whether the task is already complete and prevents further notebook mutations after the terminal condition has been met.

These mitigations do not change the scoring objective or relax the requirement that submissions come from a real model trained on the competition data. Instead, they constrain the agent's control flow around failure recovery and termination. The result is a cleaner separation between productive test-time optimization—EDA, modeling, validation, and ensembling—and non-productive orchestration loops such as repeated final checks, repeated execution of known-expensive cells, or post-completion submission edits.

4.4 KernelBench

We further evaluate NotEvolve on KernelBench, a systems-oriented benchmark that tests whether an agent can generate correct and efficient GPU kernels for PyTorch workloads. Each task provides a reference implementation, and the agent must produce an optimized kernel that both passes correctness checks and improves runtime over the baseline. This setting is a natural stress test for notebook-based agents because progress often requires a long-horizon compile–execute–debug loop: the agent must inspect the workload, propose candidate kernels, run the benchmark harness, interpret compiler or correctness failures, and revise the implementation based on measured performance.

We compare NotEvolve against CheetahHarness on KernelBench Level 1 and Level 2 tasks over three seeds. In contrast to MLEBench, where much of the difficulty lies in data exploration and model-pipeline design, KernelBench emphasizes low-level systems optimization and tight feedback from the execution harness. The notebook state is useful in this setting because it can preserve candidate kernels, benchmark outputs, error messages, timing results, and intermediate debugging notes across attempts. Rather than rediscovering the same failure modes from scratch, later iterations can build on the executable history stored in the notebook.

Figure 5 and Table 5 summarize the results. Overall, NotEvolve achieves a higher mean speedup than CheetahHarness, with an average speedup of 1.096 compared to 0.982 across 300 trials. NotEvolve also produces more successful accelerations: it achieves speedup greater than 1.0× in 76 trials, compared with 49 for CheetahHarness, and speedup greater than 2.0× in 15 trials, compared with 4 for CheetahHarness. These gains are most pronounced on Level 1, where NotEvolve obtains a mean speedup of 1.170 compared with 0.992 for CheetahHarness. On Level 2, the two methods are much closer, with mean speedups of 0.966 and 0.969, respectively.

These results suggest that notebook-based state is most helpful when the optimization problem admits incremental improvement through repeated execution feedback. For easier Level 1 kernels, preserving prior candidate implementations and benchmark traces can guide the agent toward more effective local refinements. For harder Level 2 kernels, however, the bottleneck may shift from

memory management to domain-specific kernel expertise, making the advantage of the notebook less pronounced.

KernelBench also exposes an important cost tradeoff. NotEvolve uses fewer input tokens than CheetahHarness on average, suggesting that the notebook representation can reduce the amount of prior context that must be repeatedly re-specified. However, NotEvolve produces more output tokens and has slightly higher wall-clock time per trial. This indicates that notebook-based memory is not automatically cheaper: the agent still spends substantial effort generating, testing, and revising candidate kernels through repeated compile–run–debug cycles. Future work should therefore combine notebook memory with stronger harness engineering, such as structured kernel templates, automatic profiling summaries, compile-error categorization, and caching of correctness and runtime measurements.

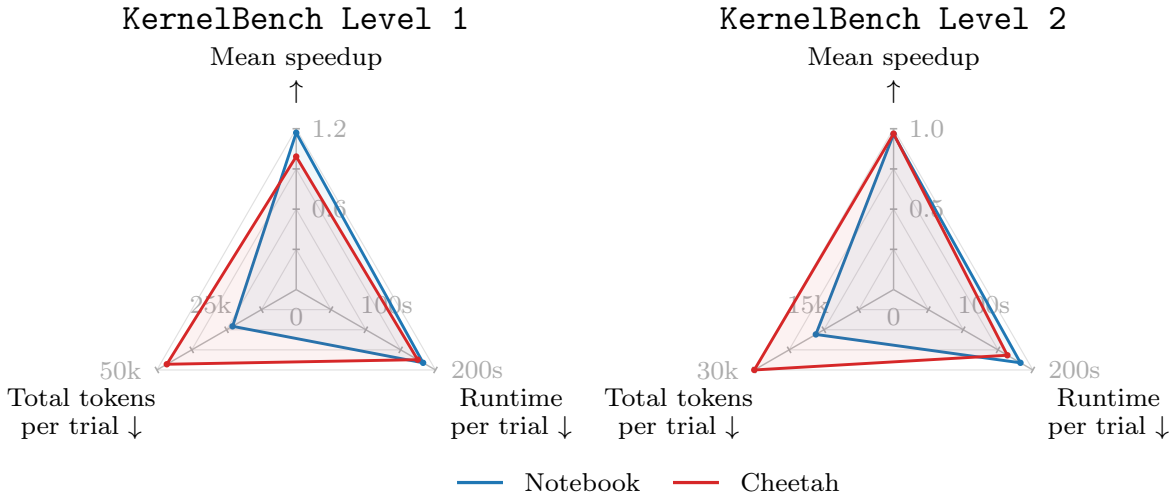


Figure 5: Radar plots for KernelBench Level 1 and Level 2 over three seeds. Each axis is linear and starts at zero, with the outer radius set to the largest observed value within that level and quantity. Polygons use means only. Total tokens are computed as input plus output tokens per trial. For speedup, higher is better; for tokens and runtime, lower is better, so lower-cost methods appear closer to the center on those axes.

5 Conclusion and Impact Statement

We proposed NotEvolve , a training-free agentic framework that uses a Jupyter notebook as a unified state for memory, execution, and evolution. By evolving the full notebook state—code cells, markdown plans, outputs, visualizations, summaries, and live kernel variables—NotEvolve gives the LLM both an executable workspace and a persistent world model. NotEvolve matches or exceeds AlphaEvolve/OpenEvolve-style baselines on open mathematical optimization problems, substantially improves the open-weight Nemotron-3-Nano-30B model on Terminal Bench, remains competitive on realistic machine-learning engineering tasks, and achieves higher overall mean speedup than CheetahHarness on KernelBench. These results suggest that for long-horizon agentic work, the state representation itself is a central part of agent capability.

Future work should focus on making notebook-state evolution more efficient, robust, and broadly applicable. The main practical limitation is token usage: notebooks can accumulate large histories, so better state distillation, cell folding policies, output summarization, and cleanup strategies are needed to preserve useful information while reducing context cost. Another direction is improving

kernel-state reliability through stronger checkpointing, replay, dependency tracking, and sandboxing. Finally, NotEvolve can be extended to broader domains such as kernel/system optimization, compiler and runtime tuning, simulation-driven design, and larger scientific or machine-learning pipelines, where iterative experimentation and reusable computational state are essential.

Table 3: Performance, token usage, and runtime across Kaggle tasks. Each task is grouped into metric score, tokens consumed, and runtime. Runtimes are reported as mean ± standard deviation. Best values are bolded within each row; for metric rows, best follows the metric direction, while for tokens and runtime lower is better. Superscripts 1/2/3 correspond to the original gold/silver/bronze markers.

Nemotron					
Task	**Measure**	**NotEvolve**	**AIDE**	**CheetahHarness**	**OpenEvolve**
random-acts-of-pizza	AUC ↑	**0.7799 (0.7690±0.0076)**	0.7746 (0.7307±0.0621)	0.7689 (0.6616±0.0897)	0.531 (0.526±0.007)
	Tokens	4.20 M	151 k	586 k	**37k**
	Runtime	21.1 min ± 12.5 min	54.6 min ± 15.8 min	**114 s ± 39 s**	58m38s ± 1h14m
nomad2018	RMSLE ↓	0.0621 (0.0618±0.0003)	0.0675 (0.0670±0.0007)	**0.0603 (0.0651±0.0034)**	0.259 (0.259±0.000)
	Tokens	2.56 M	368 k	542 k	**25k**
	Runtime	16.8 min ± 13.5 min	347.6 min ± 243.9 min	**70 s ± 21 s**	10m43s ± 7m41s
spaceship-titanic	acc ↑	0.8149 (0.8134±0.0014)	0.7977 (0.7950±0.0038)	**0.8218 (0.8218±0.0000)**	0.802 (0.802±0.000)
	Tokens	1.89 M	123 k	868 k	**83k**
	Runtime	46.3 min ± 18.1 min	35.8 min ± 10.9 min	**97 s ± 52 s**	8m24s ± 3m03s
spooky-author	LL ↓	0.4164 (0.4351±0.0154)	0.5494 (1.1055±0.3819)	0.4634 (0.4788±0.0219)	**0.363 (0.363±0.000)**
	Tokens	5.06 M	159 k	227 k	**45k**
	Runtime	28.4 min ± 13.5 min	18.3 min ± 19.2 min	**60 s ± 15 s**	9m47s ± 2m37s
jigsaw-toxic	AUC ↑	**0.9748 (0.9679±0.0052)**	**0.9748 (0.9744±0.0003)**	0.9730 (0.7633±0.1483)	—
	Tokens	988 k	**110 k**	1.41 M	—
	Runtime	35.9 min ± 8.5 min	149.4 min ± 136.5 min	**186 s ± 85 s**	—
tps-dec-2021	acc ↑	**0.9608 (0.9519±0.0102)**	0.9405 (0.9347±0.0081)	0.8934 (0.8714±0.0156)	0.908 (0.888±0.029)
	Tokens	834 k	113 k	367 k	**72k**
	Runtime	39.1 min ± 21.3 min	50.9 min ± 6.4 min	**219 s ± 103 s**	3h55m ± 5h22m
tps-may-2022	acc ↑	0.9270 (0.7346±0.1923, n=2)	0.9041 (0.8910±0.0184)	0.9082 (0.8855±0.0160)	**0.982 (0.979±0.004)**
	Tokens	5.09 M	92 k	1.02 M	**46k**
	Runtime	91.2 min ± 10.2 min	38.3 min ± 21.3 min	**130 s ± 28 s**	6m40s ± 1m48s
dog-breed	LL ↓	**0.7782 (1.1619±0.3941)**	0.8275 (1.0326±0.1768)	4.1775 (3.3142±1.3650)	4.814 (4.814±0.000)
	Tokens	2.57 M	153 k	870 k	**100k**
	Runtime	16.7 min ± 2.9 min	50.4 min ± 14.1 min	**356 s ± 167 s**	8m14s ± 34s
nyc-taxi (5.3GB)	RMSE ↓	4.4589 (5.2449±0.5828)	5.3568 (322.0±447.7)	4.6838 (6.8923±2.1806)	**3.119 (3.185±0.092)**
	Tokens	3.59 M	120 k	963 k	**68k**
	Runtime	37.4 min ± 19.6 min	13.2 min ± 4.9 min	**353 s ± 39 s**	6m56s ± 5s
dogs-vs-cats (1.2GB)	LL ↓	0.3542 (0.4718±0.1071)	**0.1213 (0.4024±0.2138)**	0.6645 (0.6175±0.1126)	0.628 (0.628±0.000)
	Tokens	4.80 M	146 k	2.70 M	**39k**
	Runtime	33.7 min ± 35.8 min	112.7 min ± 60.8 min	529 s ± 152 s	**3m47s ± 2m01s**

Table 4: Observed MLEBench failure cases and mitigations.

Failure case	**Observed behavior**	**Mitigation**
Repeated endings after a valid submission	After generating and checking a valid `submission.csv`, the agent sometimes continued to append cells such as "Task completed", "Done", and additional final-summary snippets. This inflated notebook length, token usage, and runtime without improving the submission.	We rephrased the task prompt to require exactly one final verification, a one-line summary cell, and then a plain-text stop signal rather than further tool calls. The strongest additions were the all-caps instructions to verify once and stop immediately after the final submission is ready. We also reduced the minimum-run requirement so the harness can honor early completion once the submission and validation metric exist.
Repeated execution after cell timeouts	When a training or preprocessing cell exceeded the 120s execution timeout, the agent sometimes tried to rerun similar heavy cells or continued using a potentially wedged kernel. This led to repeated timeout cycles and, in some runs, even simple diagnostic cells timing out afterward.	The prompt now states that each cell times out after 120s and that the agent must restart the entire notebook after a timeout before proceeding. We also made the persistent-state hint more explicit: reuse loaded data, fitted preprocessors, cached features, and trained models rather than reloading or refitting them every round.
Model context limitations from growing notebooks	Long notebook trajectories accumulated many cells, large outputs, repeated verification blocks, and debugging logs. As this state approached the model context limit, the agent became more likely to repeat checks, forget earlier completion evidence, or fail API calls due to prompt length.	We increased the model context length so the notebook state could be represented more faithfully. We also added prompt pressure to keep the finalization path short and avoid redundant validity checks, reducing the amount of unnecessary state that accumulates late in a run. Longer-term, this motivates notebook-specific context compaction and output-folding policies.
Working-directory and artifact confusion	Some runs read or wrote `submission.csv` relative to the repository root rather than the per-competition working directory, or repeatedly searched for sample submissions with slightly different filenames.	The prompt now emphasizes the absolute working directory, absolute final submission path, and strict header copying from `sample_submission.csv` or `sampleSubmission.csv`. This narrows the agent's search space and makes artifact validation deterministic.
Post-completion artifact mutation	An ongoing failure case is that the agent may try to update, regenerate, or re-verify the submission after it has already produced a valid final artifact. This can accidentally replace a better submission with a weaker one or create unnecessary nondeterminism late in the run.	The current mitigation is prompt-based and harness-based: stop immediately after one validation pass. A stronger future mitigation is a deterministic supervisor that marks the task as done and blocks further notebook-editing or submission-writing tool calls once the final artifact passes validation.

Table 5: KernelBench results over three seeds.

Metric	**Level 1 (50 × 3 = 150)**	**Level 2 (50 × 3 = 150)**	**Overall (300 trials)**
Speedup			
Notebook mean ± std	1.170 ± 1.340	0.966 ± 0.455	1.096 ± 1.107
Cheetah mean ± std	0.992 ± 0.499	0.969 ± 0.263	0.982 ± 0.417
Notebook wins $> 1.0\times$	42	34	76
Cheetah wins $> 1.0\times$	22	27	49
Notebook wins $\geq 1.5\times$	17	3	20
Cheetah wins $\geq 1.5\times$	9	3	12
Notebook wins $\geq 2.0\times$	13	2	15
Cheetah wins $\geq 2.0\times$	4	0	4
Tokens per trial			
Notebook input mean ± std	$20{,}029 \pm 15{,}393$	$13{,}599 \pm 8{,}747$	$16{,}814 \pm 12{,}906$
Cheetah input mean ± std	$44{,}577 \pm 50{,}405$	$28{,}139 \pm 21{,}512$	$36{,}386 \pm 39{,}597$
Notebook output mean ± std	$2{,}803 \pm 1{,}224$	$3{,}143 \pm 709$	$2{,}973 \pm 1{,}013$
Cheetah output mean ± std	$1{,}882 \pm 486$	$1{,}935 \pm 299$	$1{,}908 \pm 404$
Wall-clock per trial (seconds)			
Notebook mean ± std	182.2 ± 0.0	182.2 ± 0.0	182.2 ± 0.0
Cheetah mean ± std	174.4 ± 16.9	163.4 ± 20.4	168.9 ± 19.5

References

[1] Mert Cemri, Shubham Agrawal, Akshat Gupta, Shu Liu, Audrey Cheng, Qiuyang Mang, Ashwin Naren, Lutfi Eren Erdogan, Koushik Sen, Matei Zaharia, Alex Dimakis, and Ion Stoica. Adaevolve: Adaptive llm driven zeroth-order optimization. *arXiv preprint arXiv:2602.20133*, 2026.

[2] Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Mądry. Mle-bench: Evaluating machine learning agents on machine learning engineering. *arXiv preprint arXiv:2410.07095*, 2024.

[3] Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. Aide: Ai-driven exploration in the space of code. *arXiv preprint arXiv:2502.13138*, 2025.

[4] Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z. Pan, Alexander Du, Kurt Keutzer, Alvin Cheung, Alexandros G. Dimakis, Koushik Sen, Matei Zaharia, and Ion Stoica. Evox: Meta-evolution for automated discovery. *arXiv preprint arXiv:2602.23413*, 2026.

[5] Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

[6] 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. *arXiv preprint arXiv:2506.13131*, 2025.

[7] NVIDIA, :, Aakshita Chandiramani, Aaron Blakeman, Abdullahi Olaoye, Abhibha Gupta, Abhilash Somasamudramath, Abhinav Khattar, Adeola Adesoba, Adi Renduchintala, Adil Asif, Aditya Agrawal, Aditya Vavre, Ahmad Kiswani, Aishwarya Padmakumar, Ajay Hotchandani, Akanksha Shukla, Akhiad Bercovich, Aleksander Ficek, Aleksandr Shaposhnikov, Alex Gronskiy, Alex Kondratenko, Alex Neefus, Alex Steiner, Alex Yang, Alexander Bukharin, Alexander Young, Ali Hatamizadeh, Ali Taghibakhshi, Alina Galiautdinova, Alisa Liu, Alok Kumar, Ameya Sunil Mahabaleshwarkar, Amir Klein, Amit Zuker, Amnon Geifman, Anahita Bhiwandiwalla, Ananth Subramaniam, Andrew Tao, Anjaney Shrivastava, Anjulie Agrusa, Ankur Srivastava, Ankur Verma, Ann Guan, Anna Shors, Annamalai Chockalingam, Anubhav Mandarwal, Aparnaa Ramani, Arham Mehta, Arti Jain, Arun Venkatesan, Asha Anoosheh, Ashwath Aithal, Ashwin Poojary, Asif Ahamed, Asit Mishra, Asli Sabanci Demiroz, Asma Kuriparambil Thekkumpate, Atefeh Sohrabizadeh, Avinash Kaur, Ayush Dattagupta, Barath Subramaniam Anandan, Bardiya Sadeghi, Barnaby Simkin, Ben Lanir, Benedikt Schifferer, Benjamin Chislett, Besmira Nushi, Bilal Kartal, Bill Thiede, Bita Darvish Rouhani, Bobby Chen, Boris Ginsburg, Brandon Norick, Branislav Kisacanin, Brian Yu, Bryan Catanzaro, Buvaneswari Mani, Carlo del Mundo, Chankyu Lee, Chanran Kim, Chantal Hwang, Chao Ni, Charles Wang, Charlie Truong, Cheng-Ping Hsieh, Chenhan Yu, Chenjie Luo, Cherie Wang, Chetan Mungekar, Chintan Patel, Chris Alexiuk, Chris Holguin, Chris Wing, Christian Munley, Christopher Parisien, Chuck Desai, Chunyang

Sheng, Collin Neale, Cyril Meurillon, Dakshi Kumar, Dan Gil, Dan Su, Dane Corneil, Daniel Afrimi, Daniel Burkhardt Eliuth Triana, Daniel Egert, Daniel Fatade, Daniel Lo, Daniel Rohrer, Daniel Serebrenik, Daniil Sorokin, Daria Gitman, Daria Levy, Darko Stosic, David Edelsohn, David Messina, David Mosallanezhad, David Tamok, Deena Donia, Deepak Narayanan, Devin O'Kelly, Dheeraj Peri, Dhruv Nathawani, Di Wu, Dima Rekesh, Dina Yared, Divyanshu Kakwani, Dmitry Konyagin Brandon Tuttle, Dong Ahn, Dongfu Jiang, Dorrin Poorkay, Douglas O'Flaherty, Duncan Riach, Dusan Stosic, Dustin Van Stee, Edgar Minasyan, Edward Lin, Eileen Peters Long, Elad Segal, Elena Lantz, Elena Lewis, Ellie Evans, Elliott Ning, Eric Chung, Eric Harper, Eric Pham-Hung, Eric W. Tramel, Erick Galinkin, Erik Pounds, Esti Etrog, Evan Briones, Evan Wu, Evelina Bakhturina, Evgeny Tsykunov, Ewa Dobrowolska, Farshad Saberi Movahed, Farzan Memarian, Fay Wang, Fei Jia, Felipe Soares, Felipe Vieira Frujeri, Feng Chen, Fengguang Lin, Ferenc Galko, Fortuna Zhang, Frankie Siino, Frida Hou, Gantavya Bhatt, Gargi Prasad, Geethapriya Venkataramani, Geetika Gupta, George Armstrong, Gerald Shen, Giulio Borghesi, Gordana Neskovic, Gorkem Batmaz, Grace Lam, Grace Wu, Greg Pauloski, Greyson Davis, Grigor Nalbandyan, Guoming Zhang, Guy Farber, Guyue Huang, Haifeng Qian, Haran Kumar Shiv Kumar, Harry Kim, Harsh Sharma, Hayate Iso, Hayley Ross, Herbert Hum, Herman Sahota, Hexin Wang, Himanshu Soni, Hiren Upadhyay, Huy Nguyen, Iain Cunningham, Ido Galil, Ido Shahaf, Igino Padovani, Igor Gitman, Igor Shovkun, Ikroop Dhillon, Ilya Loshchilov, Ingrid Kelly, Itamar Schen, Itay Levy, Ivan Moshkov, Izik Golan, Izzy Putterman, Jain Tu, Jan Baczek, Jan Kautz, Jane Polak Scowcroft, Janica Rosenberg, Jared Casper, Jarrod Pflum, Jason Grant, Jason Sewall, Jatin Mitra, Jeffrey Glick, Jenny Chen, Jesse Oliver, Jiacheng Xu, Jiafan Zhu, Jialin Song, Jian Zhang, Jiaqi Zeng, Jie Lou, Jill Milton, Jim Chow, Jimmy Zhang, Jinhang Choi, Jining Huang, Jocelyn Huang, Joel Caruso, Joey Conway, Joey Guman, Johan Jatko, John Kamalu, Johnny Greco, Jonathan Cohen, Jonathan Raiman, Joseph Jennings, Joyjit Daw, Juan Yu, Julio Tapia, Junkeun Yi, Jupinder Parmar, Jyothi Achar, Kari Briski, Kartik Mattoo, Katherine Cheung, Katherine Luna, Keith Wyss, Kevin Shih, Kezhi Kong, Khanh Nguyen, Khushi Bhardwaj, Kirill Buryak, Kirthi Shankar Sivamani, Konstantinos Krommydas, Kris Murphy, Krishna C. Puvvada, Krzysztof Pawelec, Kumar Anik, Laikh Tewari, Laya Sleiman, Leo Du, Leon Derczynski, Li Ding, Lilach Ilan, Lingjie Wu, Lizzie Wei, Luis Vega, Lun Su, Maarten Van Segbroeck, Maer Rodrigues de Melo, Magaret Zhang, Mahan Fathi, Makesh Narsimhan Sreedhar, Makesh Sreedhar, Makesh Tarun Chandran, Manuel Reyes Gomez, Maor Ashkenazi, Marc Cuevas, Marc Romeijn, Margaret Zhang, Mark Cai, Mark Gabel, Markus Kliegl, Martyna Patelka, Maryam Moosaei, Matthew Varacalli, Matvei Novikov, Mauricio Ferrato, Mehrzad Samadi, Melissa Corpuz, Meng Xin, Mengdi Wang, Mengru Wang, Meredith Price, Micah Schaffer, Michael Andersch, Michael Boone, Michael Evans, Michael Z Wang, Miguel Martinez, Mikail Khona, Mike Chrzanowski, Mike Hollinger, Mingyuan Ma, Minseok Lee, Mohammad Dabbah, Mohammad Shoeybi, Mostofa Patwary, Nabin Mulepati, Nader Khalil, Najeeb Nabwani, Nancy Agarwal, Nanthini Balasubramaniam, Narimane Hennouni, Narsi Kodukula, Natalie Hereth, Nathaniel Pinckney, Nave Assaf, Negar Habibi, Nestor Qin, Neta Zmora, Netanel Haber, Nick Reamaroon, Nickson Quak, Nidhi Bhatia, Nikhil Jukar, Nikki Pope, Nikolai Ludwig, Nima Tajbakhsh, Nir Ailon, Nirmal Juluru, Nirmalya De, Nowel Pitt, Oleg Rybakov, Oleksii Hrinchuk, Oleksii Kuchaiev, Olivier Delalleau, Oluwatobi Olabiyi, Omer Ullman Argov, Omri Almog, Omri Puny, Oren Tropp, Otavio Padovani, Ouye Xie, Parth Chadha, Pasha Shamis, Paul Gibbons, Pavlo Molchanov, Peter Belcak, Peter Jin, Pinky Xu, Piotr Januszewski, Pooya Jannaty, Prachi Shevate, Pradeep Thalasta, Pranav Prashant Thombre, Prasoon Varshney, Prerana Gambhir, Pritam Gundecha, Przemek Tredak, Qing Miao, Qiyu Wan, Quan Tran Minh, Rabeeh Karimi Mahabadi, Rachel Oberman, Rachit Garg, Rahul Kandu, Raina Zhong, Ran El-Yaniv, Ran Zilberstein, Rasoul Shafipour, Renee Yao, Renjie

Pi, Richard Mazzarese, Richard Wang, Rick Izzo, Ridhima Singla, Rima Shahbazyan, Rishabh Garg, Ritika Borkar, Ritu Gala, Riyad Islam, Robert Clark, Robert Hesse, Roger Waleffe, Rohit Varma Kalidindi, Rohit Watve, Roi Koren, Ron Fan, Ruchika Kharwar, Ruisi Cai, Ruoxi Zhang, Russell J. Hewett, Ryan Prenger, Ryan Timbrook, Ryota Egashira, Sadegh Mahdavi, Sagar Singh Ashutosh Joshi, Sahil Modi, Samuel Kriman, Sandeep Pombra, Sanjay Kariyappa, Sanjeev Satheesh, Santiago Pombo, Saori Kaji, Satish Pasumarthi, Saurav Mishra, Saurav Muralidharan, Scott Hara, Sean Narenthiran, Sebastian Rogawski, Seonjin Na, Seonmyeong Bak, Sepehr Sameni, Seth Poulos, Shahar Mor, Shantanu Acharya, Shaona Ghosh Adam Lord, Sharath Turuvekere Sreenivas, Shaun Kotek, Shaya Gharghabi, Shelby Thomas, Sheng-Chieh Lin, Shibani Likhite, Shiqing Fan, Shiyang Chen, Shreya Gopal, Shrimai Prabhumoye, Shubham Pachori, Shubham Toshniwal, Shuo Zhang, Shuoyang Ding, Shyam Renjith, Shyamala Prayaga, Siddhartha Jain, Simeng Sun, Sirisha Rella, Sirshak Das, Smita Ithape, Sneha Harishchandra S, Somshubra Majumdar, Soumye Singhal, Sri Harsha Singudasu, Sriharsha Niverty, Stas Sergienko, Stefana Gloginic, Stefania Alborghetti, Stephen Ge, Stephen McCullough, Sugam Dipak Devare, Suguna Varshini Velury, Sukrit Rao, Sumeet Kumar Barua, Sunny Gai, Suseella Panguluri, Sushil Koundinyan, Swathi Patnam, Sweta Priyadarshi, Swetha Bhendigeri, Syeda Nahida Akter, Sylendran Arunagiri, Tailling Yuan, Talor Abramovich, Tan Bui, Tan Yu, Terry Kong, Thanh Do, Thomas Gburek, Thorgane Marques, Tiffany Moore, Tijmen Blankevoort, Tim Moon, Timothy Ma, Tiyasa Mitra, Tomasz Grzegorzek, Tomer Asida, Tomer Bar Natan, Tomer Keren, Tomer Ronen, Traian Rebedea, Trenton Starkey, Tugrul Konuk, Twinkle Vashishth, Tyler Condensa, Udi Karpas, Ushnish De, Vahid Noorozi, Vahid Noroozi, Vanshil Atul Shah, Veena Vaidyanathan, Venkat Srinivasan, Venmugil Elango, Victor Cui, Vijay Korthikanti, Vikas Mehta, Virginia Adams, Virginia Wu, Vitaly Kurin, Vitaly Lavrukhin, Vladimir Anisimov, Wan Seo, Wanli Jiang, Wasi Uddin Ahmad, Wei Du, Wei Ping, Wei-Ming Chen, Wendy Quan, Wenliang Dai, Wenwen Gao, Will Jennings, William Zhang, Xiaowei Ren, Xiaowen Xin, Xin Li, Yang Yu, Yangyi Chen, Yaniv Galron, Yashaswi Karnati, Yejin Choi, Yev Meyer, Yi-Fu Wu, Yian Zhang, Ying Lin, Yonatan Geifman, Yonggan Fu, Yoshi Suhara, Youngeun Kwon, Yuan Zhang, Yuki Huang, Zach Moshe, Zhilin Wang, Zhiyu Cheng, Zhongbo Zhu, Zhuolin Yang, Zihan Liu, Zijia Chen, Zijie Yan, and Zuhair Ahmed. Nemotron 3 super: Open, efficient mixture-of-experts hybrid mamba-transformer model for agentic reasoning, 2026.

[8] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

[9] 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*, 625:468–475, 2024.

[10] Asankhaya Sharma. Openevolve: an open-source evolutionary coding agent, 2025.

[11] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023.

[12] CheetahClaws Team. Cheetahclaws: An extensible, python-native agent system for autonomous multi-model workflows. *github*, 2026.

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

[14] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. *arXiv preprint arXiv:2402.01030*, 2024.

[15] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.

[16] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

A Experimental Evolution Algorithm

Algorithm 2 Multi-branch inspiration evolution used in the mathematical optimization experiments

Input: seed notebook $\mathcal{N}_0$, branch count m, generations G, rounds per generation r, inspirations per round K.
Initialize branches $\{\mathcal{B}^{(i)}\}_{i=1}^{m}$ as copies of $\mathcal{N}_0$ with independent persistent kernels.
Initialize shared snapshot database $\mathcal{D} \leftarrow \emptyset$.
for $g = 1, \ldots, G$ **do**
  for branch $i = 1, \ldots, m$ **do**
    $\mathcal{I}_g^{(i)} \leftarrow \texttt{SampleInspirations}(\mathcal{D}, i, K)$
    $C_{\text{insp}}^{(i)} \leftarrow \texttt{RenderCodeOnly}(\mathcal{I}_g^{(i)})$ *no outputs; clipped code cells*
    $(\mathcal{N}_g^{(i)}, s_g^{(i)}, h_g^{(i)}) \leftarrow \texttt{InnerLoop}(\mathcal{B}^{(i)}, C_{\text{insp}}^{(i)}, r)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathcal{N}_g^{(i)}, s_g^{(i)}, h_g^{(i)}, i, g)\}$
    if token budget exceeded **then** $\mathcal{B}^{(i)} \leftarrow \texttt{Rebirth}(\texttt{OwnBest}(\mathcal{D}, i))$
  if tournament is enabled at generation g **then**
    $i_{\text{lose}} \leftarrow \arg\min_i \texttt{BestScore}(\mathcal{D}, i)$ among eligible branches
    $\mathcal{N}_{\text{donor}} \leftarrow \texttt{SampleTopQuartile}(\mathcal{D} \setminus i_{\text{lose}})$
    $\mathcal{B}^{(i_{\text{lose}})} \leftarrow \texttt{Rebirth}(\mathcal{N}_{\text{donor}})$
return $\arg\max_{(\mathcal{N},s)\in\mathcal{D}} s$ and the final snapshot database $\mathcal{D}$.

A.1 GLM MLEBench Results

Table 6: Performance, token usage, and runtime across Kaggle tasks. Each task is grouped into metric score, tokens consumed, and runtime. Runtimes are reported as mean ± standard deviation. Best values are bolded within each row; for metric rows, best follows the metric direction, while for tokens and runtime lower is better. Superscripts 1/2/3 correspond to the original gold/silver/bronze markers.

GLM				
Task	**Measure**	**NA-GLM**	**AIDE-GLM**	**CH-GLM**
random-acts-of-pizza	AUC ↑	0.6501 (0.6451±0.0041)	0.6486 (0.6087±0.0371)	**0.6724 (0.6361±0.0257)**
	Tokens	1.80 M	**160 k**	235 k
	Runtime	57.2 min ± 38.5 min	22.9 min ± 8.3 min	**170 s ± 28 s**
nomad2018	RMSLE ↓	0.0628 (0.0647±0.0018)	**0.0612 (0.0657±0.0019)**	0.0620 (0.0632±0.0009)
	Tokens	760 k	**213 k**	477 k
	Runtime	19.1 min ± 6.9 min	**208 s ± 125 s**	11.3 min ± 4.4 min
spaceship-titanic	acc ↑	0.8081 (0.7988±0.0071)	0.8092 (0.8008±0.0082)	**0.8126 (0.8073±0.0039)**
	Tokens	969 k	**96 k**	331 k
	Runtime	26.3 min ± 20.8 min	25.8 min ± 17.6 min	**469 s ± 231 s**
spooky-author	LL ↓	0.5771 (0.6090±0.0359)	**0.5099 (0.5746±0.0524)**	0.5734 (0.6209±0.0337)
	Tokens	759 k	**64 k**	501 k
	Runtime	18.0 min ± 4.1 min	14.1 min ± 7.7 min	**12.8 min ± 3.4 min**
jigsaw-toxic	AUC ↑	0.9680 (0.9567±0.0159)	0.9707 (0.9682±0.0024)	**0.9713 (0.9521±0.0143)**
	Tokens	677 k	**95 k**	156 k
	Runtime	37.9 min ± 27.7 min	57.0 min ± 24.9 min	**11.7 min ± 6.8 min**
tps-dec-2021	acc ↑	0.9091 (0.9058±0.0024)	**0.9421 (n=1)**	0.9017 (0.8918±0.0102)
	Tokens	694 k	**29 k**	606 k
	Runtime	37.3 min ± 9.8 min	19.2 min	**18.5 min ± 1.3 min**
tps-may-2022	acc ↑	0.8304 (0.7663±0.0904)	**0.9235 (0.8633±0.0602, n=2)**	0.8211 (0.7231±0.1093)
	Tokens	922 k	**51 k**	616 k
	Runtime	50.0 min ± 26.4 min	17.1 min ± 2.2 min	**568 s ± 223 s**
dog-breed	LL ↓	4.6495 (4.9722±0.4130)	**4.4686 (4.6092±0.1406, n=2)**	4.7489 (4.8660±0.1171, n=2)
	Tokens	1.19 M	**87 k**	907 k
	Runtime	18.9 min ± 19.7 min	53.2 min ± 18.5 min	**16.7 min ± 3.3 min**
nyc-taxi (5.3GB)	RMSE ↓	5.6844 (n=1)	5.5526 (n=1)	**4.9172 (7.2956±2.1166)**
	Tokens	1.05 M	**69 k**	344 k
	Runtime	30.3 min	58.5 min	**11.4 min ± 2.0 min**
dogs-vs-cats (1.2GB)	LL ↓	**0.6382 (0.6419±0.0038, n=2)**	—	0.6481 (0.6964±0.0459)
	Tokens	813 k	—	**356 k**
	Runtime	31.5 min ± 6.5 min	—	**16.5 min ± 9.9 min**