Google

Dream-RSI: Recursive Self-Improvement through Evolving Worlds

Tong Zheng[1,2], **Xidong Wu**[1], **Zheng Zhang**[1], **Zhankui He**[3], **Chaoyi Zhang**[1], **Benjamin Coleman**[3], **Ruoqiao Wei**[1], **Di Bai**[3], **Haolin Liu**[4], **Rui Liu**[2], **Xue Wang**[1], **Yue Zhuan**[1], **Wang-Cheng Kang**[3], **Renkai Xiang**[1], **Heng Huang**[2], **Xinwu Cheng**[1] **and Yunsong Guo**[1]

[1]Google, [2]University of Maryland, College Park, [3]Google Deepmind, [4]University of Virginia

Recursive self-improvement is becoming increasingly vital for autonomous AI agents, where progress hinges on discovering high-value solutions across complex domains. The driver of this process is effective exploration, however, 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 requires navigating vast meta-search spaces under delayed and expensive feedback over long-horizon rollouts. We introduce Dream-RSI, a framework for scalable and recursively self-improving exploration. A lightweight orchestration layer makes exploration explicit and programmable while leaving the underlying coding agent unchanged. Our key insight is that accumulated discovery history can serve as a replay simulator over the realized search space. By performing dreaming in the replay simulator constructed from historical discovery trees, Dream-RSI secures immediate, low-cost off-policy feedback to evaluate and refine exploration policies without invoking repetitive, expensive online evaluations. The improved policy is subsequently redeployed online to drive further discovery, continuously expanding the simulator pool in a self-improving loop. Across algorithm engineering, mathematical optimization, and GPU kernel engineering, Dream-RSI achieves competitive or improved discovery quality while substantially reducing discovery cost in several settings.

github.com/zhengkid/Dream-RSI | dream-rsi.com

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 discovery loop wherein agents generate candidate solutions, evaluate outcomes, incorporate feedback, and refine future iterations. Such discovery loops have driven substantial progress across scientific and algorithmic domains, including algorithm design (Novikov et al., 2025; Romera-Paredes et al., 2024), open-ended mathematical optimization (Anthropic, 2026; Georgiev et al., 2025), systems design (Cao et al., 2026; Jaber and Jaber, 2026), and agent self-improvement (Lee et al., 2026; Zhang et al., 2026b,c; Zheng et al., 2026), with these discoveries increasingly feeding into the development of more capable AI systems. As agent capabilities improve and self-improvement targets become challenging, discovery increasingly requires long-horizon exploration over vast search spaces, often spanning thousands of proposal–evaluation cycles (OpenAI, 2026; Ye et al., 2026). At this scale, the ability to orchestrate exploration becomes critical. Poor exploration can waste substantial computation and time, severely limiting the efficiency and scalability of RSI.

Existing approaches have largely relied on manually designed exploration strategies that remain largely fixed throughout discovery (Du et al., 2026; Jiang et al., 2026; Novikov et al., 2025; Yan et al., 2026b; Ye et al., 2026). Fixed strategies cannot improve from accumulated discovery experience and may repeatedly allocate computation to ineffective search directions. Recent work therefore seeks to optimize exploration policies online during discovery (Liu et al., 2026a), but doing so faces

Corresponding author(s): xidongwu@google.com, zzhangx@google.com
© 2026 Google. All rights reserved

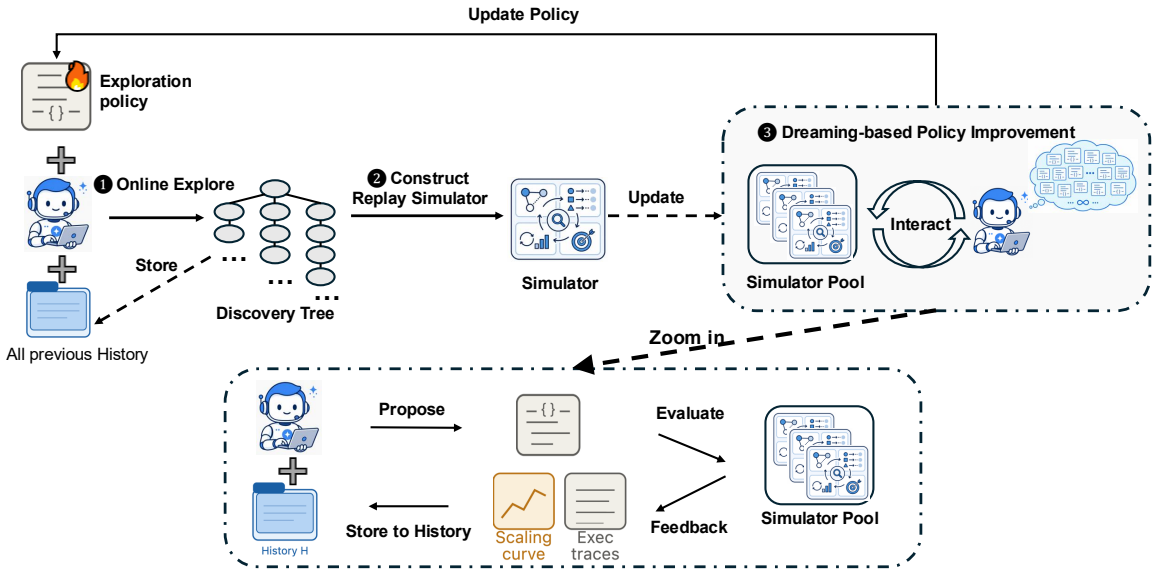


Figure 1 | **Overview of DREAM-RSI.** The system operates in a recursive self-improvement loop via three core stages: ① **Online Explore**, where the current exploration policy guides a coding agent to expand a discovery tree and log historical traces; ② **Construct Replay Simulator**, where the generated discovery tree is converted into a reusable simulator pool; and ③ **Dreaming-based Policy Improvement**, where the agent "dreams" up a massive pool of alternative policies in its mind. It then feeds these candidate policies into the replay simulator to simulate executions and derive rapid feedback, continuously refining its strategy (detailed in the Zoom-in box). The updated policy then redeploys for the next round of online exploration.

two fundamental bottlenecks. First, feedback is delayed and expensive at the meta level: unlike evaluating an individual candidate, assessing an exploration policy requires observing how it shapes the subsequent discovery process over many proposal–evaluation cycles. Second, the meta-policy space is vast: a newly proposed policy may perform poorly, so many alternatives may need to be tried. Together, these challenges make meta-level improvement particularly costly: each policy may require a long online rollout before receiving useful feedback, making it difficult to efficiently close the self-improvement loop at the exploration layer.

To address these bottlenecks, our key intuition is simple: a fast and inexpensive simulator of discovery would allow many exploration policies to be evaluated before costly online deployment. **Surprisingly, completed discovery histories already provide such a simulator.** While prior work treats past discovery history merely as static textual context (Hu et al., 2025; Ouyang et al., 2026b) or training data for weight fine-tuning (Wang et al., 2025; Yuksekgonul et al., 2026), a completed discovery process inherently records a structured tree of past exploration decisions and their realized code-execution outcomes. Drawing an analogy to model-based reinforcement learning and World Models (Ha and Schmidhuber, 2018; Hafner et al., 2023) (§2), once organized into a discovery tree, this history can serve as a **replay simulator** [1]. As illustrated in Figure 2, an alternative exploration strategy can navigate this pre-recorded tree to traverse different subsets of recorded branches, in different orders, with different parallel groupings and stopping decisions. Because all execution outcomes are already saved in the tree, evaluating a new strategy requires only reading past records without rerunning the underlying discovery agent or evaluator. This transforms meta-policy improvement from an expensive online trial-and-error process into a fast, simulation-based "dreaming"

[1]We use the terms replay simulator and worlds interchangeably.

procedure.

Building on this insight, we introduce DREAM-RSI, a framework for scalable and recursively self-improving meta-exploration in agent-driven discovery. We first make exploration explicit and programmable through a lightweight orchestration layer that controls branching, parallel exploration, and stopping while leaving the underlying coding agent unchanged. Rather than keeping this policy fixed, DREAM-RSI establishes a closed-loop self-improvement mechanism across three core stages (Figure 1): **(1) Online Exploration**, where the current policy guides real-world discovery and logs historical execution traces; **(2) Simulator Construction**, where recorded discovery trees are converted into a reusable replay simulator pool; and **(3) Dreaming-based Policy Improvement**, where candidate policies are evaluated via low-cost "dreaming" over the simulator. The updated policy is then redeployed online to generate new discovery experience and expand the simulator pool, closing a RSI loop at the meta-exploration layer.

Empirically, we evaluate DREAM-RSI across 8 scientific discovery tasks spanning three distinct domains: algorithm engineering, mathematical optimization, and GPU kernel engineering. In algorithm engineering (Lasso path solver), DREAM-RSI outperforms standard libraries like `sklearn` and strong baselines while reducing agent calls by up to 162× over SIMPLETES and 1.7× over fixed-exploration baselines. In mathematical optimization (sum-difference, autocorrelation, circle packing), it matches or surpasses strong baselines within 1k generations, yielding over 50× budget savings compared to SIMPLETES. In GPU kernel engineering (KernelBench), it either reaches target execution speeds using 1.79×–2.43× fewer generations or improves kernel performance by up to 2.09× under identical budget constraints.

In summary, our main contributions are as follows: 1) **History as Replay Simulator:** We conceptualize completed discovery histories as replay simulators. This makes delayed exploration feedback reusable for efficient meta-exploration policy evaluation.; 2) **Meta-Layer RSI Loop (DREAM-RSI):** We introduce DREAM-RSI, establishing a recursive self-improvement loop that continuously collects discovery histories through online exploration, constructs replay simulators from history to refine meta-exploration strategies via dreaming, and redeploys the upgraded policy online; 3) **Empirical Validation:** We conduct experiments to demonstrate that DREAM-RSI improves both discovery effectiveness and efficiency in several settings.

2. Motivation: Discovery History as a Replay Simulator

Consider an agent navigating toward a goal in an unfamiliar environment. During its first traversal, the agent may follow inefficient routes, encounter dead ends, backtrack, and gradually construct a map of the surrounding space. Once recorded, however, this experience becomes reusable: the resulting map supports planning without requiring the agent to physically revisit every location. A new navigation policy can instead reason over the accumulated map, avoid known dead ends, reconsider earlier decisions, and compare alternative routes before acting (Gupta et al., 2017).

This idea parallels model-based reinforcement learning (M. Moerland et al., 2023; Sutton, 1990). A model captures how an environment evolves in response to an agent's actions, allowing policies to be trained or evaluated through simulated experience rather than repeated interaction with the real environment (Ha and Schmidhuber, 2018). The Dreamer family (Hafner et al., 2019, 2020, 2023, 2025) demonstrates this principle particularly clearly: an agent learns a compact dynamics model from collected experience and improves its policy by imagining trajectories within that model.

Long-horizon discovery admits an analogous structure. An exploration policy decides which directions to pursue, which candidates to refine, which branches to explore in parallel, and when

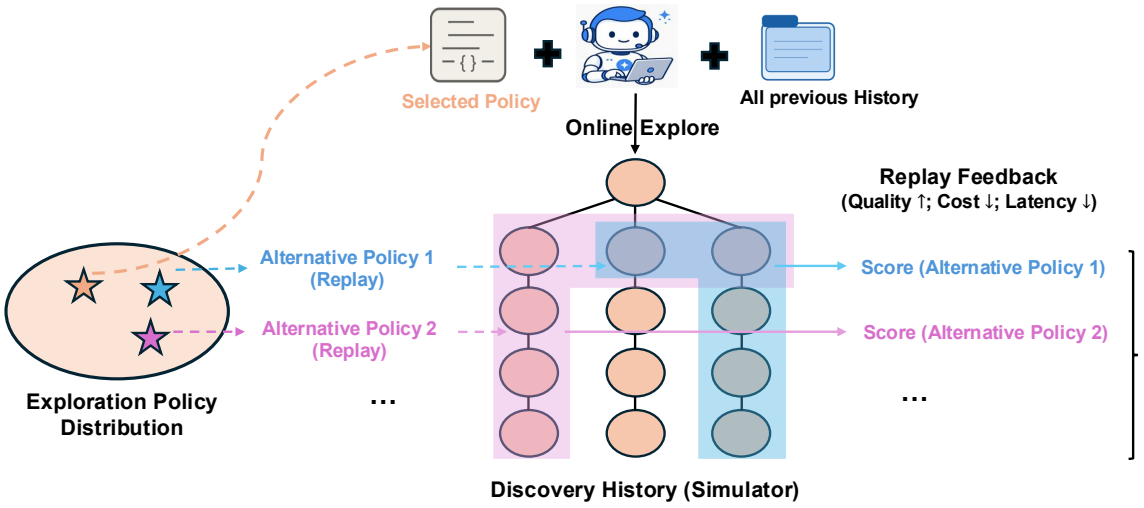


Figure 2 | **Discovery history as a replay simulator.** A deployed policy first explores online to generate a structured discovery tree containing historical execution traces (each node denote an attempt with its full observation). Thousands of candidate policies can then be tested within this simulator—evaluating alternative choices of search branches, exploration orders, concurrency levels, and stopping rules. **Since all node outcomes are pre-stored, a single costly online run enables thousands of rapid, zero-execution-cost off-policy evaluations. This enables policy improvement through historical replay: the agent can "dream" over many alternative exploration strategies before redeploying the improved policy online.**

to terminate. Executing the policy online produces a structured discovery history containing the explored branches, decision points, computational costs, and realized outcomes. As illustrated in Figure 2, this history can subsequently be treated as an *empirical replay simulator*: a grounded model of the portion of the discovery space that has already been observed.

Within this replay simulator, alternative exploration policies induce different trajectories through the recorded discovery tree. A policy may select a different subset of branches, prioritize them in a different order, issue different requests in parallel, or stop at an earlier point. Evaluating such a trajectory requires only revealing the outcomes already stored along the selected branches, rather than rerunning the underlying coding agent and evaluator. Consequently, a single expensive online discovery run can support many inexpensive evaluations of alternative exploration strategies.

3. DREAM-RSI: Recursive Self-Improvement through Evolving Worlds

As shown in Figure 1, DREAM-RSI alternates between online exploration and offline "dreaming" to improve how discovery computation is allocated. An executable *exploration policy*, implemented as code, determines where a fixed *discovery agent* continues searching, which attempts run in parallel, and when exploration stops. A fixed *evaluator* scores the resulting candidates and returns diagnostic feedback. Each online rollout records these attempts and their outcomes in a *discovery tree*. Once recorded, the same tree serves as a *replay world*: alternative exploration policies can be evaluated by revealing its stored outcomes, without rerunning the discovery agent or evaluator.

Let $t = 1, 2, \ldots$ index the outer iterations. Starting from an initial policy π_1 and an empty history $\mathcal{H}_0 = ()$, iteration t deploys π_t online to guide the *discovery agent* to collect a discovery trace $\mathcal{T}_t$ with tree structure. The completed tree is appended to the history, giving $\mathcal{H}_t = (\mathcal{T}_1, \ldots, \mathcal{T}_t)$. The method then evaluates the current policy and successive revisions on this fixed collection of trees through

replay. A fixed LLM-based *policy-development agent* uses the replay trajectories and scores to revise the exploration-policy code, and the best evaluated version is selected as π_{t+1} for the next online rollout. Thus, online exploration expands the available discovery history, while offline dreaming uses that history to improve the policy before its next deployment. Only the exploration-policy code is updated; the underlying models, evaluator, and execution interfaces remain fixed.

Both phases use the same tree-based decision interface. Their difference lies in how a continuation produces its next observation: online execution generates a new outcome, whereas replay reveals an outcome that has already been recorded.

Discovery trees and actions. A rollout incrementally builds a rooted discovery tree, initially containing only the task root r, which represents the initial workspace state. Each non-root node v has exactly one *primary parent*, either the root or a previously created node. This parent identifies where the attempt that creates v begins: the discovery agent resumes the parent's saved workspace and uses the observations accumulated there as context to generate a new candidate, which is then evaluated. Node v records the evolution history from r to v and the outcome of this single attempt, including the resulting filesystem snapshot, generated artifact, evaluation diagnostics, and score s_v. Scores follow a fixed task-scoring protocol, with larger values indicating better quality.

The only atomic operation is CONTINUE(v), which resumes the workspace associated with node v and generates and evaluates one new child. Continuing from the root opens a new branch; continuing from a non-root leaf refines or repairs the workspace at the end of an existing branch.

Let $W \geq 1$ be the number of parallel workers, each of which can execute one generation–evaluation request at a time (e.g. concurret API call). For a current tree $\mathcal{T}$, the eligible continuation nodes form the set $A(\mathcal{T}) = \{r\} \cup \{v \in \mathcal{T} : v \text{ is a leaf}\}$. The corresponding set of admissible batches is $A(\mathcal{T}; W) = \{C \subseteq A(\mathcal{T}) : |C| \leq W\}$. The exploration policy selects a batch $C \in A(\mathcal{T}; W)$ from the current tree. For every node $v \in C$,

One *decision round* consists of selecting this batch and observing its completed outcomes before the next decision. Selecting $C = \emptyset$ terminates the rollout. The root remains eligible after a branch is opened, whereas continuing from a non-root leaf advances that branch to its newly created child.

Online rollout. At outer iteration t, policy π_t guides a new online rollout for *discovery agent* with access to the completed history $\mathcal{H}_{t-1}$. Let $\mathcal{T}_{t,k}$ denote its discovery tree after k completed decision rounds, with $\mathcal{T}_{t,0} = \{r\}$. The rollout allows at most K_1 decision rounds. Whenever $k < K_1$, the policy selects a batch $C_{t,k} \in A(\mathcal{T}_{t,k}; W)$. If the batch is nonempty, the discovery agent executes CONTINUE(v) for each $v \in C_{t,k}$, and the evaluator scores the resulting attempts. Each attempt contributes a new child containing its updated workspace, artifact, score, and diagnostics. Attaching these children to their selected parents produces $\mathcal{T}_{t,k+1}$. The transition is stochastic because the discovery agent may generate different outcomes from the same workspace. The rollout ends when the policy selects an empty batch or completes K_1 decision rounds. Its final tree is recorded as $\mathcal{T}_t$ and appended to the history ($\mathcal{H}_t = \mathcal{H}_{t-1} \cup \{\mathcal{T}_t\}$). The resulting collection $\mathcal{H}_t$ is then used for offline evaluation and policy improvement.

Offline evaluation. During the offline phase of outer iteration t, the history $\mathcal{H}_t$ remains fixed while the exploration policy undergoes M code revisions. The initial version is $\pi_t^0 = \pi_t$, followed by revised versions $\pi_t^1, \ldots, \pi_t^{M-1}$. Every version is evaluated on every recorded tree $\mathcal{T}_i$, $i = 1, \ldots, t$. Here, m indexes policy versions, i indexes replay worlds, and k counts decision rounds within a single replay. The outer index t is fixed throughout this phase.

For a fixed policy version π_t^m and historical tree $\mathcal{T}_i$, replay starts from $\mathcal{T}_i^{m,0} = \{r\}$, where $\mathcal{T}_i^{m,k} \subseteq \mathcal{T}_i$ denotes the subtree revealed after k decision rounds. The full recorded tree $\mathcal{T}_i$ never changes during replay. The policy observes the revealed subtree and legal-action information, but not the outcomes of unrevealed nodes. At decision round k, policy π_t^m selects a batch $C_{t,i}^{m,k} \in A(\mathcal{T}_i^{m,k}; W)$. Unlike online execution that call the *discovery agent* to generate a new node, the transition in the offline replay is deterministic to the child of the selected node. For a nonempty batch, the replay reveals a new tree $\mathcal{T}_i^{m,k+1} = \mathcal{T}_i^{m,k} \cup \{\text{Child}(v; \mathcal{T}_i) : v \in C_i^{m,k}\}$ where $\text{Child}(v; \mathcal{T})$ denotes the child of v in tree $\mathcal{T}$ and can be an empty set. The stored workspace, artifact, score, and diagnostics of each newly revealed child are then observed by the policy. Each replay allows at most K_2 decision rounds. It terminates when the policy selects an empty batch, no recorded continuation remains available, or K_2 rounds have been completed. Let $k_i^{m,\star}$ denote the number of completed rounds. The final revealed tree therefore satisfies $\mathcal{T}_i^{m,k_i^{m,\star}} \subseteq \mathcal{T}_i$. Every policy–tree pair is replayed from the root, so evaluating a new policy version does not inherit the revealed subtree from an earlier evaluation.

Replay objective. The replay objective balances discovery quality, execution cost, and parallelism. Let $N_i^m = |\mathcal{T}_i^{m,k_i^{m,\star}}| - 1$ be the number of revealed non-root nodes. Although replay itself does not execute new discovery attempts, N_i^m counts the generation–evaluation requests represented by its trajectory. For fixed coefficients $\beta_1, \beta_2 \geq 0$, the replay score is

$$V_i^m = \underbrace{\max_{v \in \mathcal{T}_i^{m,k_i^{m,\star}}} s_v}_{\text{discovery quality}} - \underbrace{\beta_1 N_i^m}_{\text{execution cost}} + \underbrace{\beta_2 \frac{N_i^m}{\max\{1, k_i^{m,\star}\}}}_{\text{parallelism bonus}}. \tag{1}$$

The first term measures the best solution quality attained during replay. The second penalizes the number of attempted generations. For a nonempty replay, the third rewards the average number of attempts executed per decision round, favoring policies that batch useful continuations rather than execute them sequentially.

Policy improvement and selection. The evaluation score of policy version π_t^m is its average replay score across the fixed history, $V^m = \frac{1}{t}\sum_{i=1}^{t} V_i^m$. The offline phase begins by evaluating the current policy $\pi_t^0 = \pi_t$. For each $m = 0, \ldots, M-1$, the *policy-development agent* examines the replay trajectories and scores of π_t^m, together with feedback from earlier revisions, to identify successful decisions and recurring failures. It then revises the executable policy code to produce π_t^{m+1}, which is evaluated on the same t replay worlds. Replay feedback is available to the development agent between revisions.

After M revisions, the next online policy is selected from all M evaluated versions as $\pi_{t+1} = \pi_t^{m^\star}$, where $m^\star \in \arg\max_{m \in \{0,\ldots,M-1\}} V^m$. Because the candidate set includes the current policy, this selection satisfies $V^{m^\star} \geq V^0$. Thus, the selected policy π_{t+1} is no worse than the current policy π_t in average replay score on the fixed history $\mathcal{H}_t$. The selected policy is then deployed online to collect $\mathcal{T}_{t+1}$, expanding the history available for the next offline improvement phase.

4. Experiments

We evaluate DREAM-RSI across three scientific discovery domains:: algorithm engineering, kernel optimization and math optimization. Our primary controlled baseline is Recursive Fixed Exploration, which uses the same underlying discovery setting and initialization but keeps the exploration policy

fixed across recursive discovery rounds. We additionally compare against task-specific domain baselines.

Across all tasks, DREAM-RSI and Recursive Fixed Exploration use the same discovery agent, evaluator, initialization, and resource constraints. Both methods start from the same manually designed exploration policy. This exploration policy follows a simple *parallel refining* strategy: it launches multiple independent exploration workspaces in parallel, with each workspace maintaining its own local discovery trajectory and repeatedly refining its current candidate based on the history accumulated within that workspace. The two methods therefore follow the same exploration policy in the first discovery round. In subsequent rounds, while Recursive Fixed Exploration keeps its exploration policy static, DREAM-RSI progressively refines the policy by dreaming over a replay simulator conditioned on accumulated global discovery history, subsequently deploying the updated policy in each new round. The discovery cost is quantified by the total cumulative number of discovery-agent calls.

Specifically, we evaluate Gemini-3.1 Pro and Gemini-3.7-Flash across multiple recursive discovery rounds via the Gemini CLI [2]. Under Recursive Fixed Exploration, each round for Gemini-3.1 Pro executes 10 parallel workspaces with up to 11 refinement steps ($10 \times 11 = 110$ discovery-agent calls), whereas Gemini-3.7-Flash operates 32 parallel workspaces with up to 20 refinement steps ($32 \times 20 = 640$ calls). DREAM-RSI maintains identical per-round budgets, aligning with the baseline in Round 1 while progressively updating its policy in subsequent rounds. Further details on recursive rounds, task setups, resource budgets, and evaluation protocols follow below.

4.1. 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 both biological and non-biological domains.

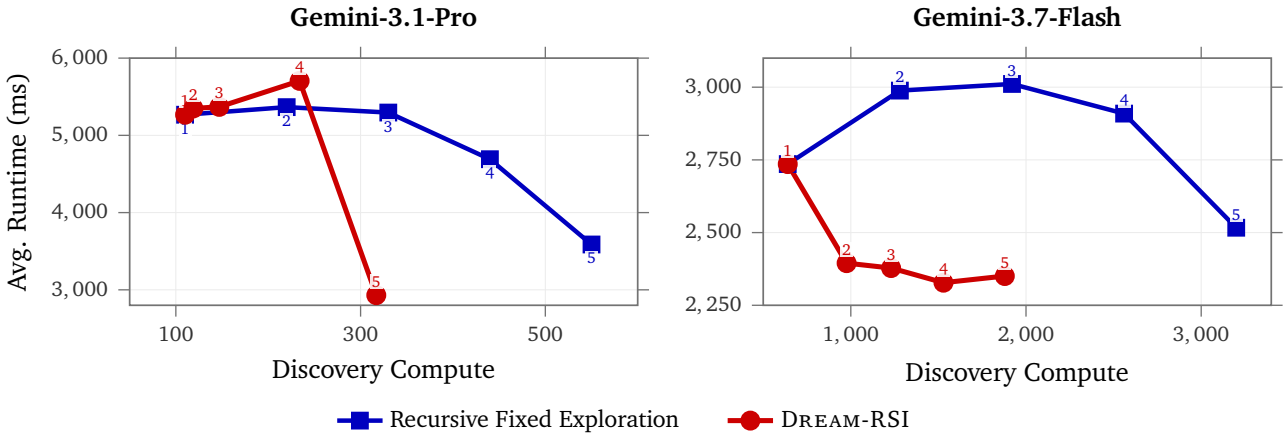
Baselines and Setup. We compare against standard Lasso solvers **sklearn** (Pedregosa et al., 2011) and **glmnet** (Friedman et al., 2010), as well as **SimpleTES** (Ye et al., 2026), which uses GPT-OSS-120B with a reported budget of 51,200 generations. We additionally include Recursive Fixed Exploration as our controlled baseline. Specifically, we run both Recursive Fixed Exploration and DREAM-RSI for 5 rounds.

Main Results. Figure 3(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

[2] https://geminicli.com/

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.



(b) Recursive Discovery Dynamics.

Figure 3 | **Lasso regularization-path discovery results. (a)** Final wall-clock runtime on six held-out downstream tasks; lower is better. **Compute** denotes the cumulative number of discovery-agent calls. **(b) Recursive discovery dynamics.** Average downstream runtime across six held-out tasks versus cumulative discovery compute for Gemini-3.1-Pro and Gemini-3.7-Flash. Numbers next to markers denote recursive rounds (iterations). Lower is better.

held-out datasets. Compared with SimpleTES, which uses 51,200 generations, DREAM-RSI achieves lower average downstream runtime with roughly two orders of magnitude fewer discovery-agent calls. Notably, the program discovered by Gemini-3.1-Pro appears particularly well suited to large-scale matrices such as RCV1. In contrast, Gemini-3.7-Flash discovers a more general-purpose program that performs consistently across different problem scales.

Recursive Discovery Dynamics. Figure 3(b) illustrates the trajectory of downstream performance across recursive discovery rounds relative to cumulative discovery compute. By design, both methods share identical search behavior in the initial round. In subsequent rounds, Recursive Fixed Exploration maintains a static exploration policy, whereas DREAM-RSI progressively refines and redeploys its policy via dreaming over accumulated discovery history. Consequently, the two trajectories diverge markedly: DREAM-RSI consistently achieves superior downstream performance while requiring substantially lower cumulative compute across both Gemini-3.1-Pro and Gemini-3.7-Flash.

Discovered Solver Analysis. We further analyze the discovered solver, with its implementation provided in the Appendix C. Unlike SimpleTES, which switches between LARS and coordinate descent

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**

according to problem dimensions, the discovered solver introduces adaptivity within the active-set optimization itself. It combines strong-rule screening with Cauchy–Schwarz-based KKT pruning, selectively recomputing exact gradients only when the bound cannot certify a feature and falling back to a full refresh when pruning becomes ineffective. This adaptive verification scheme is further integrated with efficient active-set bookkeeping, lazy Gram-matrix construction, and hardware-aware implementation.

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**. The goal of these problems is to discover high-quality solutions that optimize task-specific mathematical objectives under their respective constraints. Formal definitions of the three tasks are provided in Appendix.

We use Gemini-3.1 Pro via the Gemini CLI as the discovery agent for both Recursive Fixed Exploration and DREAM-RSI for 10 rounds. For each task, the agent iteratively proposes and evaluates candidate constructions or optimization procedures according to the task-specific objective. We compare against a broad set of existing automated discovery systems, including AlphaEvolve (Novikov et al., 2025), AlphaEvolveV2 (Georgiev et al., 2025), OpenEvolve (Sharma, 2025), CodeEvolve (Assumpção et al., 2025), ShinkaEvolve (Lange et al., 2026), TTS-Discovery (Yuksekgonul et al., 2026), ThetaEvolve (Wang et al., 2025), EvoX (Liu et al., 2026a), and SimpleTES (Ye et al., 2026).

Results. Table 1 summarizes the results across the three mathematical discovery tasks. 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. Notably, SimpleTES achieves state-of-the-art performance on Autocorrelation Inequalities, but requires 51,200 generations, significantly more than the fewer than 1,000 generations used by our approach. Overall, these results show that our DREAM-RSI generalize well

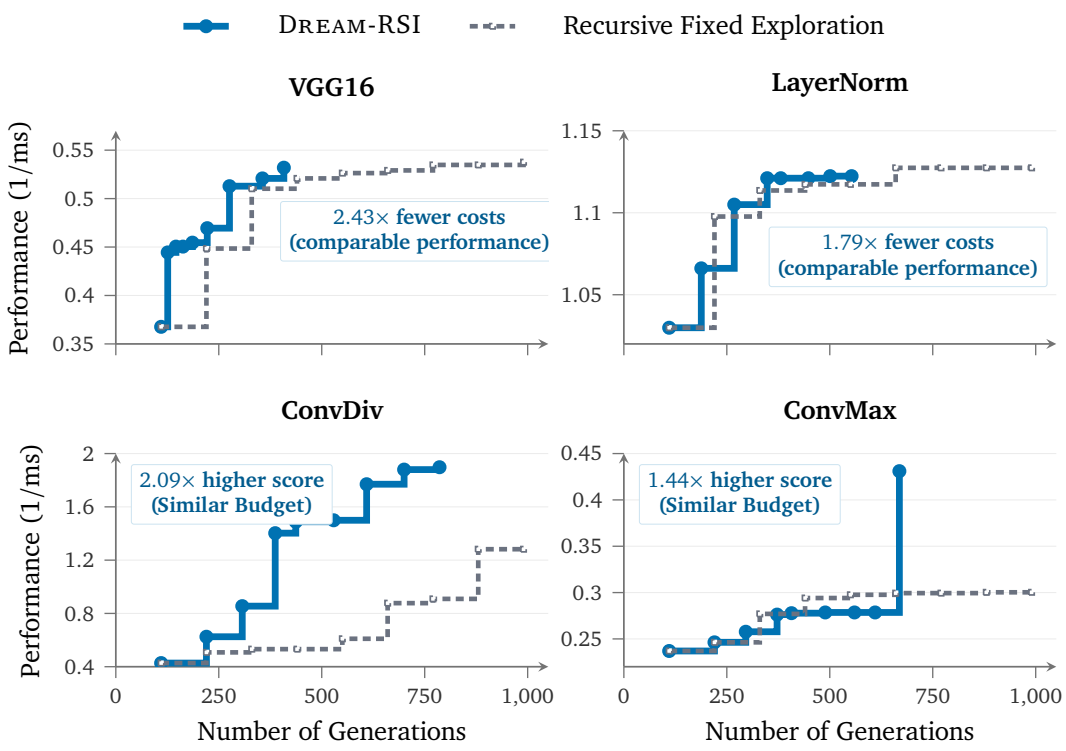


Figure 4 | **GPU kernel engineering results.** Discovery performance of DREAM-RSI and Recursive Fixed Exploration as a function of the number of generations. On VGG16 and LayerNorm, DREAM-RSI reaches comparable performance with 2.43× and 1.79× fewer generations, respectively. On ConvDiv and ConvMax, it achieves 2.09× and 1.44× higher performance under comparable discovery budgets. Higher is better for all tasks.

on mathematics optimization.

4.3. Kernel Engineering

We further evaluate DREAM-RSI on **GPU kernel engineering**, where the goal is to automatically discover high-performance implementations of kernels while preserving numerical correctness. Unlike mathematical discovery, kernel engineering requires reasoning jointly about algorithmic structure, memory access, parallelization, and hardware-specific optimizations, providing a substantially different testbed for evaluating whether our DREAM-RSI generalizes across discovery domains.

We consider four representative kernel-engineering tasks from KernelBench (Ouyang et al., 2025): **VGG16**, **LayerNorm**, **ConvDiv**, and **ConvMax**. Candidate implementations are evaluated by their execution performance, measured as inverse runtime (1/ms), subject to correctness checks against the reference implementation. We use Gemini-3.1 Pro as the coding agent and compare DREAM-RSI with Recursive Fixed Exploration under the same evaluation protocol and initialization.

Results. Figure 4 shows the discovery trajectories as the number of generations increases. On VGG16 and LayerNorm, DREAM-RSI reaches comparable final performance using 2.43× and 1.79× fewer generations, respectively. On ConvDiv and ConvMax, under comparable discovery budgets, DREAM-RSI achieves 2.09× and 1.44× higher performance, respectively. These results show that adapting the exploration policy across recursive rounds can improve the efficiency and effectiveness

of long-horizon discovery.

5. Further Analysis

5.1. 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 5, explicit directional guidance consistently underperforms its unguided counterpart across both paradigms under equivalent discovery budgets. These results suggest that in long-horizon discovery—where multiple parallel threads are deployed for exploration—imposing strong semantic inductive biases regarding future search directions tends to over-constrain the search space and impede diverse exploration.

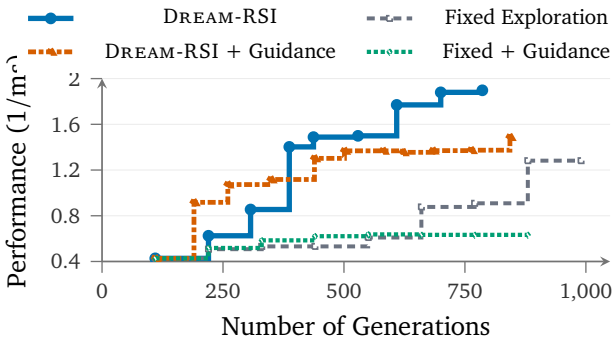


Figure 5 | Discovery performance on ConvDiv. Using history as an interactive replay simulator outperforms using it only as guidance.

5.2. Analysis of Evolution of Exploration Behavior

Figure 6 illustrates how the learned exploration policy evolves across recursive rounds on ConvDiv. As shown, the exploration policy exhibits a clear 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..

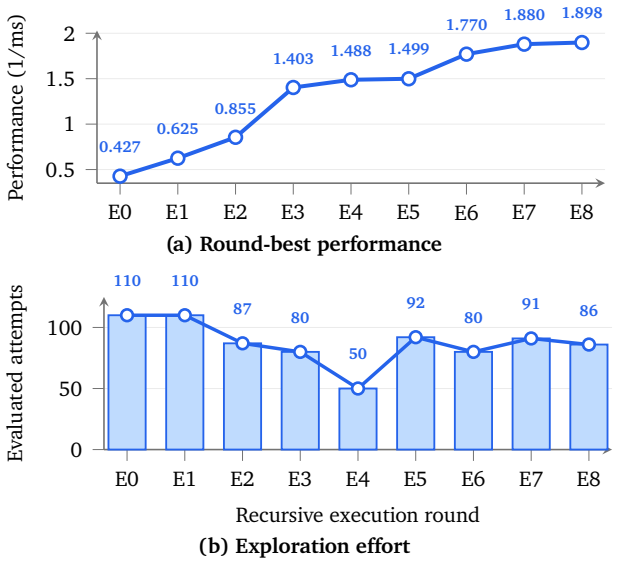


Figure 6 | **Evolution of exploration behavior on ConvDiv. (a)** Round-best performance across recursive execution rounds. **(b)** The number of evaluated attempts in each round.

6. Related Work

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 (Huang et al., 2026a,c), agent harnesses (Lee et al., 2026; Zhang et al., 2026b), contexts (Zhang et al., 2026d), skills (Ouyang et al., 2026a; Wu et al., 2026b; Zhang et al., 2026a), model behavior through test-time learning (Wang et al., 2025; Wu et al., 2026a; Yan et al., 2026a; Yuksekgonul et al., 2026), 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 (Kim et al., 2026; Liu et al., 2026a; Wang et al., 2026; Yan et al., 2026a; Zhang et al., 2026c). However, such meta-level strategies are difficult to improve because their quality is often revealed only after costly long-horizon rollouts. DREAM-RSI makes this meta-level optimization recursive and off-policy by turning accumulated discovery history into replay simulators, allowing exploration controllers to be repeatedly evaluated, improved, and redeployed without rerunning the underlying discovery process.

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 (Du et al., 2026; Virk et al., 2026); and other work improves how agents access and retain experience through evolving contexts, broader harness state, libraries, or skills (Lee et al., 2026; Ouyang et al., 2026a; Xu et al., 2026; Zhang et al., 2026d). 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 for improving exploration strategies.

7. Conclusion

We presented DREAM-RSI, a framework for recursive self-improvement of exploration in recursive self improvement. By converting accumulated discovery history from static context into an active, replayable simulator, DREAM-RSI addresses the core bottleneck of meta-optimization: delayed and expensive feedback, which is especially severe in long-horizon discovery settings. By 'dreaming' within replay simulators constructed from historical discovery trees, Dream-RSI evaluates candidate exploration policies rapidly and at negligible execution cost. The improved policies are then redeployed online to drive further discovery and expand the simulator pool, closing the recursive self-improvement loop. Across algorithm engineering, mathematical optimization, and GPU kernel engineering, DREAM-RSI achieves competitive or improved discovery quality while substantially reducing discovery cost in several settings.

References

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

H. Assumpção, D. Ferreira, L. Campos, and F. Murai. Codeevolve: an open source evolutionary coding agent for algorithmic discovery and optimization. *arXiv preprint arXiv:2510.14150*, 2025.

S. Cao, Z. Mao, J. E. Gonzalez, and I. Stoica. K-search: Llm kernel generation via co-evolving intrinsic world model. *arXiv preprint arXiv:2602.19128*, 2026.

R. Dai, K. Huang, C. Kang, and C. Liao. It takes two to match: Co-evolving generative retriever with reinforcement learning. *arXiv preprint arXiv:2609.00638*, 2026.

S. Du, X. Yan, J. Shi, Z. Cao, S. Feng, Z. Liang, B. Sun, T. Peng, Y. Zhou, X. Li, et al. Mlevolve: A self-evolving framework for automated machine learning algorithm discovery. *arXiv preprint arXiv:2606.06473*, 2026.

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

B. Georgiev, J. Gómez-Serrano, T. Tao, and A. Z. Wagner. Mathematical exploration and discovery at scale. *arXiv preprint arXiv:2511.02864*, 2025.

S. Gupta, J. Davidson, S. Levine, R. Sukthankar, and J. Malik. Cognitive mapping and planning for visual navigation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2616–2625, 2017.

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

D. Hafner, T. Lillicrap, J. Ba, and M. Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019.

D. Hafner, T. Lillicrap, M. Norouzi, and J. Ba. Mastering atari with discrete world models. *arXiv preprint arXiv:2010.02193*, 2020.

D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023.

D. Hafner, W. Yan, and T. Lillicrap. Training agents inside of scalable world models. *arXiv preprint arXiv:2509.24527*, 2025.

Y. Hu, S. Liu, Y. Yue, G. Zhang, B. Liu, F. Zhu, J. Lin, H. Guo, S. Dou, Z. Xi, et al. Memory in the age of ai agents. *arXiv preprint arXiv:2512.13564*, 2025.

C. Huang, H. Liu, T. Zheng, R. Dai, L. Huang, J. Li, Z. Li, Z. Wei, Y. Meng, and J. Huang. G-zero: Self-play for open-ended generation from zero data. *arXiv preprint arXiv:2605.09959*, 2026a.

C. Huang, Z. Wang, R. Han, J. Yan, Y. Chen, Z. CuiZhu, K. Jiang, P. Xia, H. Yu, Y. Zhuang, et al. Envharness: Awakening static worlds for agent learning. *arXiv preprint arXiv:2608.19880*, 2026b.

C. Huang, W. Yu, X. Wang, H. Zhang, Z. Li, R. Li, J. Huang, H. Mi, and D. Yu. R-zero: Self-evolving reasoning llm from zero data. In *International Conference on Learning Representations*, volume 2026, pages 130770–130790, 2026c.

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

J. Jiang, T. Ding, and Z. Zhu. Deltaevolve: Accelerating scientific discovery through momentum-driven evolution. *arXiv preprint arXiv:2602.02919*, 2026.

Z. M. Kim, Y.-J. Lee, S. Jwa, and D. Kang. Meta^n: Recursive self-improvement through emergent depth. *arXiv preprint arXiv:2608.24735*, 2026.

R. Lange, Y. Imajuku, and E. Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. In *International Conference on Learning Representations*, volume 2026, pages 74026–74078, 2026.

Y. Lee, R. Nair, Q. Zhang, K. Lee, O. Khattab, and C. Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

S. Liu, S. Agarwal, M. Maheswaran, M. Cemri, Z. Li, Q. Mang, A. Naren, E. Boneh, A. Cheng, M. Z. Pan, et al. Evox: Meta-evolution for automated discovery. *arXiv preprint arXiv:2602.23413*, 2026a.

S. Liu, M. Cemri, S. Agarwal, A. Krentsel, A. Naren, Q. Mang, Z. Li, A. Gupta, M. Maheswaran, A. Cheng, M. Pan, E. Boneh, K. Ramchandran, K. Sen, M. Zaharia, A. G. Dimakis, and I. 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, pages 1223–1227. Association for Computing Machinery, 2026b. doi: 10.1145/3786335.3813221. URL https://doi.org/10.1145/3786335.3813221.

S. Liu, Z. Lin, Y. Zhang, Y. Ren, Y. Wu, Y. Li, Z. Wang, Z. Fu, and J. 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.

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

A. Novikov, N. Vũ, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. Ruiz, A. 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/, Sept. 2026. Accessed: 2026-09-10.

A. Ouyang, S. Guo, S. Arora, A. L. Zhang, W. Hu, C. Ré, and A. Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

S. Ouyang, J. Yan, Y. Chen, R. Han, Z. Wang, B. D. Mishra, R. Meng, C.-L. Li, Y. Jiao, K. Zha, et al. Skillos: Learning skill curation for self-evolving agents. *arXiv preprint arXiv:2605.06614*, 2026a.

S. Ouyang, J. Yan, I. Hsu, Y. Chen, K. Jiang, Z. Wang, R. Han, L. Le, S. Daruki, X. Tang, et al. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, volume 2026, pages 94327–94354, 2026b.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

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

R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In B. Porter and R. Mooney, editors, *Machine Learning Proceedings 1990*, pages 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.

Y. Virk, Z. Edds, C. S. Xia, and L. Zhang. Swarmresearch: Orchestrating coding agents for open-ended discovery. *arXiv preprint arXiv:2607.02807*, 2026.

Y. Wang, S.-R. Su, Z. Zeng, E. Xu, L. Ren, X. Yang, Z. Huang, X. He, L. Ma, B. Peng, et al. Thetaevolve: Test-time learning on open problems. *arXiv preprint arXiv:2511.23473*, 2025.

Z. Wang, M. Yan, J. Bi, S. Yan, V. Tresp, and Y. Ma. Metaskill-evolve: Recursive self-improvement of llm agents via two-timescale meta-skill evolution. *arXiv preprint arXiv:2607.05297*, 2026.

S. Wu, C. Qian, X. Chen, and H. Ji. Teaching llms to self-evolve: Cultivating core meta-skills with reinforcement learning. *arXiv preprint arXiv:2607.21971*, 2026a.

X. Wu, Y. Zhuan, R. Wei, H. Chen, D. Bai, J. Liu, X. Wang, X. Wang, L. Wang, and X. Cheng. Agenticrectune: Multi-agent with self-evolving skillhub for recommendation system optimization. *arXiv preprint arXiv:2604.26969*, 2026b.

T. Xiong, Z. Yang, X. Wang, C.-C. Lin, R. Ma, K. Lin, Z. Wang, L. Li, C. Liu, R. Chen, et al. Rubrics as visual-repair context for self-evolving ui-to-code generation. *arXiv preprint arXiv:2608.24138*, 2026.

W. Xu, A. Sordoni, C. Singh, Z. Gero, M. Galley, X. Yuan, and J. Gao. Test-time learning with an evolving library. *arXiv preprint arXiv:2605.14477*, 2026.

M. Yan, B. Peng, B. Coleman, Z. Chen, Z. Xie, S. Chen, Z. He, N. Sachdeva, W. Wang, E. H. Chi, et al. Pacevolve++: Improving test-time learning for evolutionary search agents. *arXiv preprint arXiv:2605.07039*, 2026a.

M. Yan, B. Peng, B. Coleman, Z. Chen, Z. Xie, S. Chen, Z. He, N. Sachdeva, I. Ye, W. Wang, et al. Pacevolve: Enabling long-horizon progress-aware consistent evolution. *arXiv preprint arXiv:2601.10657*, 2026b.

H. Ye, H. Lin, J. Tang, Y. Luo, C. Yang, C. Su, R. Thapa, R. Yang, R. Liu, Z. Li, et al. Evaluation-driven scaling for scientific discovery. *arXiv preprint arXiv:2604.19341*, 2026.

M. Yuksekgonul, D. Koceja, X. Li, F. Bianchi, J. McCaleb, X. Wang, J. Kautz, Y. Choi, J. Zou, C. Guestrin, et al. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*, 2026.

H. Zhang, S. Fan, H. P. Zou, Y. Chen, Z. Wang, J. Zhou, C. Li, W.-C. Huang, Y. Yao, K. Zheng, et al. Coevoskills: Self-evolving agent skills via co-evolutionary verification. *arXiv preprint arXiv:2604.01687*, 2026a.

J. Zhang, S. Hu, C. Lu, R. Lange, and J. Clune. Darwin gödel machine: open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, volume 2026, pages 104223–104294, 2026b.

J. Zhang, B. Zhao, W. Yang, J. Foerster, J. Clune, M. Jiang, S. Devlin, and T. Shavrina. Hyperagents. *arXiv preprint arXiv:2603.19461*, 2026c.

Q. Zhang, C. Hu, S. Upasani, B. Ma, F. Hong, V. Kamanuru, J. Rainton, C. Wu, M. Ji, H. Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, volume 2026, pages 86069–86100, 2026d.

T. Zheng, H. Liu, C. Huang, H. Bao, S. Zhang, R. Liu, R. Dai, R. Chen, C. Liu, T. Xiong, et al. Llms improving llms: Agentic discovery for test-time scaling. *arXiv preprint arXiv:2605.08083*, 2026.

A. 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

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

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

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

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

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

B. 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.

B.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.
```

B.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.
```

```
234
235 Include a short, factual ‘‘reason‘‘ in every plan. ‘‘plan_grid‘‘ answers
236 how many directions to make available; the direction provider assigns those new
237 roots their directions, and ‘‘solve‘‘ still decides which legal roots/frontiers to
238 open, refine, prune, or stop. Do not choose roots merely because their branch id is
239 small. The runtime grid is the hard bound: controller thresholds may use less, but
240 can never create branches or attempts beyond the effective plan. Before finishing,
241 verify that the edited ‘‘method.py‘‘ contains an override of ‘‘plan_grid‘‘ that
242 returns ‘‘GridPlan(branch_count=..., refine_count=..., reason=...)‘‘ on every path.
243
244 ## Learn from history without leaking outcomes
245
246 Earlier rounds are in ‘‘{history_dir}/r####_*/‘‘. Read their policy code and
247 ‘‘proposal_results/beta_sweep.json‘‘. Start from a strong recent policy, retain
248 mechanisms that raised ‘‘pareto.reward‘‘, and make a concrete change when progress
249 stalls. A legacy AUC-only sweep is useful code history but is not numerically
250 comparable to the current reward. The baseline under ‘‘{history_dir}/baseline/‘‘ is
251 a parallel-refine floor to beat.
252
253 Each current-objective round also archives
254 ‘‘proposal_results/policy_execution_traces.jsonl‘‘: one replay episode per
255 ‘‘(frozen trace, beta)‘‘. Use it to diagnose general behavior -- serial batches,
256 premature stops, over-pruning, or wasted probes -- from the prefix state, selected
257 batch, and revealed outcomes at each decision round. It is **between-round feedback
258 only**: never read it inside ‘‘solve()‘‘, and never copy a trace-specific branch,
259 cell id, score, or target into policy logic.
260
261 ‘‘{trace_pool}‘‘, if present, may be read only outside ‘‘solve()‘‘. Prefer the
262 ‘‘live_cycle_manifest.json‘‘ sidecars over raw replay outcomes for the per-iteration
263 live trend. Never copy trace scores, targets, or cell ids into policy logic.
264
265 ## Deliverable
266
267 Write a complete adaptive policy in ‘‘{method_file}‘‘. Include a short module
268 docstring describing its prefix signals, batch rule, beta schedule, default-beta
269 rationale, grid-planning rule (if implemented), and safeguards against
270 over-pruning, over-stopping, permanent starvation after repairable failures, and
271 serial probes. Before finishing, verify trajectory-based ranking, the stated
272 success semantics, non-automatic zero-valid closure, deterministic recovery
273 competition, and portfolio-level stop.
```

C. Discovered Programs

We provide the complete implementation of the Lasso-path solver discovered by DREAM-RSI. As discussed in Section 4.1, 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.

```
1  # EVOLVE-BLOCK-START
2
3  CPP_CODE = r'''
4  #define EIGEN_NO_DEBUG
5  #define EIGEN_MPL2_ONLY
6  #define EIGEN_UNROLL_LOOPS
7
8  #include <Eigen/Dense>
9  #include <vector>
10 #include <cstdio>
11 #include <cmath>
12 #include <algorithm>
13 #include <numeric>
14 #include <omp.h>
15 #include <cstdlib>
16 #include <cstring>
```

```
17
18 using Eigen::MatrixXd;
19 using Eigen::VectorXd;
20
21 #if defined(_MSC_VER)
22 #define RESTRICT __restrict
23 #elif defined(__GNUC__) || defined(__clang__)
24 #define RESTRICT __restrict__
25 #else
26 #define RESTRICT
27 #endif
28
29 // High-performance alignment assumption
30 #if defined(__GNUC__) || defined(__clang__)
31 #define ASSUME_ALIGNED(ptr, alignment) (double*)__builtin_assume_aligned((ptr), (alignment))
32 #else
33 #define ASSUME_ALIGNED(ptr, alignment) (ptr)
34 #endif
35
36 // High-performance branch-free soft-thresholding using std::abs and std::copysign
37 static inline double soft_thresh(double z, double gamma) {
38     double abs_z = std::abs(z);
39     double val = abs_z - gamma;
40     return std::copysign(val > 0.0 ? val : 0.0, z);
41 }
42
43 // ============================================================================
44 // DISJOINT-PARTITION ACTIVE-SET LASSO PATH SOLVER WITH ALIGNED COLUMN PADDING
45 // ============================================================================
46 void solve_active_set(
47     const double* RESTRICT X_padded,
48     int n_padded,
49     int n,
50     int p,
51     const VectorXd& y,
52     const VectorXd& lam_path,
53     const VectorXd& xv,
54     const VectorXd& grad_init,
55     MatrixXd&       coef_path,  // (p, n_lam) output, pre-zeroed
56     double thresh,              // convergence threshold
57     int    maxit)               // max inner loop iterations
58 {
59     const double fn = static_cast<double>(n);
60     const double inv_fn = 1.0 / fn;
61     const double tol = thresh;
62     const int nlam = lam_path.size();
63
64     // Workload-Aware flag for activating Cauchy-Schwarz KKT Pruning
65     const bool use_cs = (p >= 500 && n >= 150);
66
67     // Initial capacity for active set structures - optimized to completely avoid reallocations on
       almost all problems
68     int current_capacity = ((std::max(128, std::min(512, p)) + 7) / 8) * 8;
69
70     // Declare raw pointers for 64-byte aligned structures
71     double* G_data = nullptr;
72     double* c_data = nullptr;
73     double* beta_active_data = nullptr;
74     double* xv_active_data = nullptr;
75     double* inv_xv_active_data = nullptr;
76     double* grad_init_active_data = nullptr;
77     double* beta_old_at_start = nullptr;
78
79     double* y_padded = nullptr;
80     double* r_padded = nullptr;
81     double* r_ref_padded = nullptr;
82
83     bool oom = false;
84
85     // 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;
```

```
535
536                     if (num_to_compute > 0.3 * p || lambdas_since_reset >= 8) {
537                         // Drift is too large or reset interval reached, do a full reset (SIMD 4x
    Register-Blocked)
538                         const bool run_parallel_reset = (static_cast<size_t>(n_padded) *
    unscreened_size >= 150000);
539                         #pragma omp parallel for schedule(static) if(run_parallel_reset)
540                         for (int i = 0; i < (unscreened_size / 4) * 4; i += 4) {
541                             int j0 = unscreened_list[i];
542                             int j1 = unscreened_list[i+1];
543                             int j2 = unscreened_list[i+2];
544                             int j3 = unscreened_list[i+3];
545
546                             const double* RESTRICT col0 = ASSUME_ALIGNED(X_padded + j0 * n_padded,
     64);
547                             const double* RESTRICT col1 = ASSUME_ALIGNED(X_padded + j1 * n_padded,
     64);
548                             const double* RESTRICT col2 = ASSUME_ALIGNED(X_padded + j2 * n_padded,
     64);
549                             const double* RESTRICT col3 = ASSUME_ALIGNED(X_padded + j3 * n_padded,
     64);
550                             const double* RESTRICT r_ptr_exact = ASSUME_ALIGNED(r_padded, 64);
551
552                             double sum0 = 0.0, sum1 = 0.0, sum2 = 0.0, sum3 = 0.0;
553                             #pragma omp simd reduction(+:sum0, sum1, sum2, sum3) aligned(
    r_ptr_exact, col0, col1, col2, col3: 64)
554                             for (int k = 0; k < n_padded; ++k) {
555                                 double rk = r_ptr_exact[k];
556                                 sum0 += rk * col0[k];
557                                 sum1 += rk * col1[k];
558                                 sum2 += rk * col2[k];
559                                 sum3 += rk * col3[k];
560                             }
561                             grad_ptr[j0] = sum0 * inv_fn;
562                             grad_ptr[j1] = sum1 * inv_fn;
563                             grad_ptr[j2] = sum2 * inv_fn;
564                             grad_ptr[j3] = sum3 * inv_fn;
565                         }
566
567                         for (int i = (unscreened_size / 4) * 4; i < unscreened_size; ++i) {
568                             int j = unscreened_list[i];
569                             const double* RESTRICT col_ptr = ASSUME_ALIGNED(X_padded + j *
    n_padded, 64);
570                             const double* RESTRICT r_ptr_exact = ASSUME_ALIGNED(r_padded, 64);
571                             double sum = 0.0;
572                             #pragma omp simd reduction(+:sum) aligned(r_ptr_exact, col_ptr: 64)
573                             for (int k = 0; k < n_padded; ++k) {
574                                 sum += r_ptr_exact[k] * col_ptr[k];
575                             }
576                             grad_ptr[j] = sum * inv_fn;
577                         }
578
579                         std::memcpy(r_ref_padded, r_padded, n_padded * sizeof(double));
580
581                         double* RESTRICT grad_ref_ptr_writable = grad_ref.data();
582                         #pragma omp parallel for schedule(static) if(unscreened_size >= 2048)
583                         for (int i = 0; i < unscreened_size; ++i) {
584                             int j = unscreened_ptr[i];
585                             grad_ref_ptr_writable[j] = grad_ptr[j];
586                         }
587                         lambdas_since_reset = 0;
588                         did_reset = true;
589                     } else {
590                         // Compute exact gradients only for the tiny unpruned subset (SIMD 4x
    Register-Blocked)
591                         const bool run_parallel_comp = (num_to_compute >= 32 && static_cast<size_t
    >(n_padded) * num_to_compute >= 150000);
592                         #pragma omp parallel for schedule(static) if(run_parallel_comp)
593                         for (int k = 0; k < (num_to_compute / 4) * 4; k += 4) {
594                             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
```