DREAM-RSI: RECURSIVE SELF-IMPROVEMENT THROUGH EVOLVING WORLDS

Anonymous authors
Paper under double-blind review

ABSTRACT

Recursive self-improvement is becoming essential for autonomous AI agents, whose progress depends on discovering high-value solutions across complex domains. Effective exploration drives this process, yet managing and improving exploration strategies remains a major bottleneck. Current systems face a fundamental dilemma: fixed strategies fail to adapt as search spaces scale, while online policy optimization must navigate vast meta-search spaces under delayed, expensive feedback from long-horizon rollouts. We introduce DREAM-RSI, a framework for scalable, recursively self-improving exploration. A lightweight orchestration layer makes exploration explicit and programmable while leaving the underlying base agent unchanged. Our key insight is that accumulated discovery history can act as a replay simulator over the realized search space. By dreaming within this simulator built from historical discovery trees, DREAM-RSI obtains immediate, low-cost off-policy feedback to evaluate and refine exploration policies without repeated, expensive online evaluation. The improved policy is then redeployed online to drive further discovery, continuously expanding the simulator pool in a self-improving loop. Across 9 tasks in 4 domains, DREAM-RSI achieves competitive quality and improves discovery efficiency in several settings.

1 INTRODUCTION

Recursive self-improvement (RSI) has emerged as an ambitious goal for autonomous AI systems (Liu et al., 2026c). A common mechanism underlying RSI is an iterative loop wherein agents propose candidate solutions, evaluate outcomes, and fold the resulting feedback into subsequent iterations. Such loops have driven substantial progress across algorithm design (Novikov et al., 2025; Romera-Paredes et al., 2024), open-ended mathematical optimization (Georgiev et al., 2025; Anthropic, 2026), systems design (Jaber & Jaber, 2026; Cao et al., 2026), and agent self-improvement (Zhang et al., 2026b;c; Lee et al., 2026; Zheng et al., 2026a). As agents grow stronger and self-improvement targets grow harder, discovery demands long-horizon exploration over vast search spaces, often spanning thousands of proposal-evaluation cycles (Ye et al., 2026; OpenAI, 2026). At this scale, orchestrating exploration becomes critical (Zheng et al., 2026b): poor exploration wastes computation and time, sharply limiting the efficiency and scalability of RSI.

Existing approaches largely rely on manually designed exploration strategies that stay fixed throughout discovery (Novikov et al., 2025; Yan et al., 2026b; Du et al., 2026; Jiang et al., 2026; Ye et al., 2026). Such strategies cannot learn from accumulated experience and may repeatedly spend computation on unproductive search directions. Recent work therefore attempts to optimize exploration policies online (Liu et al., 2026a), but this faces two fundamental bottlenecks. First, meta-level feedback is delayed and expensive. Unlike evaluating a single candidate, assessing an exploration policy requires observing how it shapes discovery over many proposal–evaluation cycles. Second, the meta-policy space is vast, so many alternatives must be tried before a good one is found. Together, these make meta-level improvement costly. Each policy demands a long online rollout before yielding useful signal, making the self-improvement loop hard to close at the exploration layer.

To address these bottlenecks, we want to design a fast and cheap simulator of discovery to prune exploration policies before costly online deployment. **Surprisingly, completed discovery histories already provide such a simulator.** Prior work treats past history merely as static textual context (Hu et al., 2025; Ouyang et al., 2026b) or as training data for fine-tuning (Yuksekgonul et al., 2026; Wang

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

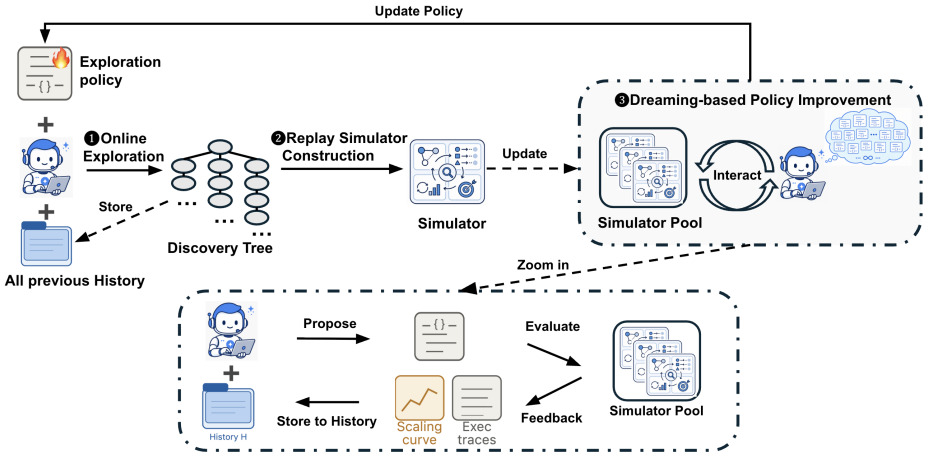


Figure 1: **Overview of DREAM-RSI.** The system runs a recursive self-improvement loop. ① **Online Exploration**: the current policy drives a base agent to expand a discovery tree, logging its execution traces. ② **Replay Simulator Construction**: the discovery tree is distilled into a pool of reusable simulators. ③ **Dreaming-based Policy Improvement**: the same base agent *dreams*, generating many candidate policies internally and replaying them in the simulator for fast, low-cost feedback (see the zoom-in box). The refined policy is redeployed for the next round of exploration.

et al., 2025). Yet a completed discovery process inherently records a structured tree of exploration decisions and their realized code-execution outcomes. By analogy to model-based reinforcement learning and World Models (Ha & Schmidhuber, 2018; Hafner et al., 2023) (§2), this discovery history can serve as a **replay simulator**[1]. Figure 2 shows an alternative strategy can navigate this pre-recorded tree, traversing different subsets of branches in different orders, with different parallel groupings and stopping decisions. Since all execution results are locally stored, evaluating a new strategy requires only reading past records. This turns meta-policy improvement from expensive online trial-and-error into a fast, simulation-based "dreaming" procedure.

Building on this insight, we introduce DREAM-RSI, a framework for scalable, recursively self-improving meta-exploration in agent-driven discovery. We first make exploration programmable through a lightweight orchestration layer that controls branching, parallelism, and stopping while leaving the underlying base agent unchanged. To improve this policy, DREAM-RSI follows a self-improvement loop (Figure 1): **(1) Online Exploration**, where the current policy drives real-world discovery and logs execution traces; **(2) Replay Simulator Construction**, where recorded discovery trees become a reusable replay simulator pool; and **(3) Dreaming-based Policy Improvement**, where candidate policies are cheaply evaluated by dreaming over the simulator. The updated policy is redeployed online, generating new experience that expands the simulator pool and closing an RSI loop at the meta-exploration layer. We evaluate DREAM-RSI on 9 discovery tasks across four domains. On *algorithm engineering* (Lasso path solver), it outperforms `sklearn` and strong baselines while using up to $162\times$ fewer agent calls than SIMPLETES. On mathematical optimization, it matches or exceeds the best results among the compared baselines on two of three tasks. On *GPU kernel engineering* (KernelBench), it reaches similar speeds with $1.79\times$–$2.43\times$ fewer generations, or improves kernel performance by up to $2.09\times$ under a comparable budget. On *harness optimization*, it improves the seed harness by over 8 points on Terminal-Bench 2.1.

Our contributions are: **1) History as Replay Simulator.** We recast completed discovery histories as replay simulators, making delayed exploration feedback reusable for cheap meta-exploration policy evaluation. **2) Meta-Layer RSI Loop.** We introduce DREAM-RSI, to our knowledge the first framework to close a meta-layer RSI loop using discovery history as a replay simulator. **3) Empirical Validation.** Across nine tasks in four domains, DREAM-RSI achieves competitive discovery quality and improves discovery-agent-call efficiency in several settings.

[1]We use *replay simulator* and *world* interchangeably.

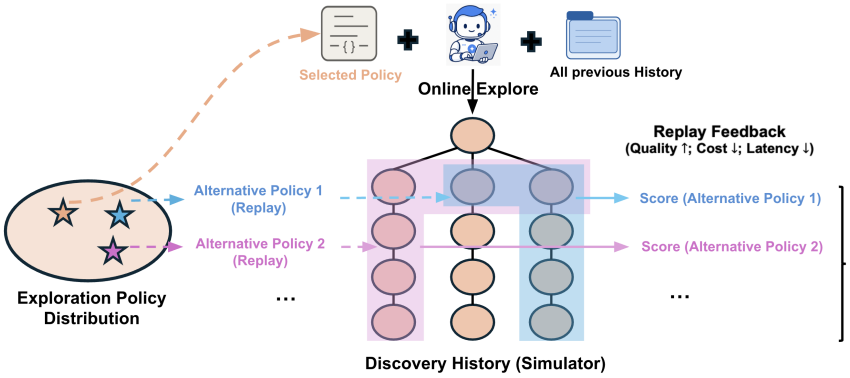


Figure 2: **Discovery history as a replay simulator.** A deployed policy first explores online, recording its execution traces as a discovery tree where each node is one attempt plus its full observation. This tree then acts as a simulator: a large number of candidate policies can be tested inside it, varying which branches to search, in what order, with how much concurrency, and when to stop. **Because every node's outcome is already stored, one expensive online run can yield a large number of fast, low-cost off-policy evaluations.** The agent can therefore "dream" over many alternative exploration strategies via historical replay, and redeploy only the improved policy.

2 Motivation: Discovery History as a Replay Simulator

Consider an agent navigating an unfamiliar environment. On its first attempt it wanders, hits dead ends, and backtracks — but in doing so it builds a map. That map is reusable: a new policy can plan over it, skip known dead ends, revisit earlier choices, and compare routes without physically walking them again (Gupta et al., 2017). This is the logic of model-based reinforcement learning (Sutton, 1990; M. Moerland et al., 2023). A model of how the environment responds to actions lets policies be trained or evaluated in simulation instead of through repeated real interaction (Ha & Schmidhuber, 2018). The Dreamer family (Hafner et al., 2019; 2020; 2023; 2025) is the clearest example: learn a compact dynamics model from collected experience, then improve the policy by imagining trajectories inside it.

Long-horizon discovery has the same structure. An exploration policy decides which directions to pursue, which candidates to refine, what to branch on in parallel, and when to stop. Running it online leaves behind a structured history — branches explored, decisions made, costs incurred, outcomes observed. As shown in Figure 2, we treat this history as an empirical replay simulator: a grounded model of the part of the discovery space already seen. Inside this simulator, a different exploration policy simply traces a different path through the recorded discovery tree, choosing other branches, another ordering, a different degree of parallelism, or an earlier stopping point. Scoring that path only requires revealing outcomes already stored along it; no rerun of the coding agent or evaluator is needed. One expensive online run therefore funds many cheap evaluations of alternative strategies.

3 Dream-RSI: RSI through Evolving Worlds

As shown in Figure 1, Dream-RSI alternates between online exploration and offline "dreaming" for policy evaluation to improve the *exploration policy* that allocates discovery computation. For online rollout, the exploration policy directs a fixed *discovery agent* to generate a tree-structure discovery trace, where each node in the tree is an discovery attempt and is scored by a fixed *evaluator*. The *discovery tree* then becomes a *replay world*: alternative policies can be re-evaluated against stored outcomes. In the offline dreaming phase, a fixed agent iteratively revises the code for the exploration policy, and every revision is evaluated in all stored replay worlds. The best version is then deployed for the next online rollout. Only the policy code changes; the models, evaluator, and execution interfaces stay fixed.

Discovery trees and the shared decision interface. A discovery tree is rooted at r, the initial workspace state. Every non-root node v has exactly one *primary parent* (root or a previous node), which decides where its attempt begins: the discovery agent resumes the parent's saved workspace

and inherits its accumulated observations as context to generate the attempt in v. Node v keeps all previous generation history for it, and its own filesystem snapshot, together with an evaluation score s_v. Scores are generated from a fixed protocol, and larger values are better. We define the root score $s_r = 0$. In both online rollout and offline evaluation, the exploration policy observes a tree $\mathcal{T}$, initially containing only the root, and selects nodes in $\mathcal{T}$ to continue exploration. The eligible nodes are $A(\mathcal{T}) = \{r\} \cup \{v \in \mathcal{T} : v \text{ is a leaf}\}$, which contains the root and all leave node in the current tree. We assume there are $W \geq 1$ parallel workers, each running one generation–evaluation request at a time (e.g. concurrent API calls). For the current tree $\mathcal{T}$, the exploration policy's action is a batch $C \in A(\mathcal{T}; W)$, where $A(\mathcal{T}; W) = \{C \subseteq A(\mathcal{T}) : |C| \leq W\}$ is the feasible batch set. The batch determines both where exploration continues and how many attempts run concurrently. Both the online and offline phase use this decision interface, and differ only in how state transitions after an action: one executes for real, the other replays.

Online rollout. Let $t = 1, 2, \ldots$ index iterations of online rollout, starting from policy π_1 and empty history $\mathcal{H}_0 = ()$. At iteration t, the fixed policy π_t guides the discovery agent to generate a new online rollout with access to the completed discovery history $\mathcal{H}_{t-1}$ as context. The history informs exploration but is kept separate from the tree under construction. Let $\mathcal{T}_t^k$ be the discovery tree after k completed rounds at iteration t, with $\mathcal{T}_t^0 = \{r\}$, and allow at most K_1 rounds. At round $k < K_1$, the policy picks $C_t^k \in A(\mathcal{T}_t^k; W)$ and assigns each $v \in C_t^k$ to a worker, which resumes v's workspace, generates a candidate, and evaluates it. These attempts run in parallel and each adds one child to its selected parent. Thus, every $v \in C_t^k$ is attached with a child, giving $\mathcal{T}_t^{k+1}$. The transition from $\mathcal{T}_t^k$ to $\mathcal{T}_t^{k+1}$ is stochastic, since the agent may produce different outcomes from the same workspace. For the next round, each newly added node becomes the selectable leaf of its branch, while r stays selectable for opening new branches. The rollout ends when $C_t^k = \emptyset$ or after K_1 completed rounds. After the rollout terminates, its final tree is recorded as $\mathcal{T}_t$ and appended to the history, giving $\mathcal{H}_t = \mathcal{H}_{t-1} \cup \{\mathcal{T}_t\}$. The method then enters the offline evaluaton phase using $\mathcal{H}_t$.

Offline evaluation. With $\mathcal{H}_t$ fixed, our method constructs and evaluates $M \geq 1$ policy versions $\pi_t^0, \ldots, \pi_t^{M-1}$, where $\pi_t^0 = \pi_t$. Each version is evaluated on every historical tree $\mathcal{T}_i$, $i = 1, \ldots, t$, under budget-control parameters $\rho \in \mathcal{R} := \{0, 0.25, 0.5, 0.75, 1\}$, where ρ controls the policy's budget preference. The resulting feedback drives the next policy version. We index policy versions by m, replay worlds by i, budget preferences by ρ, and decision rounds within one replay by k. The outer index t is fixed and omitted below. For each (m, i, ρ), replay starts from $\mathcal{T}_{i,\rho}^{m,0} = \{r\}$, where $\mathcal{T}_{i,\rho}^{m,k} \subseteq \mathcal{T}_i$ is the subtree revealed after k completed rounds. The recorded tree $\mathcal{T}_i$ never changes; only the observed portion grows. At round $k < K_2$, the current policy π_t^m, conditioned on ρ, selects $C_{i,\rho}^{m,k} \in A(\mathcal{T}_{i,\rho}^{m,k}; W)$ from the revealed observations. Unlike online execution, offline replay returns recorded children of the selected nodes deterministically rather than generating new candidates. Thus, $\mathcal{T}_{i,\rho}^{m,k+1} = \mathcal{T}_{i,\rho}^{m,k} \cup \bigcup_{v \in C_{i,\rho}^{m,k}} \mathrm{Child}(v; \mathcal{T}_i, \mathcal{T}_{i,\rho}^{m,k})$, where $\mathrm{Child}(v; \mathcal{T}_i, \mathcal{T}_{i,\rho}^{m,k})$ contains unobserved children of v in $\mathcal{T}_i$ given the current revealed subtree. For $v \neq r$, it is v's unique recorded child, if one exists. For $v = r$, it is the earliest-created child of r outside $\mathcal{T}_{i,\rho}^{m,k}$, opening one previously unrevealed branch. In either case, $\mathrm{Child}(v; \mathcal{T}_i, \mathcal{T}_{i,\rho}^{m,k}) = \emptyset$ when no unobserved continuation remains. Newly revealed nodes expose their stored observations before the policy makes its next decision. Replay permits at most K_2 rounds, counting each nonempty batch as one completed round, and stops when $C_{i,\rho}^{m,k} = \emptyset$, after K_2 completed rounds, or when $\mathcal{T}_{i,\rho}^{m,k} = \mathcal{T}_i$. Let $k_{i,\rho}^{m,\star} \in \{0, \ldots, K_2\}$ denote the number of completed rounds at termination. The final observed subtree is $\mathcal{T}_{i,\rho}^{m,k_{i,\rho}^{m,\star}} \subseteq \mathcal{T}_i$. Replay thus measures how far a policy pursues each branch, how it groups attempts into parallel batches, and when it opens a new branch or stops. These choices vary across policies and budget preferences, but every branch is traversed in its recorded parent–child order, and no outcome outside $\mathcal{T}_i$ is produced.

Replay objective. The replay objective balances discovery quality and generation cost across budget preferences while also favoring parallel execution. For each replay world $\mathcal{T}_i$ used for evaluation, define $B_i := |\mathcal{T}_i| - 1 > 0$ and $S_i := \max_{v \in \mathcal{T}_i \setminus \{r\}} s_v > 0$, the total number of evaluated non-root nodes and the best recorded score, respectively. For each $\rho \in \mathcal{R}$, let $n_{i,\rho}^m := |\mathcal{T}_{i,\rho}^{m,k_{i,\rho}^{m,\star}}| - 1$ be the number of discovery-agent calls represented by the replay. Let $s_{i,\rho}^m$ be the best score among the

revealed non-root nodes, i.e., $s^m_{i,\rho} := \max_{v \in \mathcal{T}^{m,k^{m,\star}_{i,\rho}}_{i,\rho} \setminus \{r\}} s_v$ when $n^m_{i,\rho} > 0$, and set $s^m_{i,\rho} := 0$ otherwise. We normalize the attained quality as $a^m_{i,\rho} := \mathrm{clip}(s^m_{i,\rho}/S_i, 0, 1)$. The budget sweep produces the quality–cost points $\{(n^m_{i,\rho}, a^m_{i,\rho}) : \rho \in \mathcal{R}\}$. A point is dominated if another point uses no more calls and attains no lower quality, with at least one strict improvement. Let $(\bar{n}^m_{i,j}, \bar{a}^m_{i,j})_{j=1}^{L^m_i}$ be the distinct nondominated points ordered by increasing call count, and set $\bar{a}^m_{i,0} := 0$. The replay score is

$$V^m_i := \underbrace{\frac{1}{B_i} \sum_{j=1}^{L^m_i} \big(\bar{a}^m_{i,j} - \bar{a}^m_{i,j-1}\big)\big(B_i - \bar{n}^m_{i,j}\big)}_{\text{normalized hypervolume}} - \lambda \underbrace{\frac{1}{|\mathcal{R}|} \sum_{\substack{\rho \in \mathcal{R} \\ n^m_{i,\rho} > 0}} \frac{k^{m,\star}_{i,\rho}}{n^m_{i,\rho}}}_{\text{parallelism penalty}}. \tag{1}$$

Here, λ controls the strength of the parallelism penalty. The first term measures the normalized area dominated by the quality–cost Pareto frontier, rewarding policies that attain higher quality with fewer discovery-agent calls. The second term measures the average number of completed sequential rounds per call across the budget sweep, thereby favoring policies that execute more calls in parallel; replays with zero calls contribute zero to this term.

Policy improvement and selection. A policy version π^m_t is evaluated by its average replay score over $\mathcal{H}_t$, denoted by $V^m = \frac{1}{t}\sum_{i=1}^{t} V^m_i$. The offline phase starts $\pi^0_t = \pi_t$. At each step $m = 0, \ldots, M-2$, a *policy-development agent* inspects the replay trajectories and scores of π^m_t, along with feedback from earlier revisions, to identify successful decisions and recurring failures. It then rewrites the executable policy code into π^{m+1}_t, which is evaluated on the same t replay worlds. The next online policy is the best-scoring candidate, $\pi_{t+1} = \pi^{m^\star}_t$ with $m^\star \in \arg\max_{m \in \{0,\cdots,M-1\}} V^m$. Since the current rollout policy $\pi_t = \pi^0_t$ is included among the candidates, $V^{m^\star} \geq V^0$, so π_{t+1} is never worse than π_t on the average replay score over H_t.

4 Experiments

We evaluate DREAM-RSI on algorithm engineering, kernel optimization, mathematical optimization, and harness optimization. We primarily use Gemini-3.1-Pro for scientific discovery experiments, with additional Lasso experiments using Gemini-3.7-Flash to assess applicability across backbones. For harness optimization, we use Claude Opus 4.8 with Claude Code. On the scientific discovery tasks, our main controlled baseline is Recursive Fixed Exploration. Both methods use identical discovery agents, evaluators, initialization, per-round budget limits, and the same hand-designed starting policy. From the second round onward, Recursive Fixed Exploration keeps its policy fixed, while DREAM-RSI updates it through replay-based dreaming over accumulated discovery history. For harness optimization, we focus on end-to-end performance and omit the fixed-policy comparison due to its substantial cost: an additional full search is estimated to cost over $100,000. We measure discovery cost as the cumulative number of discovery-agent calls.

4.1 Kernel Engineering

The goal of **kernel engineering** is to discover high-performance kernel implementations while preserving numerical correctness. We use four representative KernelBench tasks (Ouyang et al., 2025): **VGG16**, **LayerNorm**, **ConvDiv**, and **ConvMax**. Candidates are scored by execution performance, measured as inverse runtime $(1/\mathrm{ms})$, subject to correctness checks against the reference implementation. With Gemini-3.1 Pro as the coding agent, we compare DREAM-RSI against Recursive Fixed Exploration under an identical evaluation protocol and initialization, and against the strong PaceEvolve baseline (Yan et al., 2026b), whose maximum discovery time we cap at two days for fairness.

Results. Figure 3 shows discovery trajectories as the number of generations increases. On VGG16 and LayerNorm, DREAM-RSI matches final performance with $2.43\times$ and $1.79\times$ fewer generations. On ConvDiv and ConvMax, under comparable budgets, it achieves $2.09\times$ and $1.44\times$ higher performance. Adapting the exploration policy across recursive rounds thus improves both the efficiency and the effectiveness of long-horizon discovery, and DREAM-RSI surpasses PACEvolve in kernel

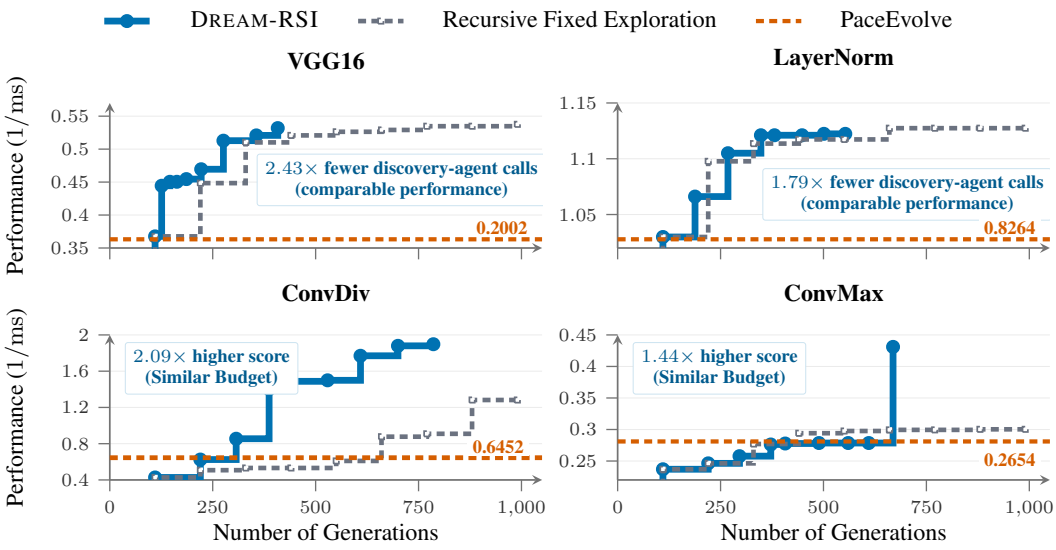


Figure 3: **GPU kernel engineering results.** Discovery performance of DREAM-RSI and Recursive Fixed Exploration as a function of the number of generations. On VGG16 and LayerNorm, DREAM-RSI reaches comparable performance with 2.43× and 1.79× fewer generations, respectively. On ConvDiv and ConvMax, it achieves 2.09× and 1.44× higher performance under comparable discovery budgets. Higher is better for all tasks.

Table 1: Performance comparison on mathematical discovery tasks. Higher is better for **Sum Diff** and **Circle Packing**, while lower is better for **Auto Correlation**. Best results are shown in **bold**.

Method	LLM	Sum Diff (↑)	Auto Correlation (↓)	Circle Packing (↑)
AlphaEvolve	Gemini-2.0 Pro + Flash	–	1.455700	2.635862
AlphaEvolveV2	Gemini-2.0 Pro + Flash	1.121936	–	**2.635983**
OpenEvolve	-	–	1.460000	-
CodeEvolve	-	–	–	2.635980
ShinkaEvolve	Mixed	–	1.457800	2.635982
TTS-Discovery	Qwen3-8B	–	–	**2.635983**
ThetaEvolve	Distilled-Qwen3-8B	–	1.493000	**2.635983**
EvoX	Gemini-3.0-Pro	–	1.458900	2.635900
SimpleTES	GPT-OSS-120B	1.143975	**1.453675**	**2.635983**
Our System				
Recursive Fixed Exploration	Gemini-3.1-Pro	1.144047	1.456001	**2.635983**
DREAM-RSI	Gemini-3.1-Pro	**1.145427**	1.456375	**2.635983**

performance (Detailed PACEvolve scaling curves are provided in Appendix C.1.). Appendix E.2 gives the discovered ConvDiv kernel. Rather than optimizing convolution in isolation, it restructures the operator chain around downstream pooling and reduction. Adjacent convolution outputs are max-pooled immediately, while width pooling and subsequent reductions run directly on register-resident values, avoiding materialization of convolution and pooled feature maps. Linear operations, such as scaling, averaging, and bias accumulation, are deferred to a lightweight final reduction.

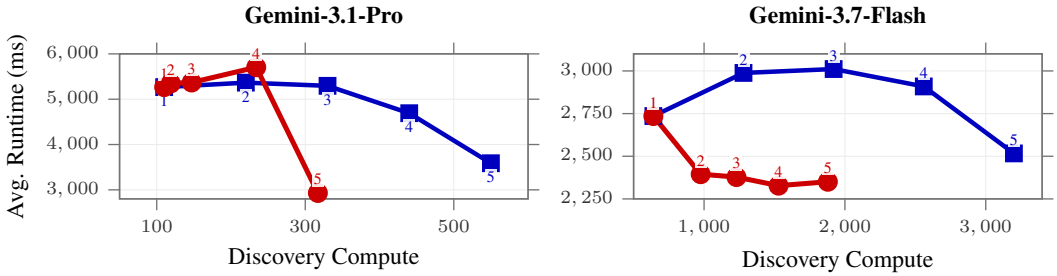
4.2 MATHEMATICS OPTIMIZATION

We further evaluate DREAM-RSI on three mathematical discovery tasks spanning discrete combinatorial optimization, geometric optimization, and functional optimization: the **Sum–Difference Problem**, **Circle Packing**, and **Autocorrelation Inequalities**. Formal definitions of the three tasks are provided in Appendix B.3. We use Gemini-3.1 Pro via the Gemini CLI as the discovery agent for both methods for 10 rounds. For each task, the agent iteratively proposes and evaluates candidate constructions or optimization procedures according to the task-specific objective.

Results. Table 1 summarizes the results. DREAM-RSI achieves a Sum–Difference score of 1.145427, outperforming SimpleTES and Recursive Fixed Exploration. On Circle Packing, it reaches 2.635983, matching the strongest reported result among the compared methods. For Autocorrelation, DREAM-RSI obtains 1.456375, remaining competitive with existing discovery systems.

Method	Model	Compute	*Non-biological*		*Biological*				Avg.
			Gisette	RCV1	DNA	Leukemia	Colon	Duke Breast	
Previous solvers									
sklearn	–	–	11275.2	252881.7	93.8	227.2	229.8	374.0	44180.3
glmnet	–	–	9063.6	73072.8	351.9	45.0	24.2	47.7	13767.5
SimpleTES	gpt-oss-120b	51,200	3141.9	19625.6	15.9	15.5	11.6	18.1	3804.8
SimpleTES †	gpt-oss-120b	51,200	8651.0	41143.1	37.6	28.2	19.5	31.1	8318.4
Our System									
Recursive Fixed Exploration	Gemini-3.1-Pro	550	1861.8	19550.1	41.5	26.1	14.5	28.4	3587.1
	Gemini-3.7-Flash	3200	1133.1	13873.0	29.8	24.1	15.7	24.4	2516.7
DREAM-RSI	Gemini-3.1-Pro	317	2841.0	14616.0	49.9	30.2	16.4	32.5	2931.0
	Gemini-3.7-Flash	1879	1091.9	12923.4	31.4	21.0	12.2	23.6	**2350.6**

(a) Final performance of running time (ms).



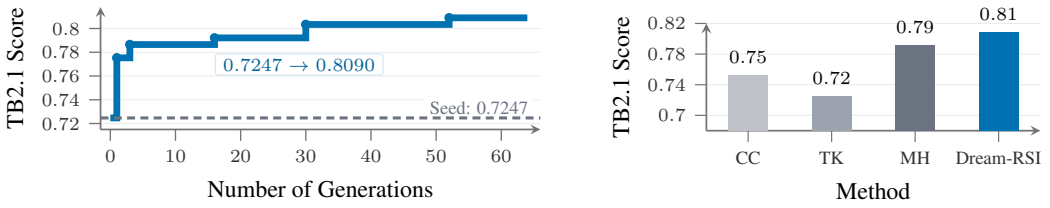
(b) Recursive Discovery Dynamics.

Figure 4: **Lasso regularization-path discovery. (a)** Final wall-clock runtime on six held-out tasks (lower is better). **Compute** is the cumulative number of discovery-agent calls. **(b) Recursive discovery dynamics.** Average runtime on the same six tasks versus cumulative discovery compute for Gemini 3.1-Pro/3.7-Flash; marker labels give the round. † denotes results reproduced by us.

4.3 ALGORITHM ENGINEERING

In this task, we consider **Lasso Regularization Path** as our algorithm-engineering task, a fundamental computational primitive in high-dimensional statistics that is widely used in model selection and cross-validation across domains such as genomics and finance. We follow the benchmark setting of SimpleTES (Ye et al., 2026), where the goal is to discover efficient implementations of the complete Lasso regularization path while preserving numerical correctness. During discovery, we use the same 17 synthetic instances as SimpleTES, which cover diverse problem regimes in terms of dimensionality, sparsity, feature correlation, and active-set structure. To evaluate whether the discovered algorithms generalize beyond the search distribution, we additionally evaluate them on six held-out downstream datasets spanning biological and non-biological domains. We compare against standard Lasso solvers **sklearn** (Pedregosa et al., 2011) and **glmnet** (Friedman et al., 2010), as well as **SimpleTES** (Ye et al., 2026). We run both controlled baseline and DREAM-RSI for 5 rounds.

Main Results. Figure 4(a) summarizes the Lasso discovery results. Across both discovery-agent backbones, DREAM-RSI achieves a better downstream quality–compute trade-off than Recursive Fixed Exploration. With Gemini-3.1 Pro, it reduces the average runtime across the six held-out datasets from 3587.1 ms to 2931.0 ms while using only 317 discovery-agent calls, compared with 550 calls for fixed exploration. With Gemini-3.7-Flash, DREAM-RSI further reduces the average runtime from 2516.7 ms to 2350.6 ms using 1879 calls instead of 3200. Despite using substantially less discovery compute, the resulting solvers also outperform the standard sklearn and glmnet implementations on all six held-out datasets. Relative to SimpleTES's reported 51,200 generations, our Pro and Flash runs use approximately 162× and 27× fewer discovery-agent calls, respectively, with lower average downstream runtime, although these comparisons use different model backbones. For 3.1 Pro, the average improvement over fixed exploration is driven by RCV1, despite slower runtimes on the other five datasets. For 3.7 Flash, runtime improves on five of the six datasets.



(a) Discovery Process **(b)** Detailed Comparison

Figure 5: Harness optimization results of DREAM-RSI. **(a)** Best-so-far TerminalBench2.1 score over the discovery process, improving the seed harness from 0.7247 to 0.8090. **(b)** Full-suite comparison against Claude Code (CC), Terminus-Kira (TK), and Meta-Harness (MH).

Recursive Discovery Dynamics. Figure 4(b) shows the trajectory of downstream performance across recursive discovery rounds against cumulative discovery compute. By design, both methods behave identically in the first round. Afterward, Recursive Fixed Exploration keeps a static exploration policy, while DREAM-RSI iteratively refines and redeploys its policy by dreaming over accumulated discovery history. By the final round, DREAM-RSI achieves lower average downstream runtime with fewer discovery-agent calls than Recursive Fixed Exploration on both backbones. We further examine the discovered solver, whose implementation appears in Appendix E.1. Unlike SIMPLETES, which switches between LARS and coordinate descent by problem dimension, the discovered solver makes the active-set optimization itself adaptive. It pairs strong-rule screening with Cauchy–Schwarz KKT pruning, computing exact gradients only when necessary and falling back to a full refresh once pruning loses effectiveness.

4.4 HARNESS OPTIMIZATION

We further evaluate DREAM-RSI on the *harness optimization* task, where the goal is to improve the program surrounding a fixed language model, including how information is stored, retrieved, and presented during agent execution. We follow the setting of Meta-Harness (Lee et al., 2026) and initialize the discovery from Terminus-Kira as the seed harness. Each discovered harness is evaluated on the full Terminal-Bench 2.1 (Merrill et al., 2026) suite of 89 tasks, with two independent runs per task. We use Claude Opus 4.8 with Claude Code as both the discovery agent and the policy-development agent, while harness evaluation is also performed with Claude Opus 4.8. We reproduce Meta-Harness following its original search protocol, using Opus 4.8 for both search and evaluation.

Results. Figure 5 shows that DREAM-RSI improves the harness throughout the discovery process, increasing the full-suite score from 0.7247 to 0.8090. The discovered harness also outperforms Claude Code (0.75) and Meta-Harness (0.79) under the same evaluation setting.

5 FURTHER ANALYSIS

Analysis of Replay World Accumulation We study the effect of accumulating replay worlds across recursive rounds. Table 2 shows that DREAM-RSI achieves a performance of 1.90 using the full discovery history, compared with 1.59 when replay is restricted to the current round. Accumulating replay worlds may help learn a more general exploration policy and avoid overly conservative exploration associated with fitting to a single world.

Table 2: Effect of accumulated discovery history on CONVDIV.

Method	Performance ↑	Compute ↓
DREAM-RSI (All History)	**1.90**	786
Current History Only	1.59	430

Analysis of Historical Inductive Biases in Long-Horizon Discovery We further investigate how the nature of the historical inductive bias affects long-horizon discovery. A natural alternative for utilizing history is to abstract prior trajectories into high-level directional insights, which are directly injected into the prompt as explicit semantic guidance for subsequent rounds. To evaluate the efficacy of this prompt-level semantic guidance, we apply it to both Recursive Fixed Exploration and DREAM-RSI. As illustrated in Figure 6, explicit directional guidance yields lower final performance for both methods on ConvDiv. This suggests that such guidance may constrain exploration.

6 Related Work

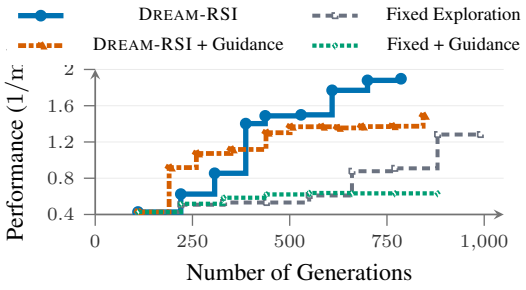


Figure 6: ConvDiv discovery results. Replaying history beats simply using it as guidance.

AI-Driven Scientific and Algorithmic Discovery. LLM-based discovery systems iteratively generate, evaluate, and refine candidate solutions using prior artifacts and feedback, as in AlphaEvolve (Novikov et al., 2025), OpenEvolve (Sharma, 2025), CodeEvolve (Assumpção et al., 2025), ShinkaEvolve (Lange et al., 2026), PACEvolve (Yan et al., 2026b), DeltaEvolve (Jiang et al., 2026) and MLEvolve Du et al. (2026). More recent work emphasizes the importance of exploration itself: SkyDiscover provides adaptive discovery infrastructure (Liu et al., 2026b), SwarmResearch dynamically orchestrates multiple search branches (Virk et al., 2026), and EvoX (Liu et al., 2026a) explicitly optimizes search strategies rather than only candidate solutions. This shift makes exploration a meta-level optimization problem, but useful supervision for exploration strategies is expensive and delayed because their quality often becomes apparent only after long discovery rollouts.

Self-Evolving Agents. A broader line of work studies agents that improve their own components during interaction. Prior methods evolve model weights (Huang et al., 2026c;a), agent harnesses (Lee et al., 2026; Zhang et al., 2026b), contexts (Zhang et al., 2026d), skills (Zhang et al., 2026a; Ouyang et al., 2026a; Wu et al., 2026b), model behavior through test-time learning (Wang et al., 2025; Yuksekgonul et al., 2026; Yan et al., 2026a; Wu et al., 2026a), rubrics (Xiong et al., 2026), environments (Huang et al., 2026b) and other applications (Dai et al., 2026). Most operate at the *object level*, improving components used for task execution or reasoning. Recent work has begun to optimize meta-level mechanisms, including search strategies and self-improvement procedures (Liu et al., 2026a; Yan et al., 2026a; Wang et al., 2026; Zhang et al., 2026c; Kim et al., 2026). However, such meta-level strategies are difficult to improve because their quality is often revealed only after costly long-horizon rollouts. DREAM-RSI turns accumulated discovery history into replay simulators for off-policy evaluation and refinement of exploration policies, then redeploys the selected policy for new online discovery, closing a recursive self-improvement loop.

Memory, History, and Experience Reuse. Prior work reuses agent experience as search history, context, memory, reusable skills, or training signals. DeltaEvolve structures evolutionary history through semantic deltas (Jiang et al., 2026); SwarmResearch and MLEvolve use cross-branch or retrospective information to guide subsequent search (Virk et al., 2026; Du et al., 2026); and other work improves how agents access and retain experience through evolving contexts, broader harness state, libraries, or skills (Zhang et al., 2026d; Lee et al., 2026; Xu et al., 2026; Ouyang et al., 2026a). We take a different view: rather than using exploration history only as context or memory for the next decision, we organize it as a *replay simulator* in which many alternative exploration controllers can be evaluated cheaply. This turns previously collected discovery experience into reusable feedback for meta-level optimization, alleviating the scarcity and high cost of training signals.

7 Conclusion

We presented DREAM-RSI, a framework for the recursive self-improvement of exploration policies in long-horizon discovery. The central idea in DREAM-RSI is to convert accumulated discovery history from static context into an active, replayable simulator, directly targeting the core bottleneck of meta-optimization: delayed and expensive feedback. By ”dreaming” within replay simulators constructed from historical discovery trees, DREAM-RSI evaluates candidate exploration policies rapidly and at negligible execution cost. Improved policies are then redeployed online, driving further discovery and expanding the simulator pool — closing the recursive self-improvement loop. Our experiments demonstrate competitive discovery quality across diverse domains and improved discovery-agent-call efficiency in several settings.

REFERENCES

Anthropic. Learning more about claude's mathematical capabilities. https://www.anthropic.com/research/riemann-zeta, August 2026. Accessed: 2026-08-13.

Henrique Assumpção, Diego Ferreira, Leandro Campos, and Fabricio Murai. Codeevolve: an open source evolutionary coding agent for algorithmic discovery and optimization. *arXiv preprint arXiv:2510.14150*, 2025.

Shiyi Cao, Ziming Mao, Joseph E Gonzalez, and Ion Stoica. K-search: Llm kernel generation via co-evolving intrinsic world model. *arXiv preprint arXiv:2602.19128*, 2026.

Runpeng Dai, Kaili Huang, Changsung Kang, and Ciya Liao. It takes two to match: Co-evolving generative retriever with reinforcement learning. *arXiv preprint arXiv:2609.00638*, 2026.

Shangheng Du, Xiangchao Yan, Jinxin Shi, Zongsheng Cao, Shiyang Feng, Zichen Liang, Boyuan Sun, Tianshuo Peng, Yifan Zhou, Xin Li, et al. Mlevolve: A self-evolving framework for automated machine learning algorithm discovery. *arXiv preprint arXiv:2606.06473*, 2026.

Jerome H Friedman, Trevor Hastie, and Rob Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of statistical software*, 33:1–22, 2010.

Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. Mathematical exploration and discovery at scale. *arXiv preprint arXiv:2511.02864*, 2025.

Saurabh Gupta, James Davidson, Sergey Levine, Rahul Sukthankar, and Jitendra Malik. Cognitive mapping and planning for visual navigation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2616–2625, 2017.

David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2(3):440, 2018.

Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019.

Danijar Hafner, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering atari with discrete world models. *arXiv preprint arXiv:2010.02193*, 2020.

Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023.

Danijar Hafner, Wilson Yan, and Timothy Lillicrap. Training agents inside of scalable world models. *arXiv preprint arXiv:2509.24527*, 2025.

Yuyang Hu, Shichun Liu, Yanwei Yue, Guibin Zhang, Boyang Liu, Fangyi Zhu, Jiahang Lin, Honglin Guo, Shihan Dou, Zhiheng Xi, et al. Memory in the age of ai agents. *arXiv preprint arXiv:2512.13564*, 2025.

Chengsong Huang, Haolin Liu, Tong Zheng, Runpeng Dai, Langlin Huang, Jinyuan Li, Zongxia Li, Zhepei Wei, Yu Meng, and Jiaxin Huang. G-zero: Self-play for open-ended generation from zero data. *arXiv preprint arXiv:2605.09959*, 2026a.

Chengsong Huang, Zifeng Wang, Rujun Han, Jun Yan, Yanfei Chen, Zoey CuiZhu, Ke Jiang, Peng Xia, Han Yu, Yufan Zhuang, et al. Envharness: Awakening static worlds for agent learning. *arXiv preprint arXiv:2608.19880*, 2026b.

Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. R-zero: Self-evolving reasoning llm from zero data. In *International Conference on Learning Representations*, volume 2026, pp. 130770–130790, 2026c.

Jaber Jaber and Osama Jaber. Autokernel: Autonomous gpu kernel optimization via iterative agent-driven search. *arXiv preprint arXiv:2603.21331*, 2026.

Jiachen Jiang, Tianyu Ding, and Zhihui Zhu. Deltaevolve: Accelerating scientific discovery through momentum-driven evolution. *arXiv preprint arXiv:2602.02919*, 2026.

Zae Myung Kim, Young-Jun Lee, Seungyeon Jwa, and Dongyeop Kang. Meta^n: Recursive self-improvement through emergent depth. *arXiv preprint arXiv:2608.24735*, 2026.

Robert Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. In *International Conference on Learning Representations*, volume 2026, pp. 74026–74078, 2026.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z Pan, et al. Evox: Meta-evolution for automated discovery. *arXiv preprint arXiv:2602.23413*, 2026a.

Shu Liu, Mert Cemri, Shubham Agarwal, Alexander Krentsel, Ashwin Naren, Qiuyang Mang, Zhifei Li, Akshat Gupta, Monishwaran Maheswaran, Audrey Cheng, Melissa Pan, Ethan Boneh, Kannan Ramchandran, Koushik Sen, Matei Zaharia, Alexandros G. Dimakis, and Ion Stoica. Skydiscover: A flexible, adaptive framework for ai-driven scientific and algorithmic discovery. In *Proceedings of the ACM Conference on AI and Agentic Systems*, CAIS '26, pp. 1223–1227. Association for Computing Machinery, 2026b. doi: 10.1145/3786335.3813221. URL https://doi.org/10.1145/3786335.3813221.

Shuaiqi Liu, Zhengkai Lin, Yuxiang Zhang, Yuanyi Ren, Yue Wu, Yongbin Li, Zheng Wang, Zhihang Fu, and Jieping Ye. The path to recursive self-improving agents: Foundation, framework, and future directions. *Preprints*, August 2026c. doi: 10.20944/preprints202608.0051.v1. URL https://doi.org/10.20944/preprints202608.0051.v1.

Thomas M. Moerland, Joost Broekens, Aske Plaat, and Catholijn M. Jonker. Model-based reinforcement learning: A survey. *Foundations and Trends in Machine Learning*, 16(1):1–118, 2023.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

OpenAI. On the navier–stokes millennium prize problem. https://openai.com/index/navier-stokes-solution/, September 2026. Accessed: 2026-09-10.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Siru Ouyang, Jun Yan, Yanfei Chen, Rujun Han, Zifeng Wang, Bhavana Dalvi Mishra, Rui Meng, Chun-Liang Li, Yizhu Jiao, Kaiwen Zha, et al. Skillos: Learning skill curation for self-evolving agents. *arXiv preprint arXiv:2605.06614*, 2026a.

Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, et al. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, volume 2026, pp. 94327–94354, 2026b.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Asankhaya Sharma. Openevolve: an open-source evolutionary coding agent, 2025. URL https://github.com/algorithmicsuperintelligence/openevolve.

Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In Bruce Porter and Raymond Mooney (eds.), *Machine Learning Proceedings 1990*, pp. 216–224. Morgan Kaufmann, San Francisco (CA), 1990. ISBN 978-1-55860-141-3. doi: https://doi.org/10.1016/B978-1-55860-141-3.50030-4. URL https://www.sciencedirect.com/science/article/pii/B9781558601413500304.

Yuvraj Virk, Zack Edds, Chunqiu Steven Xia, and Lingming Zhang. Swarmresearch: Orchestrating coding agents for open-ended discovery. *arXiv preprint arXiv:2607.02807*, 2026.

Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, Xinyu Yang, Zeyi Huang, Xuehai He, Luyao Ma, Baolin Peng, et al. Thetaevolve: Test-time learning on open problems. *arXiv preprint arXiv:2511.23473*, 2025.

Zefeng Wang, Minxi Yan, Jinhe Bi, Sikuan Yan, Volker Tresp, and Yunpu Ma. Metaskill-evolve: Recursive self-improvement of llm agents via two-timescale meta-skill evolution. *arXiv preprint arXiv:2607.05297*, 2026.

Shujin Wu, Cheng Qian, Xiusi Chen, and Heng Ji. Teaching llms to self-evolve: Cultivating core meta-skills with reinforcement learning. *arXiv preprint arXiv:2607.21971*, 2026a.

Xidong Wu, Yue Zhuan, Ruoqiao Wei, Hangxin Chen, Di Bai, Jintao Liu, Xinyi Wang, Xue Wang, Luoshu Wang, and Xinwu Cheng. Agenticrectune: Multi-agent with self-evolving skillhub for recommendation system optimization. *arXiv preprint arXiv:2604.26969*, 2026b.

Tianyi Xiong, Zhengyuan Yang, Xiaofei Wang, Chung-Ching Lin, Ruichun Ma, Kevin Lin, Zhendong Wang, Linjie Li, Chenxi Liu, Ruibo Chen, et al. Rubrics as visual-repair context for self-evolving ui-to-code generation. *arXiv preprint arXiv:2608.24138*, 2026.

Weijia Xu, Alessandro Sordoni, Chandan Singh, Zelalem Gero, Michel Galley, Xingdi Yuan, and Jianfeng Gao. Test-time learning with an evolving library. *arXiv preprint arXiv:2605.14477*, 2026.

Minghao Yan, Bo Peng, Benjamin Coleman, Ziqi Chen, Zhouhang Xie, Shuo Chen, Zhankui He, Noveen Sachdeva, Weili Wang, Ed H Chi, et al. Pacevolve++: Improving test-time learning for evolutionary search agents. *arXiv preprint arXiv:2605.07039*, 2026a.

Minghao Yan, Bo Peng, Benjamin Coleman, Ziqi Chen, Zhouhang Xie, Shuo Chen, Zhankui He, Noveen Sachdeva, Isabella Ye, Weili Wang, et al. Pacevolve: Enabling long-horizon progress-aware consistent evolution. *arXiv preprint arXiv:2601.10657*, 2026b.

Haotian Ye, Haowei Lin, Jingyi Tang, Yizhen Luo, Caiyin Yang, Chang Su, Rahul Thapa, Rui Yang, Ruihua Liu, Zeyu Li, et al. Evaluation-driven scaling for scientific discovery. *arXiv preprint arXiv:2604.19341*, 2026.

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, et al. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*, 2026.

Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, et al. Coevoskills: Self-evolving agent skills via co-evolutionary verification. *arXiv preprint arXiv:2604.01687*, 2026a.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin gödel machine: open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, volume 2026, pp. 104223–104294, 2026b.

Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents. *arXiv preprint arXiv:2603.19461*, 2026c.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026d.

Tong Zheng, Haolin Liu, Chengsong Huang, Huiwen Bao, Sheng Zhang, Rui Liu, Runpeng Dai, Ruibo Chen, Chenxi Liu, Tianyi Xiong, et al. Llms improving llms: Agentic discovery for test-time scaling. *arXiv preprint arXiv:2605.08083*, 2026a.

Tong Zheng, Hongming Zhang, Wenhao Yu, Xiaoyang Wang, He Xing, Runpeng Dai, Rui Liu, Huiwen Bao, Chengsong Huang, Heng Huang, et al. Parallel-r1: Towards parallel thinking via reinforcement learning. In *International Conference on Learning Representations*, volume 2026, pp. 121144–121166, 2026b.

A AI USE STATEMENT

In this work, we used generative AI tools as the API in the experiments, as described in the paper: we use Gemini 3.1 Pro in the Section 4.1,4.2,4.3, Gemini 3.7 Flash in Section 4.3 and Claude Opus 4.8 with Claude Code in the Section4.4. Additionally, we used Gemini 3.8 Flash for writing and grammar correction. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

B DETAILED EXPERIMENTAL SETTINGS

B.1 EVALUATION ENVIRONMENTS

Lasso regularization-path discovery. We evaluate candidate solvers in a KVM virtual machine with an AMD EPYC 7B12 processor, 8 vCPUs, and 32 GB RAM, running Debian-based Linux. The evaluation protocol follows Ye et al. (2026).

GPU kernel engineering. We evaluate candidate kernels on an NVIDIA L40S GPU. Numerical correctness is checked against the reference PyTorch implementation following the official KernelBench protocol (Ouyang et al., 2025).

Mathematical optimization. We use the same hardware environment as for Lasso regularization-path discovery.

Harness optimization. We evaluate candidate harnesses on a CPU server with 96 physical cores (192 logical CPUs) and 708 GiB RAM, using Docker to provide isolated task execution environments.

B.2 DISCOVERY CONFIGURATION

Table 3 summarizes the discovery configurations. Each offline phase evaluates the current policy and three newly generated policy versions ($M = 4$). The parenthesized budget decomposition specifies the maximum number of branches and attempts per branch, including the initial attempt. These budgets are upper limits; actual discovery-agent calls depend on the policy's allocation and stopping decisions.

Table 3: Discovery configurations. T denotes the number of online discovery rounds, W the maximum number of parallel workers, B the maximum discovery-agent calls per online round, and M the number of candidate policies evaluated per offline phase, including the current policy.

Task	**Discovery Model**	$\boldsymbol{T}$	$\boldsymbol{W}$	$\boldsymbol{B}$	$\boldsymbol{M}$
Lasso	Gemini-3.1-Pro	5	10	110 (10×11)	4
Lasso	Gemini-3.7-Flash	5	32	640 (32×20)	4
VGG16	Gemini-3.1-Pro	9	10	110 (10×11)	4
LayerNorm	Gemini-3.1-Pro	9	10	110 (10×11)	4
ConvDiv	Gemini-3.1-Pro	9	10	110 (10×11)	4
ConvMax	Gemini-3.1-Pro	9	10	110 (10×11)	4
Sum–Difference	Gemini-3.1-Pro	10	10	110 (10×11)	4
Autocorrelation	Gemini-3.1-Pro	10	10	110 (10×11)	4
Circle Packing	Gemini-3.1-Pro	10	10	110 (10×11)	4
Harness Optimization	Claude Opus 4.8	9	4	16 (4×4)	4

B.3 Detailed Task Description

Problem 1 (Lasso Regularization Path)

Given a feature matrix $X \in \mathbb{R}^{n\times p}$, response $y \in \mathbb{R}^n$, and a decreasing sequence $\lambda_1 > \cdots > \lambda_K$, define

$$F_k(w) := \frac{1}{2n}\|y - Xw\|_2^2 + \lambda_k \|w\|_1, \qquad w_k^\star \in \arg\min_{w\in\mathbb{R}^p} F_k(w).$$

A candidate solver returns approximate coefficients $\widetilde{W} = (\widetilde{w}_1, \ldots, \widetilde{w}_K)$. It passes the benchmark's objective-value check if

$$F_k(\widetilde{w}_k) \le F_k(w_{k,\mathrm{sklearn}}) + 10^{-6}, \qquad \text{for every } k,$$

where correctness is checked on fresh instances distinct from the timing instances. If any required check fails, the search score is zero.
Otherwise, letting $\mathcal{I}$ denote the timing instances and t_i the time required to compute the complete regularization path on instance i, the search score is

$$R_{\mathrm{search}} = \left(\prod_{i\in\mathcal{I}} t_i\right)^{-1/|\mathcal{I}|}.$$

Problem 2 (Sum–Difference Problem)

The Sum–Difference Problem asks for a finite set $A \subset \mathbb{Z}$ whose normalized sumset is large relative to its normalized difference set. The objective is

$$\Gamma(A) := \frac{\log(|A+A|/|A|)}{\log(|A-A|/|A|)},$$

where

$$A + A := \{a + a' : a, a' \in A\}, \qquad A - A := \{a - a' : a, a' \in A\}.$$

The goal is to construct a finite set $A \subset \mathbb{Z}$ that maximizes $\Gamma(A)$.

Problem 3 (Circle Packing in a Unit Square)

For $n \in \{26, 32\}$, the circle-packing task asks for centers $(x_i, y_i) \in [0,1]^2$ and radii $r_i \ge 0$ such that every circle lies inside the unit square and no two circles overlap:

$$r_i \le x_i \le 1 - r_i, \qquad r_i \le y_i \le 1 - r_i,$$
$$(x_i - x_j)^2 + (y_i - y_j)^2 \ge (r_i + r_j)^2, \qquad 1 \le i < j \le n.$$

The objective is to maximize

$$\sum_i r_i.$$

Problem 4 (Autocorrelation Inequalities)

For an integrable function $f : \mathbb{R} \to \mathbb{R}$, define the autoconvolution

$$(f * f)(t) := \int_{\mathbb{R}} f(t-x)f(x)\,dx, \qquad t \in \left[-\frac{1}{2}, \frac{1}{2}\right].$$

First Autocorrelation Inequality. Find a non-negative integrable function $f : \mathbb{R} \to \mathbb{R}$ supported on $\left[-\frac{1}{4}, \frac{1}{4}\right]$ such that

$$\int_{-1/4}^{1/4} f(x)\,dx = 1.$$

The objective is to minimize

$$\Phi_1(f) := \max_{t \in [-1/2,\, 1/2]} (f * f)(t).$$

Second Autocorrelation Inequality. Find a non-negative integrable function $f : \mathbb{R} \to \mathbb{R}$ supported on $\left[-\frac{1}{4}, \frac{1}{4}\right]$ such that

$$\int_{-1/4}^{1/4} f(x)\,dx = 1.$$

The objective is to maximize

$$\Phi_2(f) := \frac{\|f * f\|_2^2}{\|f * f\|_1 \|f * f\|_\infty}.$$

Third Autocorrelation Inequality. Find an integrable (possibly signed) function $f : \mathbb{R} \to \mathbb{R}$ supported on $\left[-\frac{1}{4}, \frac{1}{4}\right]$ such that

$$\int_{-1/4}^{1/4} f(x)\,dx = 1.$$

The objective is to minimize

$$\Phi_3(f) := \max_{t \in [-1/2,\, 1/2]} \left|(f * f)(t)\right|.$$

Problem 5 (VGG16)

The VGG16 task asks for an efficient implementation of the forward pass of a VGG16 network. The input is

$$x \in \mathbb{R}^{10 \times 3 \times 224 \times 224},$$

and the network consists of five convolutional blocks. Each convolution uses a 3×3 kernel with stride 1 and padding 1, followed by a ReLU activation. The channel configurations of the five blocks are

$$(64, 64), \quad (128, 128), \quad (256, 256, 256), \quad (512, 512, 512), \quad (512, 512, 512),$$

with a 2×2 max-pooling operation of stride 2 after each block.
After the final pooling layer, the resulting $512 \times 7 \times 7$ feature map is flattened and passed through three fully connected layers,

$$512 \cdot 7 \cdot 7 \;\to\; 4096 \;\to\; 4096 \;\to\; 1000,$$

with ReLU activations after the first two layers. Dropout is disabled ($p = 0$). The final output is

$$y \in \mathbb{R}^{10 \times 1000}.$$

The goal is to produce an implementation numerically equivalent to the reference PyTorch computation while minimizing GPU execution time.

Problem 6 (ConvDiv)

The ConvDiv task applies a sequence of 3D convolution, scaling, pooling, and reduction operations. The input is

$$x \in \mathbb{R}^{128\times8\times16\times64\times64}.$$

First, a 3D convolution with 8 input channels, 16 output channels, and a $3\times3\times3$ kernel is applied without padding:

$$z = \mathrm{Conv3D}(x) \in \mathbb{R}^{128\times16\times14\times62\times62}.$$

The convolution output is divided by 2 and then passed through $2 \times 2 \times 2$ max pooling:

$$u = \mathrm{MaxPool}_{2\times2\times2}\left(\frac{z}{2}\right) \in \mathbb{R}^{128\times16\times7\times31\times31}.$$

Global average pooling is subsequently applied over the three spatial dimensions,

$$v_{b,c} = \frac{1}{7\cdot31\cdot31}\sum_{d,h,w} u_{b,c,d,h,w},$$

giving

$$v \in \mathbb{R}^{128\times16\times1\times1\times1}.$$

A learnable channel-wise bias

$$\beta \in \mathbb{R}^{16\times1\times1\times1}$$

is then added, followed by summation over the channel dimension:

$$y_b = \sum_{c=1}^{16}\left(v_{b,c} + \beta_c\right).$$

The output has shape

$$y \in \mathbb{R}^{128\times1\times1\times1}.$$

The goal is to compute this operator chain exactly up to the benchmark's numerical tolerance while minimizing GPU execution time.

Problem 7 (ConvMax)

The ConvMax task consists of a 3D convolution followed by max pooling, channel-wise log-sum-exp reduction, and ReLU. The input is

$$x \in \mathbb{R}^{4\times32\times32\times128\times128}.$$

A 3D convolution maps 32 input channels to 64 output channels using a $3 \times 3 \times 3$ kernel with stride 1 and padding 1:

$$z = \mathrm{Conv3D}(x) \in \mathbb{R}^{4\times64\times32\times128\times128}.$$

This is followed by $2 \times 2 \times 2$ max pooling with stride 2,

$$u = \mathrm{MaxPool}_{2\times2\times2}(z) \in \mathbb{R}^{4\times64\times16\times64\times64}.$$

The channel dimension is then reduced using log-sum-exp:

$$s_{b,1,d,h,w} = \log\left(\sum_{c=1}^{64}\exp(u_{b,c,d,h,w})\right),$$

and the final output is

$$y = \mathrm{ReLU}(s) \in \mathbb{R}^{4\times1\times16\times64\times64}.$$

The goal is to produce an implementation numerically equivalent to the reference computation while minimizing GPU execution time.

Problem 8 (Layer Normalization)

The LayerNorm task applies layer normalization to an input tensor

$$x \in \mathbb{R}^{16\times64\times256\times256}.$$

Normalization is performed independently for each batch element over the remaining three dimensions. Let

$$M = 64 \cdot 256 \cdot 256.$$

For each batch element b, define

$$\mu_b = \frac{1}{M} \sum_{c,h,w} x_{b,c,h,w},$$

and

$$\sigma_b^2 = \frac{1}{M} \sum_{c,h,w} \left(x_{b,c,h,w} - \mu_b\right)^2.$$

The normalized output is

$$y_{b,c,h,w} = \gamma_{c,h,w} \frac{x_{b,c,h,w} - \mu_b}{\sqrt{\sigma_b^2 + \epsilon}} + \beta_{c,h,w},$$

where

$$\gamma, \beta \in \mathbb{R}^{64\times256\times256}$$

are learnable affine parameters and $\epsilon = 10^{-5}$.
The output has the same shape as the input,

$$y \in \mathbb{R}^{16\times64\times256\times256}.$$

The goal is to match the reference LayerNorm computation within the benchmark's numerical tolerance while minimizing GPU execution time.

C Additional Results

C.1 Results of PACEvolve (Yan et al., 2026b) on KernelBench

We present the detailed discovery process of PACEvolve (Yan et al., 2026b) on KernelBench in Figure 7. For a controlled comparison, we allow PACEvolve a 48-hour wall-clock budget on each task and report its best-so-far performance throughout the search. In contrast, DREAM-RSI typically completes its discovery within 10–20 hours.

C.2 Analysis of Evolution of Exploration Behavior

Figure 8 illustrates how the learned exploration policy evolves across recursive rounds on ConvDiv. The exploration policy exhibits an adaptive pattern: as performance improves, it initially conserves discovery compute (e.g., reducing the number of evaluated attempts from 110 to 50). When progress subsequently plateaus, it increases exploration effort again, coinciding with further performance gains.

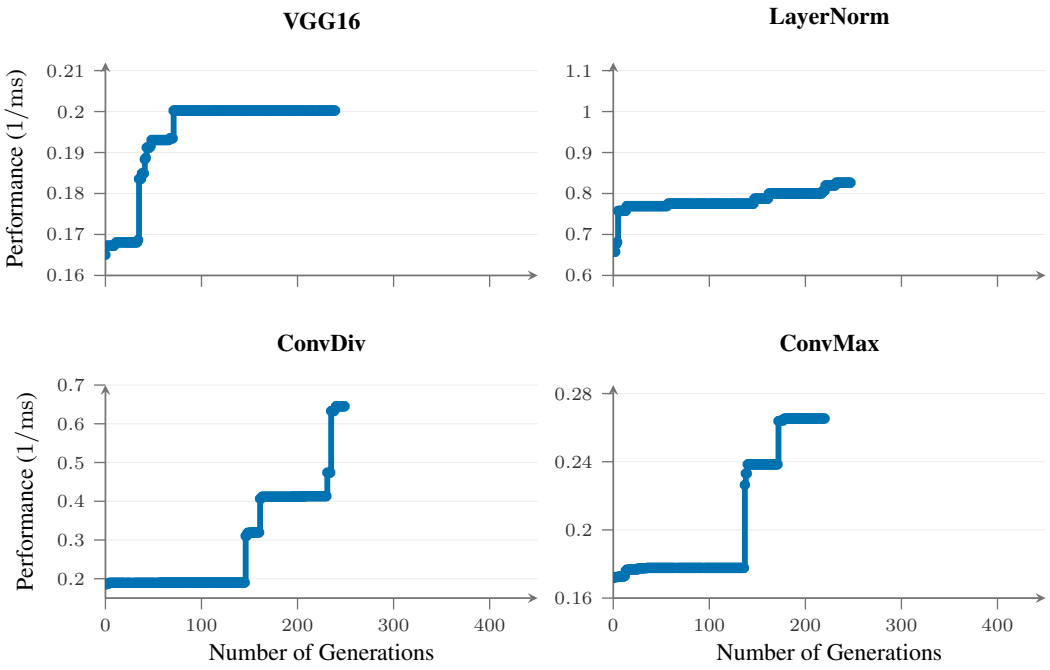


Figure 7: Evolution of the best-so-far kernel performance discovered by PACEvolve on four KernelBench tasks under a 48-hour wall-clock budget. Performance is measured as inverse runtime $(1/\mathrm{ms})$, with higher values indicating faster kernels.

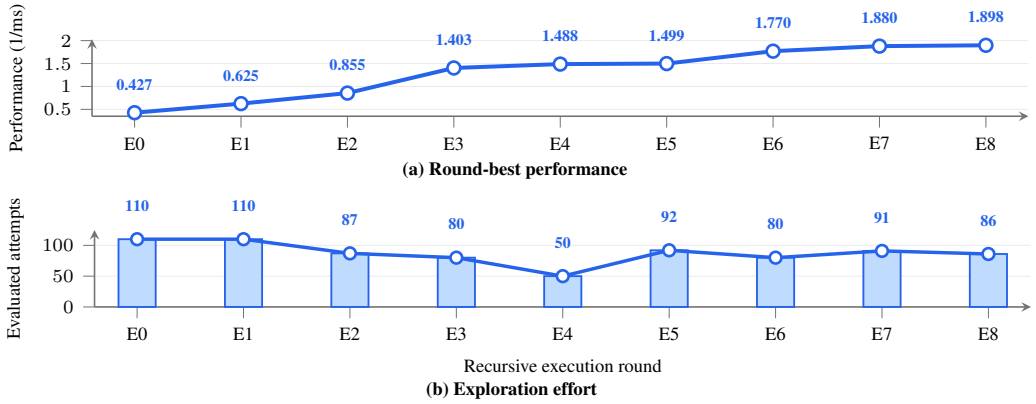


Figure 8: **Evolution of exploration behavior on ConvDiv. (a)** Round-best performance across recursive execution rounds. **(b)** The number of evaluated attempts in each round.

D PROMPTS

For reproducibility, we provide the prompts used for online exploration and replay-based exploration-policy improvement. Variables enclosed by dollar signs or braces are instantiated by the runtime system before execution.

D.1 EXPLORATION PROMPT

The following prompt is used to guide the discovery agent during online exploration. It requires the agent to inspect the complete available discovery history before proposing a new solution, explicitly reason about both successful and failed attempts, and avoid repeatedly exploiting a locally saturated direction.

Listing 1: Prompt used for online exploration.

```
You must read every historical proposal before proposing or implementing a new solution.

$direction_guidance

Variables (`$node_dir`, `$history_dir`, `$baseline_dir`, `$eval_program`, `$problem_file`)
     are filled in by the calling system. `$node_dir` is your own attempt directory ---
     exclude it when scanning sibling `attempt_*/` dirs.

## 1. Read the complete history first

Before proposing anything, read every `proposal.md` under sibling `attempt_*/` dirs, `
     $history_dir`, and `$baseline_dir` in full --- not a sample, not just recent cycles or
     the current branch. For each, read its matching `eval/score.json` (and `error.txt` if
     it failed). Trust the measured result over what the proposal claims about itself.

## 2. Learn from both successes and failures

For every past attempt, note the mechanism and how it did. For failures, figure out *why*: a
      flawed core idea, or a good idea let down by a bug, bad parameters, or an
     implementation slip? Don't repeat the former. The latter is worth retrying --- but only
      once you've actually located the bug in the code (not just guessed from the proposal),
      and only with a specific fix in hand.

## 3. Don't converge into a local optimum

Look at the shape of what's been tried. If most attempts cluster around small variations of
     one mechanism with flattening returns, that's a local optimum - resist proposing
     another small tweak there. Deliberately favor a structurally different mechanism or an
     untried combination over a safer marginal refinement. Exploration diversity matters as
     much as the next incremental gain.

## 4. Propose and implement

The new idea must be a genuinely new mechanism, a new combination of previously-successful
     pieces, or a targeted fix to a specific bug found in step 2 - never a repeat or rename
     of something already tried. Implement it in `$eval_program`. Don't claim it compiles,
     is correct, or beats SOTA until it's actually evaluated.

## Files

Write only `$node_dir/proposal.md` (mechanism, evidence from history, why it's not a repeat,
      expected benefit/risk) and `$node_dir/$eval_program`. Everything else is read-only.

## Note:
    Never execute pkill, kill, killall, or terminate unrelated processes.
```

D.2 Replay-Based Policy Improvement Prompt

The following prompt is used by the controller-development agent during historical replay. The agent modifies the exploration policy using feedback obtained from replay over previously collected discovery trajectories while remaining restricted to prefix-observable information.

Listing 2: Prompt used for replay-based improvement of the exploration policy.

```
You are improving one **prefix-only exploration policy**. Edit only
``{method_file}`` and implement ``OptimalPolicy.solve(self, question, budget=None)``.
Do not solve the scientific task and do not edit any other program.

## Objective: quality, work, and parallelism

The environment is a frozen, irregular branch×attempt grid. A policy opens a root
or refines the next cell of an already-open branch. Each revealed cell costs one
probe. The policy sees only the cells it has revealed so far; unrevealed scores are
unknown.

The evaluator sweeps your single ``beta`` knob and ranks the resulting curve by:

    pareto.reward = pareto.auc - lambda * parallel_penalty

``pareto.auc`` rewards reaching high per-trace attainment with few **total probes**.
``parallel_penalty`` is the mean of
``effective_sequential_rounds / total_probes`` over the sweep. For a batch of size
``k`` with ``W = question.max_parallelism`` workers, it costs one decision round and
``ceil(k / W)`` effective sequential rounds. A serial policy has penalty near 1;
useful full batches approach ``1/W``. Therefore choose only promising probes, but
batch independent promising probes whenever possible.
```

```
A local implementation failure does not by itself prove that its parent direction
is poor. Weigh recovery value against new roots and ordinary refinements while
keeping batches parallel.

## API

    question.reset()
    question.observed() -> dict[str, Observation]   # revealed prefix only
    question.legal_actions() -> list[str]           # roots + opened-branch frontiers
    question.legal_roots() -> list[str]             # unopened roots only
    question.opened_branches() -> list[int]
    question.meta(cell_id) -> CellMeta              # .branch .attempt .parent_id .seq .tags
    question.probe_batch(cells, on_reveal=...) -> list[Observation]
    question.baseline_score
    question.max_parallelism

``Observation`` supplies ``branch``, ``attempt``, ``score``, ``evaluated``, ``valid``,
``fail_class``, ``error``, ``delta_vs_baseline``, ``delta_vs_parent``, ``n_valid``, and
``n_total``.
Use the helpers in ``see.policy.observation_signal`` when useful:
``branch_promising``, ``branch_failed_hard``, ``probe_improved_vs_parent``, and
``probe_improved_vs_baseline``.

**Success semantics:** an evaluated observation with ``error is None`` and
``fail_class == "ok"`` is a successful evaluation, even when ``valid == False`` or
``n_valid``/``n_total`` are unavailable. Never label it repairable solely because
``valid`` is false. A *successful anchor* below means the best historical score
from such a successful evaluation.

Do **not** use ``question.best_so_far`` or ``question.budget_spent`` to decide what
to explore; they are bookkeeping only. Derive any decision statistic from
``question.observed()`` instead.

## Required branch trajectory and failure interpretation

For each opened branch, reconstruct its ordered prefix trajectory, not only its
latest observation or best score: successful anchor, score trend, regressions,
failure/repair sequence, and explored versus remaining depth.

Before closing or deprioritizing a failed frontier, classify it as
hard-unrecoverable, repairable implementation failure, weak-but-underexplored, or
repeatedly unpromising after sufficient valid evidence. Output/correctness mismatch,
shared-memory/resource limits, and variable/code, mask/layout/shape errors are
normally repairable. Do not infer algorithmic failure from one such error.
``n_valid == 0`` and ``branch_failed_hard(obs)`` are signals, not unconditional
closure: use ``fail_class`` and ``error`` to distinguish a repairable zero-valid
failure from an environment/dependency failure. ``compile_other`` alone is not
permanently hard. Classify the current failure episode: a later successful result
reopens the branch and cancels closure based only on an earlier failure.

## Required batch decision loop

At each decision round:

1. Read the prefix, reconstruct trajectories, and close only branches with
   cumulative evidence of being hard-unrecoverable or repeatedly unpromising.
2. Rank legal roots and legal branch frontiers using only prefix-derived signals:
   successful anchor, parent→child gain, complete branch trajectory, actual success
   versus failure evidence,
   failure recoverability, prior repair outcomes, remaining depth, and cross-branch
   comparison.
3. Rank actual repairable failures and underexplored frontiers in deterministic
   queues using trajectory, recoverability, remaining depth, repeated failures, and
   beta. A repairable failure retains eligibility unless cumulative evidence lowers
   its relative priority.
4. Build one **dynamic portfolio** batch of independent candidates, up to
   ``question.max_parallelism``: exploitation (strong normal refinements),
   exploration (new roots or underexplored branches), and at most one recovery
   (an actual repairable failure). When multiple roles are eligible, give
   exploration and justified recovery representation before filling remaining slots
   by priority; adapt this to prefix evidence rather than fixed quotas. Recovery
   must not displace normal successful refinements or leave workers idle. Never
   sample randomly, and do not default to a singleton merely because its top
   candidate is clear.
5. Stop only after considering the whole revealed portfolio: active, underexplored,
   recoverable, unopened, and remaining legal candidates. Do not stop while an
   eligible high-priority recovery or underexplored candidate remains; every
   remaining action needs an evidence-based decision to continue, reserve, or close.

A batch must contain distinct cells that are all legal *before* the call. It may
```

```
contain several roots and/or one frontier from each opened branch. It must never
contain a parent and its child together. Do not use a fixed widen-all / deepen-all
wave schedule: adapt batch composition after every revealed prefix.

Minimal structure:

    from see.policy.api import (
        LLMDesignedMethod, SimResult, _budget_done, _record_curve, finalize_result,
    )

    def solve(self, question, budget=None):
        question.reset()
        res, closed = SimResult(), set()
        while not _budget_done(question, budget):
            prefix = question.observed()
            update_closed(closed, prefix, question)
            batch = select_batch(prefix, question, closed)
            if not batch:
                break
            question.probe_batch(
                batch,
                on_reveal=lambda _: _record_curve(res, question),
            )
        return finalize_result(question, res)

## Hard constraints

- Keep ``NAME = "OptimalPolicy"`` and implement
  ``class OptimalPolicy(LLMDesignedMethod)`` in ``{method_file}`` only.
- **Prefix-only:** decisions may use revealed observations, ``baseline_score``, legal
  sets, structural ``meta``, and helper signals. Never use unrevealed scores, a true
  optimum, hardcoded winning cell ids, absolute score targets, or internal trace data.
- Every prune, widen, deepen, batch, and stop decision must be explainable from the
  current prefix. Shallow weak scores are not enough to discard a branch: deeper
  attempts can recover. A repairable latest failure must not erase its historical
  successful anchor or by itself cause permanent starvation.
- Replay calls with ``budget=None``. Always terminate when no batch is selected; do
  not assume a budget cap exists.
- A selected batch must be legal, have no duplicate ids, and contain at most
  ``question.max_parallelism`` cells.

## Beta: fixed per run, adaptive across cycles

Read exactly one scalar in ``__init__``:

    beta = float(self.config.get("beta", <sensible_default>))

Beta has three distinct roles. Do not conflate them:

1. **Within one replay or live episode:** beta is fixed. Route every behavioral
   threshold through one ``_schedule(beta) -> dict``. High beta means more width,
   deeper patience, and weaker pruning. Low beta means fewer probes, earlier
   stagnation stops, and stronger pruning. Never change beta from observations inside
   ``solve()``. Route recovery eligibility, reserve threshold, and waiting through
   the same schedule: high beta is more patient; low beta remains selective without
   treating one repairable failure as automatic closure.
2. **During offline evaluation:** eval sweeps a fixed beta grid. This measures whether
   the policy exposes a real attainment/work/parallelism trade-off; it is not online
   beta adaptation.
3. **When proposing the next policy version:** choose the baked-in default beta once,
   using evidence from earlier *live* cycles and their beta sweeps. That default will
   remain fixed throughout the next live exploration episode.

Keep all thresholds relative to the prefix; never use absolute score cutoffs.

Use the following cross-cycle default-beta rule. Read the most recent 2--3
**live** ``trace_pool/iter*/live_cycle_manifest.json`` sidecars (and ``_current``
when present) for each iteration's final best score and actual baked-in beta. Read
the matching archived ``beta_sweep.json`` values (``pareto.reward``, AUC, parallel
penalty, and the per-beta frontier). Scores alone do not establish that beta caused a
change, so always use both sources:

- live best is still improving: keep the prior default beta unless its sweep clearly
  shows a better nearby beta;
- live best has plateaued, and higher beta reaches higher attainment for a reasonable
  work/parallelism cost in the sweep: raise the default by a small step (about
  0.1--0.2, clamped to [0, 1]);
- a high default beta has already been tried through a plateau, and high-beta sweep
  points add work without higher attainment: lower it by a small step;
- history is insufficient or evidence conflicts: use a moderately exploratory default
  (about 0.6), rather than pretending the replay ceiling is a live stopping signal.
```

```
The beta sweep is non-degenerate only if beta changes the attainment/work trade-off.
It also reveals whether the policy batches. Do not select the default simply as the
smallest beta that reaches a frozen trace's known ceiling.

## Required next-cycle grid planning

Every proposed policy **must** implement this deterministic method:

    from see.policy.api import GridPlan, GridPlanningContext

    def plan_grid(self, context: GridPlanningContext) -> GridPlan:
        ...

This method runs **before** a new live grid is created. It does not make a
within-episode decision and must never inspect a current episode's outcomes.
It must always return a non-``None`` ``GridPlan``: do not inherit the template
stub and do not delegate grid choice to the runner's fallback. When history is
empty or insufficient, still return an explicit conservative bootstrap plan
derived from the context's fallback/hard-cap fields, with a factual reason.

``GridPlan(branch_count=W, refine_count=R)`` accepts arbitrary integers, not a
fixed set of presets. It creates branches ``0..W-1`` and attempts ``0..R``; ``R`` is
the number of refinements allowed after each root. The runner validates
``1 <= W <= context.hard_max_branch_count`` and
``0 <= R <= context.hard_max_refine_count``. In replay, a requested plan beyond the
frozen trace's ``context.trace_branch_count`` or ``context.trace_refine_count`` is
out of support and cannot earn replay reward.

Use only the prefix-safe facts in ``context``:

- ``history``: completed earlier live manifests, including prior planned/effective
  grids, actual opened width/depth, probe work, decision rounds, scores, and beta;
- fallback/hard caps and worker cap;
- replay structural support fields. Do not read raw trace outcomes or a current
  cycle result inside ``plan_grid``.

Choose width versus depth from evidence, not a default preference:

- many semantically distinct roots improve early while deeper refinements stall:
  increase width and reduce/hold depth;
- high gains arrive late on a small, repeatable set of directions: reduce/hold width
  and increase depth;
- all explored directions plateau after sufficient depth while meaningful direction
  classes remain uncovered: increase width;
- repeated hard, unrecoverable failures or strongly redundant directions: reduce
  width and depth conservatively;
- conflicting or insufficient history: return an explicit conservative bootstrap
  plan derived from the context, and state that evidence is insufficient.

Include a short, factual ``reason`` in every plan. ``plan_grid`` answers
how many directions to make available; the direction provider assigns those new
roots their directions, and ``solve`` still decides which legal roots/frontiers to
open, refine, prune, or stop. Do not choose roots merely because their branch id is
small. The runtime grid is the hard bound: controller thresholds may use less, but
can never create branches or attempts beyond the effective plan. Before finishing,
verify that the edited ``method.py`` contains an override of ``plan_grid`` that
returns ``GridPlan(branch_count=..., refine_count=..., reason=...)`` on every path.

## Learn from history without leaking outcomes

Earlier rounds are in ``{history_dir}/r####_*/``. Read their policy code and
``proposal_results/beta_sweep.json``. Start from a strong recent policy, retain
mechanisms that raised ``pareto.reward``, and make a concrete change when progress
stalls. A legacy AUC-only sweep is useful code history but is not numerically
comparable to the current reward. The baseline under ``{history_dir}/baseline/`` is
a parallel-refine floor to beat.

Each current-objective round also archives
``proposal_results/policy_execution_traces.jsonl``: one replay episode per
``(frozen trace, beta)``. Use it to diagnose general behavior --- serial batches,
premature stops, over-pruning, or wasted probes --- from the prefix state, selected
batch, and revealed outcomes at each decision round. It is **between-round feedback
only**: never read it inside ``solve()``, and never copy a trace-specific branch,
cell id, score, or target into policy logic.

``{trace_pool}``, if present, may be read only outside ``solve()``. Prefer the
``live_cycle_manifest.json`` sidecars over raw replay outcomes for the per-iteration
live trend. Never copy trace scores, targets, or cell ids into policy logic.

## Deliverable
```

```
Write a complete adaptive policy in ``{method_file}``. Include a short module
docstring describing its prefix signals, batch rule, beta schedule, default-beta
rationale, grid-planning rule (if implemented), and safeguards against
over-pruning, over-stopping, permanent starvation after repairable failures, and
serial probes. Before finishing, verify trajectory-based ranking, the stated
success semantics, non-automatic zero-valid closure, deterministic recovery
competition, and portfolio-level stop.
```

E DISCOVERED PROGRAMS

E.1 LASSO REGULARIZATION PATH

We provide the complete implementation of the Lasso-path solver discovered by DREAM-RSI. As discussed in Section 4.3, the solver combines strong-rule screening with adaptive Cauchy–Schwarz KKT pruning, disjoint active-set bookkeeping, lazy Gram-matrix construction, and hardware-aware optimizations.

Listing 3: Complete Lasso-path solver discovered by DREAM-RSI.

```
# EVOLVE-BLOCK-START

CPP_CODE = r'''
#define EIGEN_NO_DEBUG
#define EIGEN_MPL2_ONLY
#define EIGEN_UNROLL_LOOPS

#include <Eigen/Dense>
#include <vector>
#include <cstdio>
#include <cmath>
#include <algorithm>
#include <numeric>
#include <omp.h>
#include <cstdlib>
#include <cstring>

using Eigen::MatrixXd;
using Eigen::VectorXd;

#if defined(_MSC_VER)
#define RESTRICT __restrict
#elif defined(__GNUC__) || defined(__clang__)
#define RESTRICT __restrict__
#else
#define RESTRICT
#endif

// High-performance alignment assumption
#if defined(__GNUC__) || defined(__clang__)
#define ASSUME_ALIGNED(ptr, alignment) (double*)__builtin_assume_aligned((ptr), (alignment))
#else
#define ASSUME_ALIGNED(ptr, alignment) (ptr)
#endif

// High-performance branch-free soft-thresholding using std::abs and std::copysign
static inline double soft_thresh(double z, double gamma) {
    double abs_z = std::abs(z);
    double val = abs_z - gamma;
    return std::copysign(val > 0.0 ? val : 0.0, z);
}

// ============================================================================
// DISJOINT-PARTITION ACTIVE-SET LASSO PATH SOLVER WITH ALIGNED COLUMN PADDING
// ============================================================================
void solve_active_set(
    const double* RESTRICT X_padded,
    int n_padded,
    int n,
    int p,
    const VectorXd& y,
    const VectorXd& lam_path,
    const VectorXd& xv,
    const VectorXd& grad_init,
    MatrixXd&       coef_path,  // (p, n_lam) output, pre-zeroed
    double thresh,              // convergence threshold
```

```
    int    maxit)               // max inner loop iterations
{
    const double fn = static_cast<double>(n);
    const double inv_fn = 1.0 / fn;
    const double tol = thresh;
    const int nlam = lam_path.size();

    // Workload-Aware flag for activating Cauchy-Schwarz KKT Pruning
    const bool use_cs = (p >= 500 && n >= 150);

    // Initial capacity for active set structures - optimized to completely avoid reallocations on almost all problems
    int current_capacity = ((std::max(128, std::min(512, p)) + 7) / 8) * 8;

    // Declare raw pointers for 64-byte aligned structures
    double* G_data = nullptr;
    double* c_data = nullptr;
    double* beta_active_data = nullptr;
    double* xv_active_data = nullptr;
    double* inv_xv_active_data = nullptr;
    double* grad_init_active_data = nullptr;
    double* beta_old_at_start = nullptr;

    double* y_padded = nullptr;
    double* r_padded = nullptr;
    double* r_ref_padded = nullptr;

    bool oom = false;

    // Allocate 64-byte aligned arrays
    if (posix_memalign((void**)&G_data, 64, static_cast<size_t>(current_capacity) * current_capacity * sizeof(double)) != 0) goto cleanup;
    if (posix_memalign((void**)&c_data, 64, static_cast<size_t>(current_capacity) * sizeof(double)) != 0) goto cleanup;
    if (posix_memalign((void**)&beta_active_data, 64, static_cast<size_t>(current_capacity) * sizeof(double)) != 0) goto cleanup;
    if (posix_memalign((void**)&xv_active_data, 64, static_cast<size_t>(current_capacity) * sizeof(double)) != 0) goto cleanup;
    if (posix_memalign((void**)&inv_xv_active_data, 64, static_cast<size_t>(current_capacity) * sizeof(double)) != 0) goto cleanup;
    if (posix_memalign((void**)&grad_init_active_data, 64, static_cast<size_t>(current_capacity) * sizeof(double)) != 0) goto cleanup;
    if (posix_memalign((void**)&beta_old_at_start, 64, static_cast<size_t>(current_capacity) * sizeof(double)) != 0) goto cleanup;

    if (posix_memalign((void**)&y_padded, 64, static_cast<size_t>(n_padded) * sizeof(double)) != 0) goto cleanup;
    if (posix_memalign((void**)&r_padded, 64, static_cast<size_t>(n_padded) * sizeof(double)) != 0) goto cleanup;
    if (posix_memalign((void**)&r_ref_padded, 64, static_cast<size_t>(n_padded) * sizeof(double)) != 0) goto cleanup;

    std::fill(G_data, G_data + static_cast<size_t>(current_capacity) * current_capacity, 0.0);
    std::fill(c_data, c_data + current_capacity, 0.0);
    std::fill(beta_active_data, beta_active_data + current_capacity, 0.0);
    std::fill(xv_active_data, xv_active_data + current_capacity, 0.0);
    std::fill(inv_xv_active_data, inv_xv_active_data + current_capacity, 0.0);
    std::fill(grad_init_active_data, grad_init_active_data + current_capacity, 0.0);
    std::fill(beta_old_at_start, beta_old_at_start + current_capacity, 0.0);

    std::memcpy(y_padded, y.data(), n * sizeof(double));
    for (int i = n; i < n_padded; ++i) y_padded[i] = 0.0;

    std::memcpy(r_padded, y_padded, n_padded * sizeof(double));

    // Consistently initialize r_ref_padded to y_padded (instead of all zeros) to guarantee 100% tight bounds at start
    std::memcpy(r_ref_padded, y_padded, n_padded * sizeof(double));

    // Run the solver in a nested block to make goto compile-safe
    {
        VectorXd beta = VectorXd::Zero(p);

        std::vector<char> screened(p, 0);     // 1 if screened, 0 otherwise
        std::vector<int>  active;             // indices of active features (beta != 0)
        std::vector<int>  feat_to_idx(p, -1); // maps feature to index in active set

        // Disjoint tracking partition vectors
        std::vector<int> unscreened_list(p);
        std::vector<int> screened_list(p);
```

```
        std::vector<int> screened_to_idx(p, -1);

        int unscreened_size = p;
        int screened_size = 0;
        for (int j = 0; j < p; ++j) {
            unscreened_list[j] = j;
        }

        VectorXd grad = grad_init; // grad can be modified/overwritten

        // Reference state for Cauchy-Schwarz KKT pruning
        VectorXd grad_ref;
        std::vector<double> s;
        int lambdas_since_reset = 0;

        if (use_cs) {
            grad_ref = grad_init;
            s.resize(p);
            for (int j = 0; j < p; ++j) {
                s[j] = std::sqrt(xv(j) * inv_fn);
            }
        }

        auto add_active = [&](int j) {
            if (feat_to_idx[j] != -1) return;

            // O(1) swap-deletion from screened_list to maintain partition disjointness
            int idx_in_screened = screened_to_idx[j];
            if (idx_in_screened >= 0) {
                int last_j = screened_list[screened_size - 1];
                screened_list[idx_in_screened] = last_j;
                screened_to_idx[last_j] = idx_in_screened;
                --screened_size;
                screened_to_idx[j] = -1;
            }

            int old_k = static_cast<int>(active.size());
            feat_to_idx[j] = old_k;
            active.push_back(j);
            int new_k = old_k + 1;

            if (new_k > current_capacity) {
                int new_capacity = current_capacity * 2;

                double* G_data2 = nullptr;
                double* c_data2 = nullptr;
                double* beta_active_data2 = nullptr;
                double* xv_active_data2 = nullptr;
                double* inv_xv_active_data2 = nullptr;
                double* grad_init_active_data2 = nullptr;
                double* beta_old_at_start2 = nullptr;

                if (posix_memalign((void**)&G_data2, 64, static_cast<size_t>(new_capacity) *
 new_capacity * sizeof(double)) != 0) { oom = true; return; }
                if (posix_memalign((void**)&c_data2, 64, static_cast<size_t>(new_capacity) *
 sizeof(double)) != 0) { free(G_data2); oom = true; return; }
                if (posix_memalign((void**)&beta_active_data2, 64, static_cast<size_t>(
new_capacity) * sizeof(double)) != 0) { free(G_data2); free(c_data2); oom = true;
return; }
                if (posix_memalign((void**)&xv_active_data2, 64, static_cast<size_t>(
new_capacity) * sizeof(double)) != 0) { free(G_data2); free(c_data2); free(
beta_active_data2); oom = true; return; }
                if (posix_memalign((void**)&inv_xv_active_data2, 64, static_cast<size_t>(
new_capacity) * sizeof(double)) != 0) { free(G_data2); free(c_data2); free(
beta_active_data2); free(xv_active_data2); oom = true; return; }
                if (posix_memalign((void**)&grad_init_active_data2, 64, static_cast<size_t>(
new_capacity) * sizeof(double)) != 0) { free(G_data2); free(c_data2); free(
beta_active_data2); free(xv_active_data2); free(inv_xv_active_data2); oom = true;
return; }
                if (posix_memalign((void**)&beta_old_at_start2, 64, static_cast<size_t>(
new_capacity) * sizeof(double)) != 0) { free(G_data2); free(c_data2); free(
beta_active_data2); free(xv_active_data2); free(inv_xv_active_data2); free(
grad_init_active_data2); oom = true; return; }

                std::fill(G_data2, G_data2 + static_cast<size_t>(new_capacity) *
new_capacity, 0.0);

                if (old_k > 0) {
                    int old_k_padded = (old_k + 7) & ~7;
                    for (int col = 0; col < old_k; ++col) {
                        double* dest_col = G_data2 + col * new_capacity;
```

```
                        const double* src_col = G_data + col * current_capacity;
                        #pragma omp simd aligned(dest_col, src_col: 64)
                        for (int row = 0; row < old_k_padded; ++row) {
                            dest_col[row] = src_col[row];
                        }
                    }

                    #pragma omp simd aligned(c_data2, c_data: 64)
                    for (int i = 0; i < old_k_padded; ++i) c_data2[i] = c_data[i];

                    #pragma omp simd aligned(beta_active_data2, beta_active_data: 64)
                    for (int i = 0; i < old_k_padded; ++i) beta_active_data2[i] =
beta_active_data[i];

                    #pragma omp simd aligned(xv_active_data2, xv_active_data: 64)
                    for (int i = 0; i < old_k_padded; ++i) xv_active_data2[i] =
xv_active_data[i];

                    #pragma omp simd aligned(inv_xv_active_data2, inv_xv_active_data: 64)
                    for (int i = 0; i < old_k_padded; ++i) inv_xv_active_data2[i] =
inv_xv_active_data[i];

                    #pragma omp simd aligned(grad_init_active_data2, grad_init_active_data:
64)
                    for (int i = 0; i < old_k_padded; ++i) grad_init_active_data2[i] =
grad_init_active_data[i];

                    #pragma omp simd aligned(beta_old_at_start2, beta_old_at_start: 64)
                    for (int i = 0; i < old_k_padded; ++i) beta_old_at_start2[i] =
beta_old_at_start[i];
                }

                free(G_data);
                free(c_data);
                free(beta_active_data);
                free(xv_active_data);
                free(inv_xv_active_data);
                free(grad_init_active_data);
                free(beta_old_at_start);

                G_data = G_data2;
                c_data = c_data2;
                beta_active_data = beta_active_data2;
                xv_active_data = xv_active_data2;
                inv_xv_active_data = inv_xv_active_data2;
                grad_init_active_data = grad_init_active_data2;
                beta_old_at_start = beta_old_at_start2;
                current_capacity = new_capacity;
            }

            // SIMD 4x Register-Blocked Lazy Gram Precomputation (reduces column loads by
75%)
            const double* RESTRICT col_j = ASSUME_ALIGNED(X_padded + j * n_padded, 64);
            const bool run_parallel_lazy = (old_k >= 64 && static_cast<size_t>(n_padded) *
old_k >= 150000);

            #pragma omp parallel for schedule(static) if(run_parallel_lazy)
            for (int i = 0; i < (old_k / 4) * 4; i += 4) {
                const double* RESTRICT col0 = ASSUME_ALIGNED(X_padded + active[i] * n_padded
, 64);
                const double* RESTRICT col1 = ASSUME_ALIGNED(X_padded + active[i+1] *
n_padded, 64);
                const double* RESTRICT col2 = ASSUME_ALIGNED(X_padded + active[i+2] *
n_padded, 64);
                const double* RESTRICT col3 = ASSUME_ALIGNED(X_padded + active[i+3] *
n_padded, 64);

                double sum0 = 0.0, sum1 = 0.0, sum2 = 0.0, sum3 = 0.0;
                #pragma omp simd reduction(+:sum0, sum1, sum2, sum3) aligned(col_j, col0,
col1, col2, col3: 64)
                for (int k = 0; k < n_padded; ++k) {
                    double vj = col_j[k];
                    sum0 += vj * col0[k];
                    sum1 += vj * col1[k];
                    sum2 += vj * col2[k];
                    sum3 += vj * col3[k];
                }

                double r0 = sum0 * inv_fn;
                double r1 = sum1 * inv_fn;
                double r2 = sum2 * inv_fn;
```

```
                double r3 = sum3 * inv_fn;

                G_data[old_k * current_capacity + i] = r0;
                G_data[i * current_capacity + old_k] = r0;

                G_data[old_k * current_capacity + i + 1] = r1;
                G_data[(i + 1) * current_capacity + old_k] = r1;

                G_data[old_k * current_capacity + i + 2] = r2;
                G_data[(i + 2) * current_capacity + old_k] = r2;

                G_data[old_k * current_capacity + i + 3] = r3;
                G_data[(i + 3) * current_capacity + old_k] = r3;
            }

            for (int i = (old_k / 4) * 4; i < old_k; ++i) {
                const double* RESTRICT col_act = ASSUME_ALIGNED(X_padded + active[i] *
        n_padded, 64);
                double dot_val = 0.0;
                #pragma omp simd reduction(+:dot_val) aligned(col_j, col_act: 64)
                for (int k = 0; k < n_padded; ++k) {
                    dot_val += col_j[k] * col_act[k];
                }
                dot_val *= inv_fn;
                G_data[old_k * current_capacity + i] = dot_val;
                G_data[i * current_capacity + old_k] = dot_val;
            }
            G_data[old_k * current_capacity + old_k] = xv(j); // xv(j) is already scaled by
        inv_fn

            // Zero-O(n) initial correlation computation
            double sum_val = 0.0;
            const double* RESTRICT G_col = ASSUME_ALIGNED(G_data + old_k * current_capacity,
             64);
            const double* RESTRICT beta_act = ASSUME_ALIGNED(beta_active_data, 64);
            #pragma omp simd reduction(+:sum_val) aligned(G_col, beta_act: 64)
            for (int i = 0; i < old_k; ++i) {
                sum_val += G_col[i] * beta_act[i];
            }
            c_data[old_k] = grad_init(j) - sum_val;

            xv_active_data[old_k] = xv(j);
            inv_xv_active_data[old_k] = 1.0 / xv(j);
            grad_init_active_data[old_k] = grad_init[j];
            beta_active_data[old_k] = 0.0;
        };

        double prev_lam = 0.0;

        // Preallocate vectors to avoid repeated heap allocation
        std::vector<int> to_activate;
        std::vector<int> screened_violators;
        std::vector<int> unscreened_violators;
        std::vector<int> to_compute;

        to_activate.reserve(p);
        screened_violators.reserve(p);
        unscreened_violators.reserve(p);
        if (use_cs) {
            to_compute.reserve(p);
        }

        for (int li = 0; li < nlam; ++li) {
            const double lam  = lam_path(li);
            const double tlam = 2.0 * lam - prev_lam;

            // ---- Step 1: Strong-rule screening (with O(1) swap-deletion) ----
            double* RESTRICT grad_ptr = grad.data();
            for (int i = 0; i < unscreened_size; ) {
                int j = unscreened_list[i];
                if (std::abs(grad_ptr[j]) > tlam) {
                    screened[j] = 1;
                    screened_to_idx[j] = screened_size;
                    screened_list[screened_size++] = j;
                    unscreened_list[i] = unscreened_list[--unscreened_size];
                } else {
                    ++i;
                }
            }

            // ---- Step 2: Outer loop ----
```

```
            int nlp = 0;
            while (true) {
                // 2a. Identify violating features among screened features
                to_activate.clear();
                const double KKT_bound_screen = lam * (1.0 + 1e-9);
                for (int i = 0; i < screened_size; ++i) {
                    int j = screened_list[i];
                    // At this point, screened_list only contains non-active screened features.
                    // Absolutely no feat_to_idx branches needed!
                    if (std::abs(grad_ptr[j]) > KKT_bound_screen) {
                        to_activate.push_back(j);
                    }
                }

                // If some screened features violate KKT, add them to active set
                if (!to_activate.empty()) {
                    for (int j : to_activate) {
                        add_active(j);
                        if (oom) goto cleanup;
                    }
                }

                // 2b. CD over active set until convergence
                int active_size = static_cast<int>(active.size());

                // Save beta at the start of the outer iteration to track changes
                if (active_size > 0) {
                    int active_size_padded = (active_size + 7) & ~7;
                    #pragma omp simd aligned(beta_old_at_start, beta_active_data: 64)
                    for (int i = 0; i < active_size_padded; ++i) {
                        beta_old_at_start[i] = beta_active_data[i];
                    }
                }

                if (active_size > 0) {
                    double dmax = tol; // Ensure at least one sweep
                    while (dmax >= tol && nlp < maxit) {
                        ++nlp;
                        dmax = 0.0;
                        for (int idx = 0; idx < active_size; ++idx) {
                            const double bj_old = beta_active_data[idx];
                            // Division-free gradient calculation
                            const double gj     = c_data[idx] + bj_old * xv_active_data[idx];
                            const double bj_new = soft_thresh(gj, lam) * inv_xv_active_data[idx];
                            if (bj_new == bj_old) continue;
                            const double delta = bj_new - bj_old;
                            beta_active_data[idx] = bj_new;

                            // Extremely fast SIMD cache update (padded up to a multiple of 8)
                            int active_size_padded = (active_size + 7) & ~7;
                            double* RESTRICT c_ptr = ASSUME_ALIGNED(c_data, 64);
                            const double* RESTRICT G_col_ptr = ASSUME_ALIGNED(G_data + idx * current_capacity, 64);
                            #pragma omp simd aligned(c_ptr, G_col_ptr: 64)
                            for (int i = 0; i < active_size_padded; ++i) {
                                c_ptr[i] -= delta * G_col_ptr[i];
                            }

                            const double ch = xv_active_data[idx] * delta * delta;
                            if (ch > dmax) dmax = ch;
                        }
                    }
                }

                // Safety limit check
                if (nlp >= maxit) break;

                // Incremental O(n) residual update & any_changed check (Raw-Pointer hand-vectorized loop)
                bool any_changed = false;
                if (active_size > 0) {
                    double* RESTRICT r_ptr = ASSUME_ALIGNED(r_padded, 64);
                    for (int idx = 0; idx < active_size; ++idx) {
                        const double delta = beta_active_data[idx] - beta_old_at_start[idx];
                        if (delta != 0.0) {
                            const double* RESTRICT col_ptr = ASSUME_ALIGNED(X_padded + active[idx] * n_padded, 64);
```

```
                            #pragma omp simd aligned(r_ptr, col_ptr: 64)
                            for (int i = 0; i < n_padded; ++i) {
                                r_ptr[i] -= delta * col_ptr[i];
                            }
                            any_changed = true;
                        }
                    }
                }

                // O(k^2) exact re-sync of correlation cache c (Sparse-Skipping Custom Loop)
                if (any_changed && active_size > 0) {
                    int active_size_padded = (active_size + 7) & ~7;
                    #pragma omp simd aligned(c_data, grad_init_active_data: 64)
                    for (int i = 0; i < active_size_padded; ++i) {
                        c_data[i] = grad_init_active_data[i];
                    }
                    for (int j = 0; j < active_size; ++j) {
                        const double bj = beta_active_data[j];
                        if (bj != 0.0) {
                            const double* RESTRICT G_col = ASSUME_ALIGNED(G_data + j * current_capacity, 64);
                            double* RESTRICT c_ptr = ASSUME_ALIGNED(c_data, 64);
                            #pragma omp simd aligned(c_ptr, G_col: 64)
                            for (int i = 0; i < active_size_padded; ++i) {
                                c_ptr[i] -= bj * G_col[i];
                            }
                        }
                    }
                }

                // 2c. Robust Two-Stage KKT check
                bool screened_kkt_ok = true;
                screened_violators.clear();
                const double KKT_bound = lam * (1.0 + 1e-9);

                // SIMD 4x Register-Blocked Screened KKT Checks (reduces residual vector loads by 75%)
                const double* RESTRICT r_ptr = ASSUME_ALIGNED(r_padded, 64);
                const bool run_parallel_screened = (static_cast<size_t>(n_padded) * screened_size >= 150000);

                #pragma omp parallel for schedule(static) if(run_parallel_screened)
                for (int i = 0; i < (screened_size / 4) * 4; i += 4) {
                    int j0 = screened_list[i];
                    int j1 = screened_list[i+1];
                    int j2 = screened_list[i+2];
                    int j3 = screened_list[i+3];

                    const double* RESTRICT col0 = ASSUME_ALIGNED(X_padded + j0 * n_padded, 64);
                    const double* RESTRICT col1 = ASSUME_ALIGNED(X_padded + j1 * n_padded, 64);
                    const double* RESTRICT col2 = ASSUME_ALIGNED(X_padded + j2 * n_padded, 64);
                    const double* RESTRICT col3 = ASSUME_ALIGNED(X_padded + j3 * n_padded, 64);

                    double sum0 = 0.0, sum1 = 0.0, sum2 = 0.0, sum3 = 0.0;
                    #pragma omp simd reduction(+:sum0, sum1, sum2, sum3) aligned(r_ptr, col0, col1, col2, col3: 64)
                    for (int k = 0; k < n_padded; ++k) {
                        double rk = r_ptr[k];
                        sum0 += rk * col0[k];
                        sum1 += rk * col1[k];
                        sum2 += rk * col2[k];
                        sum3 += rk * col3[k];
                    }
                    grad_ptr[j0] = sum0 * inv_fn;
                    grad_ptr[j1] = sum1 * inv_fn;
                    grad_ptr[j2] = sum2 * inv_fn;
                    grad_ptr[j3] = sum3 * inv_fn;
                }

                for (int i = (screened_size / 4) * 4; i < screened_size; ++i) {
                    int j = screened_list[i];
                    const double* RESTRICT col_ptr = ASSUME_ALIGNED(X_padded + j * n_padded, 64);
                    double dot_val = 0.0;
                    #pragma omp simd reduction(+:dot_val) aligned(r_ptr, col_ptr: 64)
                    for (int k = 0; k < n_padded; ++k) {
                        dot_val += col_ptr[k] * r_ptr[k];
```

```
                    }
                    grad_ptr[j] = dot_val * inv_fn;
                }

                for (int i = 0; i < screened_size; ++i) {
                    int j = screened_list[i];
                    if (std::abs(grad_ptr[j]) > KKT_bound) {
                        screened_violators.push_back(j);
                        screened_kkt_ok = false;
                    }
                }

                if (!screened_kkt_ok) {
                    // Add screened violators to active set and run CD again
                    for (int j : screened_violators) {
                        add_active(j);
                        if (oom) goto cleanup;
                    }
                    continue; // Skip full KKT check, go back to CD
                }

                // Only perform full KKT check on unscreened features if screened is 100% OK
                bool full_kkt_ok = true;
                unscreened_violators.clear();

                if (use_cs) {
                    // Dual-Phase Adaptive Cauchy-Schwarz KKT Pruning!
                    double d2 = 0.0;
                    const double* RESTRICT r_curr_ptr = ASSUME_ALIGNED(r_padded, 64);
                    const double* RESTRICT r_ref_ptr = ASSUME_ALIGNED(r_ref_padded, 64);
                    #pragma omp simd reduction(+:d2) aligned(r_curr_ptr, r_ref_ptr: 64)
                    for (int k = 0; k < n_padded; ++k) {
                        double diff = r_curr_ptr[k] - r_ref_ptr[k];
                        d2 += diff * diff;
                    }
                    double d = std::sqrt(d2);

                    const double* RESTRICT grad_ref_ptr = grad_ref.data();
                    const double* RESTRICT s_ptr = s.data();
                    const int* RESTRICT unscreened_ptr = unscreened_list.data();

                    to_compute.clear();
                    for (int i = 0; i < unscreened_size; ++i) {
                        int j = unscreened_ptr[i];
                        double bound = std::abs(grad_ref_ptr[j]) + s_ptr[j] * d;
                        if (bound > KKT_bound) {
                            to_compute.push_back(j);
                        }
                    }

                    int num_to_compute = to_compute.size();
                    bool did_reset = false;

                    if (num_to_compute > 0.3 * p || lambdas_since_reset >= 8) {
                        // Drift is too large or reset interval reached, do a full reset (SIMD 4x Register-Blocked)
                        const bool run_parallel_reset = (static_cast<size_t>(n_padded) * unscreened_size >= 150000);
                        #pragma omp parallel for schedule(static) if(run_parallel_reset)
                        for (int i = 0; i < (unscreened_size / 4) * 4; i += 4) {
                            int j0 = unscreened_list[i];
                            int j1 = unscreened_list[i+1];
                            int j2 = unscreened_list[i+2];
                            int j3 = unscreened_list[i+3];

                            const double* RESTRICT col0 = ASSUME_ALIGNED(X_padded + j0 * n_padded, 64);
                            const double* RESTRICT col1 = ASSUME_ALIGNED(X_padded + j1 * n_padded, 64);
                            const double* RESTRICT col2 = ASSUME_ALIGNED(X_padded + j2 * n_padded, 64);
                            const double* RESTRICT col3 = ASSUME_ALIGNED(X_padded + j3 * n_padded, 64);
                            const double* RESTRICT r_ptr_exact = ASSUME_ALIGNED(r_padded, 64);

                            double sum0 = 0.0, sum1 = 0.0, sum2 = 0.0, sum3 = 0.0;
                            #pragma omp simd reduction(+:sum0, sum1, sum2, sum3) aligned(r_ptr_exact, col0, col1, col2, col3: 64)
                            for (int k = 0; k < n_padded; ++k) {
                                double rk = r_ptr_exact[k];
```

```
                                sum0 += rk * col0[k];
                                sum1 += rk * col1[k];
                                sum2 += rk * col2[k];
                                sum3 += rk * col3[k];
                            }
                            grad_ptr[j0] = sum0 * inv_fn;
                            grad_ptr[j1] = sum1 * inv_fn;
                            grad_ptr[j2] = sum2 * inv_fn;
                            grad_ptr[j3] = sum3 * inv_fn;
                        }

                        for (int i = (unscreened_size / 4) * 4; i < unscreened_size; ++i) {
                            int j = unscreened_list[i];
                            const double* RESTRICT col_ptr = ASSUME_ALIGNED(X_padded + j * n_padded, 64);
                            const double* RESTRICT r_ptr_exact = ASSUME_ALIGNED(r_padded, 64);
                            double sum = 0.0;
                            #pragma omp simd reduction(+:sum) aligned(r_ptr_exact, col_ptr: 64)
                            for (int k = 0; k < n_padded; ++k) {
                                sum += r_ptr_exact[k] * col_ptr[k];
                            }
                            grad_ptr[j] = sum * inv_fn;
                        }

                        std::memcpy(r_ref_padded, r_padded, n_padded * sizeof(double));

                        double* RESTRICT grad_ref_ptr_writable = grad_ref.data();
                        #pragma omp parallel for schedule(static) if(unscreened_size >= 2048)
                        for (int i = 0; i < unscreened_size; ++i) {
                            int j = unscreened_ptr[i];
                            grad_ref_ptr_writable[j] = grad_ptr[j];
                        }
                        lambdas_since_reset = 0;
                        did_reset = true;
                    } else {
                        // Compute exact gradients only for the tiny unpruned subset (SIMD 4 x Register-Blocked)
                        const bool run_parallel_comp = (num_to_compute >= 32 && static_cast<size_t>(n_padded) * num_to_compute >= 150000);
                        #pragma omp parallel for schedule(static) if(run_parallel_comp)
                        for (int k = 0; k < (num_to_compute / 4) * 4; k += 4) {
                            int j0 = to_compute[k];
                            int j1 = to_compute[k+1];
                            int j2 = to_compute[k+2];
                            int j3 = to_compute[k+3];

                            const double* RESTRICT col0 = ASSUME_ALIGNED(X_padded + j0 * n_padded, 64);
                            const double* RESTRICT col1 = ASSUME_ALIGNED(X_padded + j1 * n_padded, 64);
                            const double* RESTRICT col2 = ASSUME_ALIGNED(X_padded + j2 * n_padded, 64);
                            const double* RESTRICT col3 = ASSUME_ALIGNED(X_padded + j3 * n_padded, 64);
                            const double* RESTRICT r_ptr_exact = ASSUME_ALIGNED(r_padded, 64);

                            double sum0 = 0.0, sum1 = 0.0, sum2 = 0.0, sum3 = 0.0;
                            #pragma omp simd reduction(+:sum0, sum1, sum2, sum3) aligned(r_ptr_exact, col0, col1, col2, col3: 64)
                            for (int m = 0; m < n_padded; ++m) {
                                double rk = r_ptr_exact[m];
                                sum0 += rk * col0[m];
                                sum1 += rk * col1[m];
                                sum2 += rk * col2[m];
                                sum3 += rk * col3[m];
                            }
                            grad_ptr[j0] = sum0 * inv_fn;
                            grad_ptr[j1] = sum1 * inv_fn;
                            grad_ptr[j2] = sum2 * inv_fn;
                            grad_ptr[j3] = sum3 * inv_fn;
                        }

                        for (int k = (num_to_compute / 4) * 4; k < num_to_compute; ++k) {
                            int j = to_compute[k];
                            const double* RESTRICT col_ptr = ASSUME_ALIGNED(X_padded + j * n_padded, 64);
```

```
                            const double* RESTRICT r_ptr_exact = ASSUME_ALIGNED(r_padded, 64);
                            double sum = 0.0;
                            #pragma omp simd reduction(+:sum) aligned(r_ptr_exact, col_ptr: 64)
                            for (int m = 0; m < n_padded; ++m) {
                                sum += r_ptr_exact[m] * col_ptr[m];
                            }
                            grad_ptr[j] = sum * inv_fn;
                        }
                    }

                    for (int i = 0; i < unscreened_size; ) {
                        int j = unscreened_list[i];
                        if (std::abs(grad_ptr[j]) > KKT_bound) {
                            screened[j] = 1;
                            unscreened_violators.push_back(j);
                            screened_to_idx[j] = screened_size;
                            screened_list[screened_size++] = j;
                            unscreened_list[i] = unscreened_list[--unscreened_size];
                            full_kkt_ok = false;
                        } else {
                            ++i;
                        }
                    }

                    if (full_kkt_ok) {
                        if (!did_reset) {
                            lambdas_since_reset++;
                        }
                    }
                } else {
                    // Standard, clean KKT check without CS pruning overhead on small/medium problems (SIMD 4x Register-Blocked)
                    const bool run_parallel_uns_std = (static_cast<size_t>(n_padded) * unscreened_size >= 150000);
                    #pragma omp parallel for schedule(static) if(run_parallel_uns_std)
                    for (int i = 0; i < (unscreened_size / 4) * 4; i += 4) {
                        int j0 = unscreened_list[i];
                        int j1 = unscreened_list[i+1];
                        int j2 = unscreened_list[i+2];
                        int j3 = unscreened_list[i+3];

                        const double* RESTRICT col0 = ASSUME_ALIGNED(X_padded + j0 * n_padded, 64);
                        const double* RESTRICT col1 = ASSUME_ALIGNED(X_padded + j1 * n_padded, 64);
                        const double* RESTRICT col2 = ASSUME_ALIGNED(X_padded + j2 * n_padded, 64);
                        const double* RESTRICT col3 = ASSUME_ALIGNED(X_padded + j3 * n_padded, 64);
                        const double* RESTRICT r_ptr = ASSUME_ALIGNED(r_padded, 64);

                        double sum0 = 0.0, sum1 = 0.0, sum2 = 0.0, sum3 = 0.0;
                        #pragma omp simd reduction(+:sum0, sum1, sum2, sum3) aligned(r_ptr, col0, col1, col2, col3: 64)
                        for (int k = 0; k < n_padded; ++k) {
                            double rk = r_ptr[k];
                            sum0 += rk * col0[k];
                            sum1 += rk * col1[k];
                            sum2 += rk * col2[k];
                            sum3 += rk * col3[k];
                        }
                        grad_ptr[j0] = sum0 * inv_fn;
                        grad_ptr[j1] = sum1 * inv_fn;
                        grad_ptr[j2] = sum2 * inv_fn;
                        grad_ptr[j3] = sum3 * inv_fn;
                    }

                    for (int i = (unscreened_size / 4) * 4; i < unscreened_size; ++i) {
                        int j = unscreened_list[i];
                        const double* RESTRICT col_ptr = ASSUME_ALIGNED(X_padded + j * n_padded, 64);
                        const double* RESTRICT r_ptr = ASSUME_ALIGNED(r_padded, 64);
                        double sum = 0.0;
                        #pragma omp simd reduction(+:sum) aligned(r_ptr, col_ptr: 64)
                        for (int k = 0; k < n_padded; ++k) {
                            sum += r_ptr[k] * col_ptr[k];
                        }
                        grad_ptr[j] = sum * inv_fn;
                    }
```

```

                    for (int i = 0; i < unscreened_size; ) {
                        int j = unscreened_list[i];
                        if (std::abs(grad_ptr[j]) > KKT_bound) {
                            screened[j] = 1;
                            unscreened_violators.push_back(j);
                            screened_to_idx[j] = screened_size;
                            screened_list[screened_size++] = j;
                            unscreened_list[i] = unscreened_list[--unscreened_size];
                            full_kkt_ok = false;
                        } else {
                            ++i;
                        }
                    }
                }

                if (full_kkt_ok) {
                    break; // Converged completely!
                }

                // Add unscreened violators to active set
                for (int j : unscreened_violators) {
                    add_active(j);
                    if (oom) goto cleanup;
                }
            }

            // Synchronize beta with beta_active and save coefficients
            for (size_t idx = 0; idx < active.size(); ++idx) {
                beta(active[idx]) = beta_active_data[idx];
            }
            coef_path.col(li) = beta;
            prev_lam = lam;
        }
    }

cleanup:
    if (G_data) free(G_data);
    if (c_data) free(c_data);
    if (beta_active_data) free(beta_active_data);
    if (xv_active_data) free(xv_active_data);
    if (inv_xv_active_data) free(inv_xv_active_data);
    if (grad_init_active_data) free(grad_init_active_data);
    if (beta_old_at_start) free(beta_old_at_start);
    if (y_padded) free(y_padded);
    if (r_padded) free(r_padded);
    if (r_ref_padded) free(r_ref_padded);
}

int main() {
    int32_t n, p, n_lambda;
    if (fread(&n,        sizeof(int32_t), 1, stdin) != 1) return 1;
    if (fread(&p,        sizeof(int32_t), 1, stdin) != 1) return 1;
    if (fread(&n_lambda, sizeof(int32_t), 1, stdin) != 1) return 1;

    // X arrives row-major. Allocate RowMajor matrix to read the bytes directly!
    Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor> X_row(n, p);
    if (fread(X_row.data(), sizeof(double), static_cast<size_t>(n) * p, stdin)
            != static_cast<size_t>(n) * p) return 1;

    // Pad row dimension of X to the multiple of 8 (guarantees perfect alignment for each
     column)
    int n_padded = ((n + 7) / 8) * 8;
    double* X_padded = nullptr;
    if (posix_memalign((void**)&X_padded, 64, static_cast<size_t>(n_padded) * p * sizeof(
     double)) != 0) return 1;

    VectorXd y(n);
    if (fread(y.data(), sizeof(double), n, stdin) != static_cast<size_t>(n)) return 1;

    VectorXd lam_path(n_lambda);
    if (fread(lam_path.data(), sizeof(double), n_lambda, stdin)
            != static_cast<size_t>(n_lambda)) return 1;

    MatrixXd coef_path = MatrixXd::Zero(p, n_lambda);

    VectorXd xv(p);
    VectorXd grad_init(p);

    const double* RESTRICT y_ptr = y.data();
    const double inv_fn = 1.0 / n;
```

```
    // 2D Cache-Blocked parallel Fused Transposition-Precomputation-Padding (FTPP)
    // Avoids separate allocation/std::fill overhead of X_padded and completely saves a full pass reading X!
    #pragma omp parallel
    {
        int nthreads = omp_get_num_threads();
        int tid = omp_get_thread_num();

        // Static partition of columns j to completely prevent thread false-sharing
        int j_per_thread = (p + nthreads - 1) / nthreads;
        int sj = tid * j_per_thread;
        int ej = std::min(sj + j_per_thread, p);

        if (sj < ej) {
            const int col_block = 64;
            const int row_block = 64;
            for (int bj = sj; bj < ej; bj += col_block) {
                int lim_j = std::min(bj + col_block, ej);

                double local_xx[64] = {0.0};
                double local_xy[64] = {0.0};

                for (int bi = 0; bi < n; bi += row_block) {
                    int lim_i = std::min(bi + row_block, n);
                    for (int j = bj; j < lim_j; ++j) {
                        int local_j = j - bj;
                        double* RESTRICT dest = X_padded + j * n_padded;
                        const double* RESTRICT src = X_row.data() + j;

                        double sum_xx = 0.0;
                        double sum_xy = 0.0;
                        #pragma omp simd reduction(+:sum_xx, sum_xy)
                        for (int i = bi; i < lim_i; ++i) {
                            double val = src[i * p];
                            dest[i] = val;
                            sum_xx += val * val;
                            sum_xy += val * y_ptr[i];
                        }
                        local_xx[local_j] += sum_xx;
                        local_xy[local_j] += sum_xy;
                    }
                }

                // Set the padded elements of each column to 0.0, and store precomputed xv and grad_init
                for (int j = bj; j < lim_j; ++j) {
                    double* RESTRICT dest = X_padded + j * n_padded;
                    for (int i = n; i < n_padded; ++i) {
                        dest[i] = 0.0;
                    }
                    xv(j) = local_xx[j - bj] * inv_fn;
                    grad_init(j) = local_xy[j - bj] * inv_fn;
                }
            }
        }
    }

    // Immediately free memory of X_row to minimize memory footprint
    X_row.resize(0, 0);

    const double thresh = 1e-9;
    const int    maxit  = 100000;

    solve_active_set(X_padded, n_padded, n, p, y, lam_path, xv, grad_init, coef_path, thresh, maxit);

    fwrite(coef_path.data(), sizeof(double),
           static_cast<size_t>(p) * n_lambda, stdout);

    free(X_padded);
    return 0;
}
'''

COMPILE_FLAGS = ["-fopenmp", "-ffast-math"]

# EVOLVE-BLOCK-END
```

E.2 CONVDIV

Listing 4: Complete ConvDiv Kernel discovered by DREAM-RSI.

```python
# EVOLVE-BLOCK-START
import torch
import torch.nn as nn
import triton
import triton.language as tl
import importlib.util
import os
import sys

def generate_kernel_code():
    code = """
import triton
import triton.language as tl

@triton.jit
def fused_conv_pool_kernel_optimized(
    X_ptr, W_ptr, Out_ptr,
    B: tl.constexpr
):
    pid_h = tl.program_id(0)
    pid_w = tl.program_id(1)
    pid_b = tl.program_id(2)

    h_conv_0 = 2 * pid_h

    w_start = pid_w * 32
    w_out_idx = w_start + tl.arange(0, 32)
    # W is physically 64, but kernel is 3, so valid output is 62.
    mask_w = (w_out_idx < 62)[None, :]

    # 99.6% of the execution takes the safe path
    is_safe = (pid_w == 0) or (pid_b != B - 1)

    w_ptrs_base = W_ptr + tl.arange(0, 16)[:, None] * 80 + tl.arange(0, 16)[None, :]

"""
    for i in range(14):
        code += f"    acc_{i}_0 = tl.zeros((16, 32), dtype=tl.float32)\n"
        code += f"    acc_{i}_1 = tl.zeros((16, 32), dtype=tl.float32)\n"

    code += """
    X_spatial_base_0 = pid_b * 524288 + h_conv_0 * 64 + w_out_idx
    X_base_ptr_0 = X_ptr + X_spatial_base_0[None, :]

    for k_start in range(0, 80, 16):
        k_idx = k_start + tl.arange(0, 16)
        k_idx_clamped = tl.minimum(k_idx, 71)
        ic = k_idx_clamped // 9
        kh = (k_idx_clamped % 9) // 3
        kw = k_idx_clamped % 3
        k_off_vec = ic * 65536 + kh * 64 + kw

        x_ptrs_0 = X_base_ptr_0 + k_off_vec[:, None]

        w_ptrs = w_ptrs_base + k_start
        w0 = tl.load(w_ptrs)
        w1 = tl.load(w_ptrs + 1280)
        w2 = tl.load(w_ptrs + 2560)

        # --- H = 0 Pass ---
        # Branch intentionally kept inside loop to prevent cp.async and preserve SM occupancy
        if is_safe:
"""
    for d in range(16):
        code += f"            x{d} = tl.load(x_ptrs_0 + {d * 4096}).to(tl.float16)\n"
    code += "        else:\n"
    for d in range(16):
        code += f"            x{d} = tl.load(x_ptrs_0 + {d * 4096}, mask=mask_w, other=0.0).to(tl.float16)\n"

    code += "\n        # Perfect Wide-ILP sequences (14-instruction dependency gap)\n"
    for w_idx in range(3):
        for d in range(14):
            code += f"        acc_{d}_0 = tl.dot(w{w_idx}, x{d + w_idx}, acc_{d}_0)\n"

```

```
    code += """
        # --- H = 1 Pass (Registers completely reused) ---
        if is_safe:
"""
    for d in range(16):
        code += f"            x{d} = tl.load(x_ptrs_0 + {64 + d * 4096}).to(tl.float16)\n"
    code += "        else:\n"
    for d in range(16):
        code += f"            x{d} = tl.load(x_ptrs_0 + {64 + d * 4096}, mask=mask_w, other=0.0).to(tl.float16)\n"

    code += "\n"
    for w_idx in range(3):
        for d in range(14):
            code += f"        acc_{d}_1 = tl.dot(w{w_idx}, x{d + w_idx}, acc_{d}_1)\n"

    code += "\n    # Pool D and H passes simultaneously\n"
    for d in range(7):
        code += f"    max_h_d{d} = tl.maximum(tl.maximum(acc_{d*2}_0, acc_{d*2+1}_0), tl.maximum(acc_{d*2}_1, acc_{d*2+1}_1))\n"

    code += "\n    # Deferred W-pooling outside the loop on 16x32 registers\n"
    for d in range(7):
        code += f"    max_pool_d{d} = tl.max(tl.reshape(max_h_d{d}, (16, 16, 2)), axis=2)\n"

    code += """
    # Balanced Tree Reduction
    s0 = max_pool_d0 + max_pool_d1
    s1 = max_pool_d2 + max_pool_d3
    s2 = max_pool_d4 + max_pool_d5
    s01 = s0 + s1
    s26 = s2 + max_pool_d6
    sum_all = s01 + s26

    # Deferred W-Masking: Zero out garbage from unpredicated reads on the right boundary.
    if pid_w == 1:
        w_pool_idx = tl.arange(0, 16)[None, :]
        sum_all = tl.where(w_pool_idx < 15, sum_all, 0.0)

    total_sum = tl.sum(sum_all, axis=1)

    # Unscaled block sum
    out_val = tl.sum(total_sum, axis=0)

    out_idx = pid_b * 62 + pid_w * 31 + pid_h
    tl.store(Out_ptr + out_idx, out_val)

@triton.jit
def reduce_scale_bias_kernel(
    Out_buf_ptr,
    Out_ptr,
    scale,
    bias,
    B: tl.constexpr
):
    pid_b = tl.program_id(0)
    if pid_b >= B:
        return

    offsets = tl.arange(0, 64)
    mask = offsets < 62
    vals = tl.load(Out_buf_ptr + pid_b * 62 + offsets, mask=mask, other=0.0)

    sum_val = tl.sum(vals)
    final_val = sum_val * scale + bias

    tl.store(Out_ptr + pid_b, final_val)
"""
    return code

_KERNEL_CODE = generate_kernel_code()

_kernel_file = "/tmp/fused_kernel_kb8_fp16_optimized_v4.py"
with open(_kernel_file, "w") as f:
    f.write(_KERNEL_CODE)

spec = importlib.util.spec_from_file_location("fused_kernel_kb8_fp16_optimized_v4", _kernel_file)
mod = importlib.util.module_from_spec(spec)
sys.modules["fused_kernel_kb8_fp16_optimized_v4"] = mod
spec.loader.exec_module(mod)
```

```
fused_conv_pool_kernel_optimized = mod.fused_conv_pool_kernel_optimized
reduce_scale_bias_kernel = mod.reduce_scale_bias_kernel


class ModelNew(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, divisor, pool_size, bias_shape, sum_dim):
        super(ModelNew, self).__init__()
        self.conv = nn.Conv3d(in_channels, out_channels, kernel_size)
        self.divisor = divisor
        self.max_pool = nn.MaxPool3d(pool_size)
        self.global_avg_pool = nn.AdaptiveAvgPool3d((1, 1, 1))
        self.bias = nn.Parameter(torch.randn(bias_shape))
        self.sum_dim = sum_dim

        self._cached_W = None
        self._cached_scale = None
        self._total_bias = None
        self._cached_Out_buf = None

    def forward(self, x):
        B = x.shape[0]
        # Guard: Ensure input is exactly the expected shape
        if x.shape[1:] != (8, 16, 64, 64) or not x.is_cuda or not x.is_contiguous() or self.sum_dim not in (1, -4):
            x = self.conv(x)
            x = x / self.divisor
            x = self.max_pool(x)
            x = self.global_avg_pool(x)
            x = x + self.bias
            x = torch.sum(x, dim=self.sum_dim)
            return x

        device = x.device

        if self._cached_W is None:
            W = self.conv.weight
            W_permuted = W.permute(2, 0, 1, 3, 4).contiguous()
            W_flat = W_permuted.view(3, 16, 72)

            W_padded = torch.zeros((3, 16, 80), dtype=torch.float16, device=device)
            W_padded[:, :, :72] = W_flat.to(torch.float16)
            self._cached_W = W_padded

            # The spatial dimensions before global average pooling are:
            # Depth = 7, Height = 31, Width = 31.
            # 7 * 31 * 31 = 6727.
            self._cached_scale = float(1.0 / (6727.0 * self.divisor))
            bias_per_channel = (self.conv.bias / self.divisor + self.bias.view(-1)).to(torch.float32)
            self._total_bias = float(bias_per_channel.sum().item())

        if self._cached_Out_buf is None or self._cached_Out_buf.shape[0] != B:
            self._cached_Out_buf = torch.empty((B, 62), dtype=torch.float32, device=device)

        Out_buf = self._cached_Out_buf
        Out = torch.empty((B, 1, 1, 1), dtype=torch.float32, device=device)

        # Launching on 2D grid packed perfectly to valid H values: (31, 2, B)
        grid = (31, 2, B)
        # Using num_stages=4 to preserve original execution profile
        fused_conv_pool_kernel_optimized[grid](
            x, self._cached_W, Out_buf,
            B=B,
            num_warps=4,
            num_stages=4
        )

        reduce_grid = (B,)
        reduce_scale_bias_kernel[reduce_grid](
            Out_buf, Out,
            self._cached_scale, self._total_bias,
            B=B,
            num_warps=2
        )

        return Out
# EVOLVE-BLOCK-END
```