OPTSCIENTIST: TOWARD RECURSIVE SELF-IMPROVING AGENTS VIA AUTOMATED OPTIMIZER DISCOVERY

Anonymous authors
Paper under double-blind review

ABSTRACT

Designing optimizers for modern deep learning remains a challenging scientific problem, requiring the joint consideration of optimization geometry, state dynamics, numerical stability, implementation constraints, and empirical generalization. Existing automated discovery methods typically search either over unconstrained code spaces or within narrowly parameterized optimizer families, trading flexibility for validity and interpretability. We introduce OPTScientist, a theory-guided multi-agent framework that casts optimizer discovery as a collaborative scientific process. Specialized agents propose hypotheses, instantiate candidate optimizers in an extensible typed DSL, and evaluate them through transformer pretraining, while a shared knowledge system accumulates candidate lineages, experimental outcomes, failure modes, and reusable lessons to guide subsequent search. The DSL provides compiler-checked validity while supporting conservative extensions when existing primitives limit promising hypotheses. Combining multi-fidelity evaluation, cross-scale validation, and controlled interventions, OPTScientist identifies proxy hacking and translates diagnosed failures into executable constraints. The system discovers eighteen optimizer designs that achieve lower target-scale validation bpb than all baselines in our main benchmark, with selected designs also improving downstream performance and transferring to vision after learning-rate retuning. These results position OPTScientist as a concrete step toward recursive self-improvement, where experimental evidence refines the knowledge, search space, and constraints that guide subsequent discovery.

1 INTRODUCTION

Optimizers determine how neural networks use gradients, shaping training efficiency, stability, memory requirements, and generalization (Kingma & Ba, 2014; Loshchilov & Hutter, 2019). Designing them for large Transformers requires balancing update geometry, state dynamics, numerical safeguards, and implementation cost. Recent advances explore low-rank gradients, curvature information, structured preconditioning, and matrix-normalized updates (Zhao et al., 2024; Liu et al., 2023; Vyas et al., 2024; Zhang et al., 2024; Liu et al., 2025a). Yet a promising update rule must also survive changes in model scale and training protocol: improvements measured in a small experiment need not persist in the intended deployment setting (Wen et al., 2025).

Automated discovery offers a way to explore this design space systematically. Evolutionary and symbolic search have produced learning algorithms and practical optimizers (Real et al., 2020; Bello et al., 2017; Chen et al., 2023), while language-model agents can propose hypotheses, implement experiments, and reuse their outcomes (Romera-Paredes et al., 2024; Novikov et al., 2025; Lu et al., 2024). These capabilities make iterative optimizer research increasingly feasible. However, evaluating every proposal with full-scale pretraining remains prohibitively expensive, making small-model and short-horizon proxies an essential part of the discovery process.

This dependence introduces a failure mode that stronger search alone does not resolve. In our experiments, candidate optimizers repeatedly improve proxy scores while losing their advantage at the target scale. Several failures share related amplitude-modulation mechanisms. When the system records their proxy success as reusable experience, subsequent proposals can concentrate

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

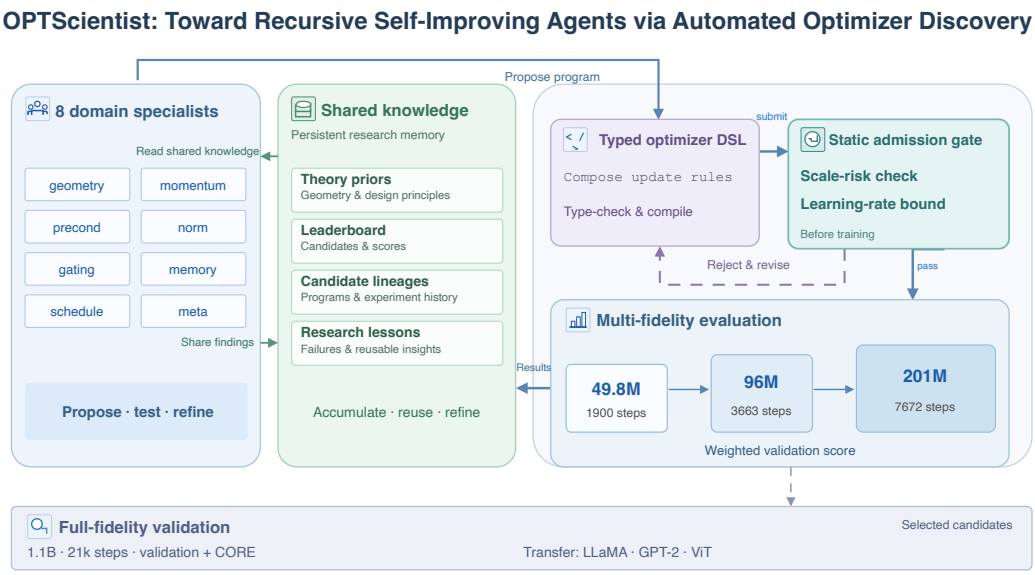


Figure 1: **Overview of OPTSCIENTIST.** Eight domain specialists share research knowledge, propose optimizer hypotheses, and submit programs through an extensible typed DSL. Compilation and static admission checks precede a multi-fidelity pretraining cascade. Results, lineages, and lessons are written back to guide subsequent proposals. Full-fidelity validation establishes transfer and informs later system generations; it does not enter the online ranking of the same proxy-search run.

on the same fragile designs. We call this phenomenon *proxy hacking*: search exploits a mismatch between the surrogate evaluation and the intended target while following the experimental protocol. The challenge is therefore to improve exploration while learning which experimental lessons remain valid beyond the proxy.

We introduce OPTSCIENTIST, a theory-guided multi-agent framework that addresses this challenge through three generations of optimizer research. Generation 1 combines a single proposer with persistent experience and exposes recurrent proxy-to-target reversals. Generation 2 introduces domain specialists and explicit failure lessons, broadening exploration but leaving related failures accessible through different program expressions. Generation 3 introduces an extensible typed domain-specific language (DSL), making candidate structure available to static checks, together with a separately calibrated learning-rate constraint. The first two generations use code-based candidates; the DSL and admission gate are introduced only in the third. In the final system, eight specialists share candidate lineages, experimental outcomes, and lessons; propose optimizer mechanisms; and evaluate admitted programs through a multi-fidelity pretraining cascade (Figure 1). Cross-scale validation and controlled interventions connect specific failures to program dependencies and hyperparameter choices. Supported diagnoses become executable admission rules, so experience changes both what the agents propose and what receives training resources. Conservative DSL extensions allow the expressive space to grow when existing primitives limit a promising hypothesis.

The system discovers multiple optimizers with strong target-scale validation and downstream performance. We evaluate this research output separately from proxy-search efficiency and gate discrimination: a higher search score does not by itself establish better transfer, and passing a static check does not guarantee a better optimizer. Our contributions are:

- We identify proxy hacking in optimizer autoresearch and investigate recurring failures through cross-scale evaluation and controlled program interventions.
- We develop a complete multi-agent research system that combines shared knowledge, an extensible typed DSL, multi-fidelity evaluation, and executable constraints derived from diagnosed failures.
- We discover multiple competitive optimizers and evaluate their target-scale performance, repeated-training behavior, and transfer across model configurations.

2 Related Work

Optimizers for Transformer pretraining. Adam and AdamW provide widely used adaptive update rules (Kingma & Ba, 2014; Loshchilov & Hutter, 2019). Adafactor, GaLore, and Adam-mini explore different ways to reduce optimizer memory requirements (Shazeer & Stern, 2018; Zhao et al., 2024; Zhang et al., 2024). Sophia incorporates curvature information, SOAP combines Shampoo-style preconditioning with Adam, and Muon uses matrix-structured updates (Liu et al., 2023; Vyas et al., 2024; Liu et al., 2025a). MARS, AdEMAMix, and COSMOS further explore variance reduction, momentum design, and hybrid adaptive optimization (Yuan et al., 2024; Pagliardini et al., 2024; Liu et al., 2025b). These methods motivate a rich space of mechanisms whose behavior depends on the training protocol. Empirical optimizer comparisons underscore the importance of evaluation conditions (Wen et al., 2025). Our contribution is a research process that explores such mechanisms and diagnoses failures of proxy-based selection, producing several useful optimizers rather than centering the framework on one manually specified rule.

Learned optimizers and programmatic discovery. Learned optimizers train parameterized update rules through meta-learning (Andrychowicz et al., 2016). Celo explores compute-efficient training of learned optimizers (Moudgil et al., 2025).A complementary approach searches explicit programs. Genetic programming provides an evolutionary foundation (Koza, 1994), neural optimizer search uses reinforcement learning (Bello et al., 2017), and AutoML-Zero and subsequent evolutionary work search learning algorithms or optimizer rules (Real et al., 2020; Marfinetz, 2025). Symbolic optimizer discovery produced Lion and explicitly considered proxy-to-target generalization (Chen et al., 2023). We build on that concern by examining how persistent agent experience can reinforce proxy-specific mechanisms. Our final system represents candidates as inspectable typed programs and turns diagnosed failures into admission constraints; it does not meta-train a neural update function or claim that proxy generalization is a previously unrecognized issue.

Language-model-guided optimization and program search. Large language models can use evaluation feedback to generate improved solutions (Yang et al., 2023). Evolution of Heuristics and ReEvo combine language-model proposals with evolutionary search and reflection (Liu et al., 2024; Ye et al., 2024). FunSearch and AlphaEvolve extend evaluated program search to mathematical, scientific, and algorithmic discovery (Romera-Paredes et al., 2024; Novikov et al., 2025). These approaches establish the value of iterative generation and evaluation. OPTScientist focuses on the additional difficulty created by a training-based surrogate: candidate programs can be valid and improve the measured objective while failing at the target scale. Its distinctive mechanism is the connection from transfer failures, through controlled diagnosis, to constraints on future search.

Scientific agents, collaboration, and memory. Toolformer and ReAct develop tool use and reasoning–action interfaces (Schick et al., 2023; Yao et al., 2022); Reflexion and Voyager illustrate how feedback and accumulated experience can support subsequent behavior (Shinn et al., 2023; Wang et al., 2023). AutoGen, CAMEL, and MetaGPT organize collaboration among specialized agents (Wu et al., 2024; Li et al., 2023; Hong et al., 2023). Scientific applications span autonomous chemistry, literature-based idea generation, multi-agent scientific reasoning, and research assistance (Boiko et al., 2023; Baek et al., 2025; Ghafarollahi & Buehler, 2025; Lu et al., 2024; Schmidgall et al., 2025). Our contribution is not specialization or memory alone. We investigate when shared experience amplifies an evaluation bias, distinguish natural-language lessons from executable restrictions, and integrate domain specialists with candidate provenance, typed implementation, and cross-scale confirmation.

3 System Evolution

3.1 Generation 1: Experience Amplifies the Proxy's Artifacts

Generation 1 uses a single research agent to propose and implement optimizers directly in code. A knowledge system stores outcomes, candidate relationships, and lessons, while proxy scores guide subsequent exploration. This generation has neither DSL instantiation nor a static transfer-risk gate.

Let u denote a candidate, u_0 denote Muon, and $\ell_f(u)$ denote validation loss at fidelity f. We define improvement over the reference as

$$\Delta_f(u) = \ell_f(u_0) - \ell_f(u). \tag{1}$$

Positive values indicate better performance under the same protocol. A proxy-to-target reversal occurs when

$$\Delta_{\text{proxy}}(u) > 0, \qquad \Delta_{\text{full}}(u) < 0. \tag{2}$$

Here, proxy loss refers to a particular evaluation stage; the aggregate search score is defined in Section 4.4.

Although proxy performance improves, candidate histories reveal recurring scale-dependent modifications whose gains do not reliably transfer. Such mechanisms can modulate update magnitudes through parameter norms, gradient statistics, or explicit dimension-dependent factors. Trust-ratio-style modulation is one example: it uses parameter and normalized-update norms to determine step magnitude. The concern is not the presence of a norm ratio itself, but whether the resulting scaling remains appropriate beyond the proxy setting. Across representative candidates, small-model gains can shrink as evaluation fidelity increases and reverse only beyond the largest proxy stage (Figure 2). Passing several proxy stages therefore does not establish target-scale superiority.

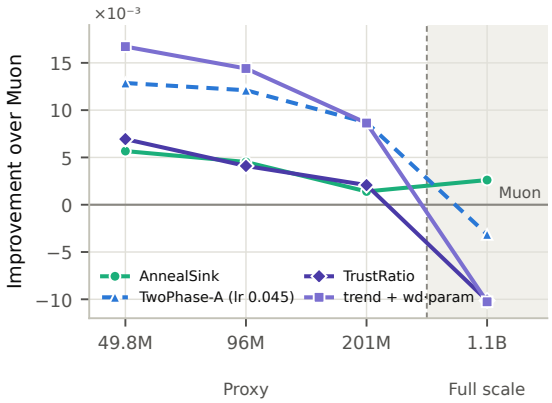


Figure 2: **Proxy gains can reverse at full scale.** Improvement is measured as Muon minus candidate validation bpb; positive values indicate better performance. All four representative candidates outperform Muon at the proxy stages, but only AnnealSink retains its advantage at full scale. Shading marks full-fidelity evaluation.

Candidate histories suggest a reinforcing process: high proxy scores enter the leaderboard and shared lessons, encouraging further exploration of related mechanisms. Different program expressions can exploit the same proxy-specific scaling behavior, so the resulting bias need not concentrate on a single formula or optimizer family. When locally effective modifications are recorded as broadly reusable improvements, accumulating successful proxy experiments can reinforce fragile design choices. These observations motivate attaching applicability conditions to learned experience and examining the underlying scale dependencies rather than only the surface form of an update rule.

3.2 Generation 2: Specialists for Diversity, Experience as Soft Knowledge

Generation 2 introduces domain specialists with independent contexts and a shared knowledge system. It also converts Generation 1 failures into explicit lessons about parameter dependence and update scaling. Candidates remain code-based. Specialization broadens the explored structures, and several designs retain their gains at full fidelity. Yet natural-language lessons do not consistently constrain an entire mechanism class.

One new proxy winner adds a parameter-dependent term to the final update. Although it avoids the earlier trust-ratio expression, it again changes the relationship between update and parameter magnitudes and degrades at the target scale. Other candidates obtain proxy gains with learning rates that do not transfer to longer target training. These are distinct issues: one concerns program structure, the other hyperparameter calibration. Together, they motivate executable constraints that act independently of the wording of past lessons.

3.3 Generation 3: Experience Compiled into Constraints

3.3.1 Criterion and Rationale

Generation 3 introduces a typed DSL that exposes candidate structure before training. For the family of normalized updates analyzed here, we separate reference shape scaling, candidate amplitude modulation, and explicit weight decay:

$$\Delta W = -\eta c(m,n)\mu(G,W,S;m,n)D(G,W,S) - \eta\gamma W. \tag{3}$$

Here, $W \in \mathbb{R}^{m\times n}$ is a parameter matrix, G its gradient, S the optimizer state, and D a normalized direction. The factor $c(m,n)$ specifies shape scaling in the reference protocol, μ is candidate-dependent scalar modulation, and γ is the explicit weight-decay coefficient. Row-wise modulation instead takes the form $\mathrm{Diag}(\boldsymbol{\mu})D$. This decomposition describes the analyzed family, not every possible optimizer program.

Consider positive rescalings $W \mapsto aW$ and $G \mapsto bG$, together with corresponding transformations of optimizer states. Width scaling $(m,n) \mapsto (\lambda m, \lambda n)$ additionally requires a specified initialization and matrix-statistics model. For homogeneous expressions, the scaling signature $\mathbf{d}(\mu) = (g,p,L)$ describes

$$\mu(T_{a,b,\lambda}z) = b^g a^p \lambda^L \mu(z), \qquad z = (G,W,S;m,n). \tag{4}$$

Under these independent scaling transformations, a zero signature denotes invariant modulation, while a nonzero signature identifies scale sensitivity. Multiplication and division respectively add and subtract exponents, enabling compositional analysis of candidate expressions.

Scale sensitivity alone does not imply transfer failure. Factors may cancel under a particular parameterization, and amplitude changes may compensate for changes in the model. For example, a pure trust-ratio update, considered without additional shape scaling or weight decay, satisfies

$$\mu_{\mathrm{trust}} = \frac{\|W\|_F}{\|D\|_F}, \qquad \frac{\|-\eta\mu_{\mathrm{trust}}D\|_F}{\|W\|_F} = \eta, \tag{5}$$

for nonzero W and D and without numerical stabilizers. Its failure therefore cannot be attributed to inevitable drift in the relative Frobenius step. Instead, we ask whether a candidate changes calibration relative to the reference update under the target protocol, and whether controlled experiments identify that change as harmful.

Normalization also does not remove all shape dependence:

$$\|X\|_F = \sqrt{mn}\,\mathrm{rms}(X), \qquad \|D\|_F = \sqrt{\min(m,n)} \tag{6}$$

when D is a full-rank semi-orthogonal matrix. The checker's scale tags are thus an abstraction for identifying risky dependency paths, rather than a proof of equivalent training dynamics. Numerical stabilizers and fixed-threshold nonlinearities further limit exact homogeneity. Likewise, parameter-dependent terms can implement legitimate weight decay; our restriction targets additional terms that conflict with the calibrated training protocol. We therefore treat scale-risk screening and learning-rate calibration as separate constraints.

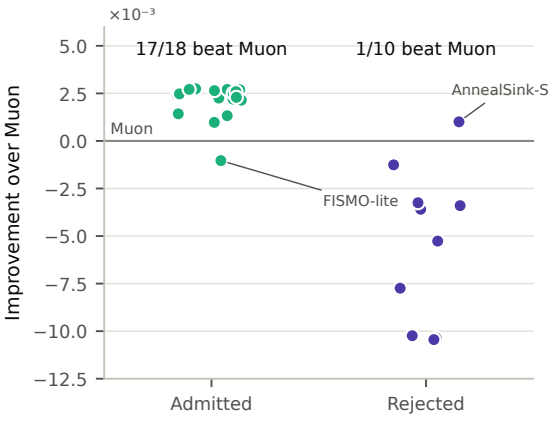


Figure 3: **Retrospective gate analysis.** Each point represents a fully evaluated optimizer. Improvement is measured as matched-seed Muon minus candidate validation bpb at full scale.

3.3.2 Checker Calibration and Mechanistic Validation

The checker traces parameter and gradient dependencies into the final update and separately checks the declared learning rate against the calibrated range. Absolute-step

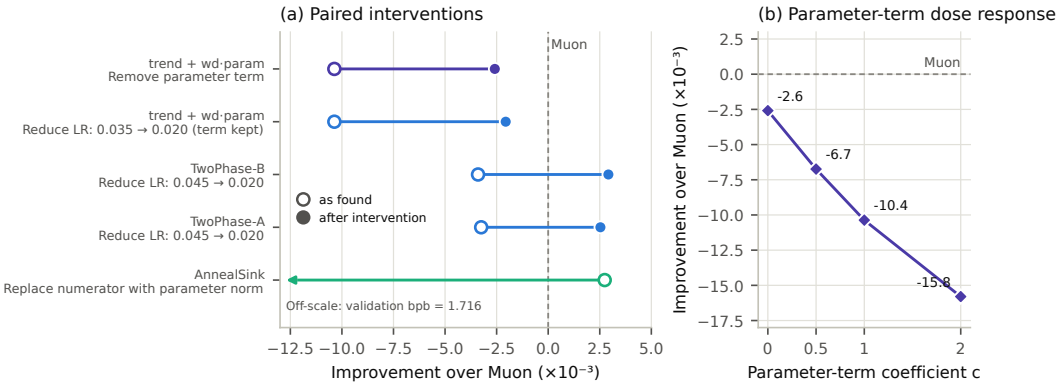


Figure 4: **Controlled interventions support the failure diagnoses.** Improvement is Muon minus candidate validation bpb at full scale; positive values favor the candidate. **(a)** Open and filled circles show results before and after intervention. Removing the additional parameter term or reducing the learning rate improves performance; introducing parameter-norm dependence into AnnealSink causes severe degradation (off-scale arrow). **(b)** Increasing the additional term's coefficient worsens performance with learning rate fixed at 0.035 and trainer-side weight decay unchanged. All experiments use the same training seed.

schedules trigger warnings rather than rejection, as retrospective evidence does not support their use as a reliable discriminator. Across 28 fully evaluated designs, 17 of 18 admitted candidates outperform Muon, compared with 1 of 10 rejected candidates (Figure 3). FISMO-lite passes but fails to improve, whereas AnnealSink-S improves despite a learning rate of 0.025, above the calibrated limit of 0.02. The gate therefore provides useful but imperfect separation. Because historical failures informed its rules, this retrospective analysis is a calibration diagnostic, not an independent test of generalization.

Controlled interventions further support the diagnoses (Figure 4). Removing an additional parameter term or reducing an excessive learning rate improves target-scale performance. For the parameter-term candidate, either change alone leaves a gap to Muon, while increasing the term's coefficient progressively worsens validation performance. Conversely, replacing AnnealSink's amplitude-correction numerator with the parameter norm causes severe degradation. These results connect specific program dependencies and hyperparameter choices to observed failures, without implying that all parameter-dependent optimizers are flawed.

4 Method

The final system integrates asynchronous specialist research, program constraints, staged evaluation, and knowledge updates (Figure 1). Each hypothesis is linked to an inspectable implementation, a traceable lineage, and comparable experimental evidence.

4.1 Eight Domain Specialists and Shared Knowledge

Eight specialists focus on geometry, momentum, preconditioning, normalization, gating, memory, scheduling, and meta-design. Each expert combines domain-specific prior knowledge with the common knowledge maintained by the research system. These specializations provide complementary perspectives rather than disjoint operator sets.

Within each run, the eight experts collaborate asynchronously. Each independently carries out a research cycle: consulting available knowledge, formulating a hypothesis, implementing a candidate, obtaining evaluation results, and recording lessons. Shared records allow findings from one expert to inform subsequent work by the others without requiring synchronized research rounds. Different runs start from the same initial conditions, including the specialist priors, shared knowledge, reference candidates, and evaluation protocol.

Six tools support this workflow: `read_snapshot` retrieves research state; `rebase_to` selects an existing candidate as a starting point; `append_lesson` records findings; `compile_dsl` checks

programs; `submit_candidate` requests evaluation; and `propose_dsl_extension` proposes new expressive capabilities.

The knowledge system maintains common research priors, run state, and accumulated lessons. Run state includes leaderboards, candidate lineages, and snapshots. Lineages record parent candidates, modifications, motivations, and outcomes, supporting attribution of both improvements and failures. Lessons document effective changes, failure stages, and the scope of supporting evidence. Together, these records guide refinement and reduce repeated failures. Reproducible diagnoses can inform executable constraints in subsequent system generations.

4.2 Dimension-Typed, Extensible Optimizer DSL

The DSL represents updates as programs over gradients, parameters, and explicitly declared persistent state. Compilation checks tensor shapes, state usage, and operator interfaces before training. Static state accounting also characterizes memory requirements. Alongside structural types, the system propagates abstract scaling signatures. Products and ratios combine exponents, square roots rescale them, and compatible additions preserve signatures. Normalization and similarity operators follow their declared behavior. Incompatible mixtures and dimensional inputs to fixed-threshold nonlinearities generate risk reports. Structural validity and scale-risk analysis remain separate: a compilable program can still contain a risky dependency. This representation supports controlled edits to normalization, amplitude correction, and schedules while preserving state interfaces and output shapes. When existing primitives cannot express a hypothesis, experts can propose extensions specifying implementations, interfaces, state requirements, and scale behavior. Accepted extensions enter a versioned compiler environment, to which candidates and cached results are bound. Extension admission establishes interface compliance, not optimization quality.

4.3 Static Gate at Submission

Let $V(u)$ indicate compilation validity, $R_{\mathrm{scale}}(u)$ the reported scale-risk set, and $\mathcal{H}_{\mathrm{cal}}$ the calibrated range of declared learning rates. Admission is defined as

$$A(u) = V(u)\,\mathbf{1}[R_{\mathrm{scale}}(u) = \varnothing]\;\mathbf{1}[\eta_{\mathrm{decl}}(u) \in \mathcal{H}_{\mathrm{cal}}]\,. \tag{7}$$

The default learning rate and calibrated range are specified in Appendix B.5. The learning-rate check applies before trainer-side batch scaling and is separate from dimensional analysis. Absolute-step schedules are reported but do not automatically trigger rejection.

Rejected candidates receive specific diagnostics and can be revised. The gate therefore guides exploration while avoiding training expenditure on identified risk patterns. Rules are versioned and linked to supporting failures and interventions. Historical reanalysis and prospective gated search are reported separately. Rejected successes are retained as evidence for revising overly conservative rules.

4.4 Multi-Fidelity Evaluation and Knowledge Updates

Admitted candidates enter three proxy stages with increasing model size. Let $\ell_i(u)$ denote validation bits per byte (bpb) at stage i, and let $I_i(u)$ indicate completion of that stage. Search uses

$$S(u) = \sum_{i=1}^{3} w_i I_i(u)\big(2.5 - \ell_i(u)\big), \qquad (w_1, w_2, w_3) = (0.2, 0.3, 0.5), \tag{8}$$

with zero contribution from uncompleted stages. Advancement requires meeting the preceding stage's criterion. The score reflects both performance and progression through the cascade, assigning greater weight to the largest proxy. Stage-level losses, completion status, and failure reasons are retained separately.

Caching binds results to the candidate program, cascade configuration, and compiler version. Evaluation outcomes update the shared leaderboard, candidate lineages, and research records, while experts write lessons that guide subsequent proposals.

After proxy search, selected candidates undergo full-fidelity training against matched-seed Muon runs. These results assess transfer and inform later system generations, but do not enter the same run's online proxy ranking.

Table 1: **Target-scale benchmark results.** Four representative discovered optimizers are compared with three baselines using training seed 0. Task scores, Avg(9), and CORE are on a 0–100 scale. Avg(9) averages the nine displayed tasks; CORE aggregates the full CORE task set. Higher is better except for validation bpb (val). Bold denotes the best listed value, including ties at the reported precision. Multi-seed results are reported in Appendix A.1.

Optimizer	Wino	BoolQ	CSQA	BB-Ops	ARC-C	CoQA	HS-0s	HS	ARC-E	Avg(9)	CORE	val ↓
Baselines												
AdamW	63.0	59.8	24.9	12.9	35.8	20.7	46.4	46.3	64.6	41.59	22.187	0.77083
Muon	69.6	**62.3**	22.4	17.6	38.0	29.7	53.6	53.8	67.8	46.09	27.349	0.73776
NorMuon	72.5	53.4	31.0	**21.4**	37.0	29.1	53.1	53.4	68.3	46.59	26.145	0.74064
Discovered												
AnnealSink	**72.9**	47.5	32.0	17.6	39.1	29.2	**53.9**	54.5	68.1	46.09	26.186	**0.73502**
Kronecker×Muon	71.8	56.1	25.6	17.1	**39.8**	**30.6**	**53.9**	54.5	67.6	46.34	27.538	0.73556
SignGate-Sink	70.7	61.7	28.7	18.1	37.8	30.1	53.5	54.1	67.0	46.85	27.576	0.73507
NormGrad-Sink	68.1	57.4	**34.0**	21.0	38.5	28.9	53.6	**54.8**	**68.6**	**47.21**	**27.625**	0.73551

5 Experiments and Optimizers

Experimental setup. OPTScientist and OpenEvolve both use GLM-5.2. The DSL-based comparison shares the DSL, compiler, initial program, proxy cascade, and evaluation cache. Each configuration uses three independent search runs with 60 candidate training evaluations per run. Early training failures count toward this budget; compilation failures and gate rejections do not. Budgets are aligned by training evaluations rather than LLM calls or token consumption.

The proxy cascade trains GPT-2-style models with 49.8M, 96.0M, and 201.1M parameters on climbmix at approximately 20 training tokens per parameter, using one H100 GPU per candidate. Full-fidelity evaluation trains the 1.1B-parameter nanochat d21 model on FineWeb for 21,000 steps (approximately 22B tokens) using four H100 GPUs (Karpathy, 2025). Both settings use the same tokenizer and sequence length of 2,048. Discovered optimizers handle matrix updates, while auxiliary parameters retain the standard nanochat recipe.

Muon and the discovered designs share the trainer and its common scheduling and scaling rules. Beyond the default-setting benchmark, Muon and AnnealSink receive matched one-dimensional learning-rate tuning, followed by a five-seed tuned comparison. The main benchmark uses training seed 0 throughout; repeated-seed results report means and sample standard deviations. Detailed training configurations, promotion thresholds, and tuning protocols are provided in the appendix.

Search performance and ablations. OPTScientist outperforms both OpenEvolve baselines in best proxy score and the number of proxy winners (Table 2). Removing shared knowledge or replacing the specialists with a single agent reduces search performance, supporting the contributions of experience sharing and complementary research perspectives. The full system preserves the proxy quality of the ungated specialist configuration while ensuring that all evaluated candidates satisfy the v2 constraints. The pass fraction measures constraint compliance; target-scale evaluation separately assesses whether the discovered optimizers retain their benefits.

Table 2: **Search baselines and ablations.** Each configuration uses three runs of 60 training evaluations. Best is the mean best proxy score ± sample SD; Wins is the mean number of proxy wins over Muon. Pass is the v2 gate-pass fraction among evaluated candidates.

Configuration	Best ↑	Wins ↑	Pass ↑
OE-over-DSL	$1.6207 \pm .0011$	7.33	0.382
OE-over-code	$1.6215 \pm .0007$	8.67	0.445
8 experts + knowledge	$\mathbf{1.6274} \pm .0014$	46.33	0.519
w/o knowledge	$1.6255 \pm .0023$	30.67	0.473
Single agent + knowledge	$1.6262 \pm .0020$	41.00	0.494
OPTScientist (full)	$1.6273 \pm .0011$	**48.00**	**1.000**

Target-scale optimizer performance. Table 1 presents four representative examples from the broader collection of discovered optimizers. All four achieve lower validation bpb than the listed baselines, and several also improve both downstream aggregates over Muon. The gains span mul-

tiple evaluation tasks, indicating that useful discoveries are not confined to a single update rule or benchmark metric. Validation and downstream rankings nevertheless differ, highlighting the importance of evaluating both rather than selecting solely by pretraining loss. These results demonstrate that the system discovers multiple effective update rules with complementary strengths across tasks. The complete collection and its mechanisms are documented in Appendix A.1.

Repeated-seed validation and resource costs. The observed validation gains persist across repeated training: AnnealSink outperforms matched-seed Muon on all five evaluated seeds, while three additional designs do so on all three evaluated seeds each. These paired comparisons support the consistency of the improvements beyond the single-seed benchmark. AnnealSink also retains its advantage after matched learning-rate tuning, reducing dependence on the baseline's default setting. The discovered collection includes designs that preserve Muon's persistent-state memory footprint and others that use additional preconditioning state. Thus, improved validation performance does not necessarily require larger optimizer states, while richer preconditioners offer alternative performance–resource trade-offs. Detailed results and resource measurements are reported in the appendix.

Transfer beyond language-model pretraining. We directly reuse the discovered update rules on CIFAR-100 with ViT-S/4, selecting learning rates without redesigning the optimizers. Several gate-admitted designs outperform Muon in mean accuracy across three training seeds (Table 3). Their performance supports cross-domain transfer of the discovered rules with learning-rate retuning, although admission does not guarantee improvement for every design.

Table 3: **Transfer to CIFAR-100.** ViT-S/4 top-1 accuracy (%), mean ± sample SD over three seeds. Discovered update rules are reused without redesign, with optimizer-specific learning-rate selection.

Optimizer	LR	Top-1 ↑
Muon	0.0025	62.89 ± 0.15
AnnealSink	0.0050	63.57 ± 0.28
Kronecker×Muon	0.0025	63.12 ± 0.23
SignGate-Sink	0.0025	62.69 ± 0.35
NormGrad-Sink	0.0025	$\mathbf{63.67} \pm 0.25$

Additional language-model experiments retain optimizer hyperparameters: AnnealSink performs comparably to the reference on a LLaMA-style model and slightly worse on a faithful GPT-2 configuration. These results further characterize the scope of transfer beyond the original search setting.

6 Conclusion

We introduce OPTScientist, a theory-guided multi-agent system that identifies proxy hacking, converts diagnosed failures into executable constraints, and combines specialist collaboration, shared knowledge, and an extensible typed DSL for optimizer discovery. The system discovers eighteen designs with lower target-scale validation bpb than all baselines in the main benchmark under the matched seed-0 protocol; selected designs also improve downstream performance and transfer to vision. These results show how experimental evidence can improve both the discovered training algorithms and the research process that produces them.

Limitations. The gate is a conservative policy calibrated to the current protocol: it can reject useful mechanisms and admit ineffective designs. Its scaling abstractions do not fully characterize state dynamics, numerical stabilizers, schedules, or distribution shifts. Retrospective screening also includes failures that informed the rules, motivating broader prospective validation. Limited computational resources prevent multi-seed evaluation of every discovered optimizer; repeated validation is therefore concentrated on selected designs.

Toward recursive self-improvement. Our work connects to recursive self-improvement in two ways. First, the research system improves its own discovery process by accumulating evidence and refining the knowledge, search space, and constraints used in subsequent iterations. Second, agents improve the optimization infrastructure used to train language models, contributing to the training process from which future agents can emerge. These complementary forms of improvement make optimizer discovery a concrete route toward recursive self-improving AI research.

REFERENCES

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. *Advances in neural information processing systems*, 29, 2016.

Jinheon Baek, Sujay Kumar Jauhar, Silviu Cucerzan, and Sung Ju Hwang. Researchagent: Iterative research idea generation over scientific literature with large language models. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 6709–6738, 2025.

Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V Le. Neural optimizer search with reinforcement learning. In *International Conference on Machine Learning*, pp. 459–468. PMLR, 2017.

Daniil A Boiko, Robert MacKnight, Ben Kline, and Gabe Gomes. Autonomous chemical research with large language models. *Nature*, 624(7992):570–578, 2023.

Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, et al. Symbolic discovery of optimization algorithms. *Advances in neural information processing systems*, 36:49205–49233, 2023.

Alireza Ghafarollahi and Markus J Buehler. Sciagents: automating scientific discovery through bioinspired multi-agent intelligent graph reasoning. *Advanced Materials*, 37(22):2413523, 2025.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *The twelfth international conference on learning representations*, 2023.

Andrej Karpathy. nanochat: The best chatgpt that $100 can buy, 2025. URL `https://github.com/karpathy/nanochat`.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112, 1994.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in neural information processing systems*, 36:51991–52008, 2023.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. *arXiv preprint arXiv:2401.02051*, 2024.

Hong Liu, Zhiyuan Li, David Hall, Percy Liang, and Tengyu Ma. Sophia: A scalable stochastic second-order optimizer for language model pre-training. *arXiv preprint arXiv:2305.14342*, 2023.

Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, et al. Muon is scalable for llm training. *arXiv preprint arXiv:2502.16982*, 2025a.

Liming Liu, Zhenghao Xu, Zixuan Zhang, Hao Kang, Zichong Li, Chen Liang, Weizhu Chen, and Tuo Zhao. Cosmos: A hybrid adaptive optimizer for memory-efficient training of llms. *arXiv preprint arXiv:2502.17410*, 2025b.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. arxiv. *arXiv preprint arXiv:1711.05101*, 10, 2019.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The ai scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024.

Mitchell Marfinetz. Evolving deep learning optimizers. *arXiv preprint arXiv:2512.11853*, 2025.

Abhinav Moudgil, Boris Knyazev, Guillaume Lajoie, and Eugene Belilovsky. Celo: Training versatile learned optimizers on a compute diet. *arXiv preprint arXiv:2501.12670*, 2025.

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

Matteo Pagliardini, Pierre Ablin, and David Grangier. The ademamix optimizer: Better, faster, older. *arXiv preprint arXiv:2409.03137*, 2024.

Esteban Real, Chen Liang, David So, and Quoc Le. Automl-zero: Evolving machine learning algorithms from scratch. In *International conference on machine learning*, pp. 8007–8019. Pmlr, 2020.

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

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.

Samuel Schmidgall, Yusheng Su, Ze Wang, Ximeng Sun, Jialian Wu, Xiaodong Yu, Jiang Liu, Michael Moor, Zicheng Liu, and Emad Barsoum. Agent laboratory: Using llm agents as research assistants. *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 5977–6043, 2025.

Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *International conference on machine learning*, pp. 4596–4604. PMLR, 2018.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Nikhil Vyas, Depen Morwani, Rosie Zhao, Mujin Kwun, Itai Shapira, David Brandfonbrener, Lucas Janson, and Sham Kakade. Soap: Improving and stabilizing shampoo using adam. *arXiv preprint arXiv:2409.11321*, 2024.

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

Kaiyue Wen, David Hall, Tengyu Ma, and Percy Liang. Fantastic pretraining optimizers and where to find them. *arXiv preprint arXiv:2509.02046*, 2025.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First conference on language modeling*, 2024.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2023.

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

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. Reevo: Large language models as hyper-heuristics with reflective evolution. *Advances in neural information processing systems*, 37:43571–43608, 2024.

Huizhuo Yuan, Yifeng Liu, Shuang Wu, Xun Zhou, and Quanquan Gu. Mars: Unleashing the power of variance reduction for training large models. *arXiv preprint arXiv:2411.10438*, 2024.

Yushun Zhang, Congliang Chen, Ziniu Li, Tian Ding, Chenwei Wu, Diederik P Kingma, Yinyu Ye, Zhi-Quan Luo, and Ruoyu Sun. Adam-mini: Use fewer learning rates to gain more. *arXiv preprint arXiv:2406.16793*, 2024.

Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. Galore: Memory-efficient llm training by gradient low-rank projection. *arXiv preprint arXiv:2403.03507*, 2024.

A Appendix

A.1 Discovered Optimizers: Results and Update Rules

We document eighteen optimizer designs that outperform matched-seed Muon in validation bpb at the target scale of 1.1B parameters and 21,000 training steps. The collection includes designs from different search generations, system ablations, and a factorization experiment. It is a catalog of successful designs, rather than the evaluation cohort for estimating retrospective gate performance.

We separate single-seed comparisons from repeated-seed validation. Table 4 uses training seed 0 throughout, while Table 5 reports additional evaluations across seeds. Search-run identifiers below indicate provenance and should not be confused with training seeds.

Names reflect the principal mechanisms: *Anneal* denotes momentum scheduling, *Sink* denotes alternating row/column normalization, *Kron* denotes matrix-statistic preconditioning, *Gate* denotes sign-agreement modulation, and *Pre* indicates normalization before the polar operator. Earlier designs are expressed retrospectively in a common notation; this does not imply that the earlier search generations used DSL-based synthesis.

Single-seed comparison. All results in Table 4 use training seed 0. The corresponding Muon validation bpb and CORE are 0.73776 and 27.349, respectively. We report

$$\Delta_0 = 10^3 \left(\mathrm{bpb}_{\mathrm{Muon},0} - \mathrm{bpb}_{\mathrm{candidate},0} \right), \tag{9}$$

so that positive values favor the candidate. CORE is the centered aggregate over 22 tasks, displayed on a 0–100 scale. The proxy score is the combined score of the three-stage cascade at 20 training tokens per parameter.

All eighteen designs improve validation bpb in this single-seed comparison, but improvements in CORE are not uniform. Validation gains should therefore not be interpreted as improvements on every downstream metric.

Seventeen designs pass checker v2. AnnealSink-S is rejected because its declared learning rate of 0.025 exceeds the calibrated upper bound of 0.02, despite improving target-scale validation. SinkMuon was evaluated as a factorization control despite failing the stage-1 proxy promotion criterion. These exceptions illustrate the conservatism of the static gate and the limitations of early-stage screening.

Repeated-seed validation. We additionally evaluate four designs across multiple training seeds. For seed set $\mathcal{S}$, define

$$\overline{\Delta} = \frac{10^3}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \left(\mathrm{bpb}_{\mathrm{Muon},s} - \mathrm{bpb}_{\mathrm{candidate},s} \right). \tag{10}$$

The Muon validation bpb values for seeds 0–4 are 0.73776, 0.74106, 0.73613, 0.73625, and 0.73813. Each candidate is compared against Muon on the same seed set.

Table 5 reports means and sample standard deviations, using the $n - 1$ denominator. AnnealSink outperforms matched-seed Muon on all five seeds; the other three designs do so on all three seeds. These repeated evaluations support the consistency of the observed validation improvements.

Table 4: **Single-seed target-scale comparison.** All validation and CORE results use training seed 0. Δ_0 is the validation improvement over same-seed Muon in units of 10^{-3}, calculated from the displayed bpb values. CORE is reported on a 0–100 scale. Memory is the static FP32 optimizer-state estimate in MB, rounded to the nearest integer using the same accounting function for all designs. It excludes activations and temporary workspaces. Gate decisions refer to checker v2. A dash indicates an unavailable or inapplicable entry.

ID	Optimizer	Proxy	Val bpb ↓	Δ_0 ↑	CORE ↑	Memory	Gate
–	Muon	1.6203	0.73776	0.00	27.349	1906	–
E1	AnnealSink	1.6235	0.73502	2.74	26.186	1906	Pass
E2	AnnealResidual	1.6233	0.73526	2.50	26.889	1906	Pass
E3	AnnealMuon	1.6222	0.73633	1.43	26.707	1906	Pass
E4	SinkMuon	–	0.73679	0.97	26.900	1906	Pass
E5	Kronecker×Muon	1.6284	0.73556	2.20	27.538	5241	Pass
E6	SignGate-Sink	1.6276	0.73507	2.69	27.576	1906	Pass
E7	NormGrad-Sink	1.6276	0.73551	2.25	27.625	1906	Pass
E8	AgreeGate-Sink	1.6276	0.73512	2.64	26.700	1906	Pass
E9	FactorWhite-Sink	1.6276	0.73505	2.71	27.420	1908	Pass
E10	KronGate-Sink-A	1.6288	0.73532	2.44	28.380	8576	Pass
E11	KronGate-Sink-B ($\gamma = 0.25$)	1.6292	0.73517	2.59	27.610	8576	Pass
E12	KronGate-Sink-B ($\gamma = 0.30$)	1.6293	0.73525	2.51	27.750	8576	Pass
E13	KronGate-Sink-LA	1.6289	0.73529	2.47	27.960	8576	Pass
E14	CosLookahead-Sink	1.6239	0.73562	2.14	27.190	1906	Pass
E15	DebiasAnneal-RowNorm	1.6278	0.73643	1.33	26.930	1906	Pass
E16	PreSink-Muon	1.6245	0.73505	2.71	28.000	1906	Pass
E17	AnnealSink-S	1.6248	0.73676	1.00	27.700	1906	Reject
E18	SoftPreSink-Muon	1.6236	0.73547	2.29	26.450	1906	Pass

Table 5: **Repeated-seed target-scale validation.** Values are mean ± sample standard deviation. CORE is reported on a 0–100 scale. $\overline{\Delta}$ is the mean matched-seed validation improvement in units of 10^{-3}. Wins counts seeds with lower validation bpb than Muon.

Optimizer	Seeds	Val bpb ↓	CORE ↑	$\overline{\Delta}$ ↑	Wins
AnnealSink	0–4	0.73526 ± 0.00034	26.935 ± 0.679	2.60	5/5
AnnealResidual	0–2	0.73549 ± 0.00071	26.561 ± 0.663	2.83	3/3
AnnealMuon	0–2	0.73598 ± 0.00032	26.567 ± 0.161	2.34	3/3
Kronecker×Muon	0–2	0.73514 ± 0.00061	27.093 ± 0.793	3.17	3/3

Shared notation and primitives. Let G_t denote a matrix gradient and M_t its momentum state. Unless otherwise specified,

$$M_t = \beta_t M_{t-1} + (1 - \beta_t) G_t. \tag{11}$$

We use $\mathcal{N}_\beta(M, G)$, $\mathcal{P}(X)$, $\mathcal{A}(M, G)$, and $\mathcal{Q}(X)$ for the implementation's `nesterov`, `polar`, `sign_agree`, and `row_norm` primitives, respectively. This notation preserves the implementation semantics without imposing alternative conventions.

One round of alternating row/column normalization is

$$\begin{aligned}\mathcal{R}(X) &= X \odot (\mathrm{rowmean}(X \odot X) + \epsilon)^{-1/2}, \\ \mathcal{C}(X) &= X \odot (\mathrm{colmean}(X \odot X) + \epsilon)^{-1/2}, \\ \mathcal{S}(X) &= \mathcal{C}\big(\mathcal{R}(X)\big),\end{aligned} \tag{12}$$

where $\odot$ denotes elementwise multiplication and statistics are broadcast over the corresponding dimensions. $\mathcal{S}^k$ denotes k rounds. These finite normalization operations do not assume convergence to a doubly stochastic matrix.

We define the norm-restoring operation

$$\mathcal{F}(X; D) = X \frac{\|D\|_F}{\|X\|_F + \epsilon}. \tag{13}$$

It restores the reference Frobenius norm up to the numerical stabilizer.

The following expressions specify the candidate update tensor U_t before the shared training wrapper applies learning-rate scheduling and common parameter-update handling. All designs declare a learning rate of 0.02 except E17, which declares 0.025. E1–E13 and E17 explicitly disable automatic bias correction; the remaining designs retain their recorded implementation defaults.

E1–E4: Momentum annealing and post-polar balancing. AnnealSink, AnnealResidual, and AnnealMuon originate from Generation 1, with search identifiers `seed1/069_momentum`, `seed1/070_norm`, and `seed1/034_momentum`. SinkMuon is the no-annealing factorization control.

AnnealSink (E1) uses

$$\begin{aligned}\beta_t &= 0.95\,\sigma((t-50)/25), \\ D_t &= \mathcal{P}\big(\mathcal{N}_{0.95}(M_t, G_t)\big), \\ U_t &= \mathcal{F}\big(\mathcal{S}(D_t); D_t\big).\end{aligned} \tag{14}$$

The momentum coefficient is annealed, but the coefficient supplied to the Nesterov primitive remains fixed at 0.95.

AnnealResidual (E2) uses $\beta_t = 0.95\,\sigma((t-25)/20)$ and the same fixed-coefficient Nesterov construction. Its update is

$$\begin{aligned}Y_t &= \mathcal{F}\big(\mathcal{S}(D_t); D_t\big), \\ U_t &= \mathcal{F}(D_t + Y_t; D_t).\end{aligned} \tag{15}$$

AnnealMuon (E3) uses the E2 annealing schedule but removes alternating normalization:

$$U_t = \mathcal{P}\big(\mathcal{N}_{\beta_t}(M_t, G_t)\big). \tag{16}$$

SinkMuon (E4) retains the E1 update construction with $\beta_t = 0.95$ throughout training.

E5, E6, and E8: Preconditioning and sign gating. Define the two-stage momentum schedule

$$b(t; a, h) = 0.80 + 0.17\ \mathrm{clip}(t/a, 0, 1) + 0.02\ \mathrm{clip}((t-a)/h, 0, 1). \tag{17}$$

For compactness, define gated post-processing as

$$\begin{aligned}Z &= D \odot (1 + \gamma \mathcal{A}(M, G)), \\ \mathcal{H}_\gamma(D; M, G) &= \mathcal{F}\big(\mathcal{S}^2(Z); Z\big).\end{aligned} \tag{18}$$

The reference norm is that of the gated tensor Z.

SignGate-Sink (E6; `seed1/089_meta`) uses $\beta_t = b(t; 300, 1400)$ and

$$\begin{aligned}D_t &= \mathcal{P}\big(\mathcal{N}_{\beta_t}(M_t, G_t)\big), \\ U_t &= \mathcal{H}_{0.25}(D_t; M_t, G_t).\end{aligned} \tag{19}$$

AgreeGate-Sink (E8; `seed1/130_schedule`) uses the same construction with $\beta_t = b(t; 400, 1400)$.

Kronecker×Muon (E5; `seed1/149_geometry`) uses $\beta_t = b(t; 300, 1400)$ and adds left preconditioning:

$$\begin{aligned} L_t &= 0.95L_{t-1} + 0.05G_tG_t^\top, \\ A_t &= \text{block_inv_sqrt}(L_t), \\ \widetilde{M}_t &= A_tM_t, \qquad \widetilde{G}_t = A_tG_t, \\ D_t &= \mathcal{P}\big(\mathcal{N}_{\beta_t}(\widetilde{M}_t, \widetilde{G}_t)\big), \\ U_t &= \mathcal{H}_{0.25}(D_t; M_t, G_t). \end{aligned} \tag{20}$$

The agreement gate uses the original momentum and gradient. This implementation maintains only a left matrix statistic; E10–E13 use two-sided statistics.

E7: Normalized gradients and pre-polar balancing. NormGrad-Sink (E7; `seed3/076_momentum`) normalizes each gradient before momentum accumulation:

$$\begin{aligned} \beta_t &= 0.85 + 0.12\min(t/400, 1), \\ \widehat{G}_t &= \frac{G_t}{\text{rms}(G_t) + 10^{-8}}, \\ M_t &= \beta_tM_{t-1} + (1-\beta_t)\widehat{G}_t, \\ X_t &= \mathcal{N}_{\beta_t}(M_t, \widehat{G}_t), \\ U_t &= \mathcal{Q}\left(\mathcal{P}\left(\mathcal{S}^2(X_t)\right)\right). \end{aligned} \tag{21}$$

Alternating normalization precedes the polar operation. The final row normalization can change the spectral properties of the polar output.

E9: Factorized second-moment preconditioning. FactorWhite-Sink (E9; `seed1/110_geometry`) uses $\beta_t = b(t; 300, 1400)$ and row/column states:

$$\begin{aligned} q_t &= 0.95q_{t-1} + 0.05\,\text{rowmean}(G_t \odot G_t), \\ s_t &= 0.95s_{t-1} + 0.05\,\text{colmean}(G_t \odot G_t), \\ X_t &= (q_t + \epsilon)^{-1/4} \odot \mathcal{N}_{\beta_t}(M_t, G_t) \odot (s_t + \epsilon)^{-1/4}, \\ D_t &= \mathcal{P}(X_t), \\ U_t &= \mathcal{H}_{0.25}(D_t; M_t, G_t). \end{aligned} \tag{22}$$

The factorized state incurs substantially less optimizer-state memory than the two-sided matrix variants in this setup.

E10–E13: Two-sided matrix-statistic variants. These designs use $\beta_t = b(t; 300, 1400)$ and

$$\begin{aligned} L_t &= 0.95L_{t-1} + 0.05G_tG_t^\top, \\ R_t &= 0.95R_{t-1} + 0.05G_t^\top G_t, \\ A_t &= \text{block_inv_sqrt}(L_t), \\ B_t &= \text{block_inv_sqrt}(R_t), \\ \widetilde{M}_t &= A_tM_tB_t, \qquad \widetilde{G}_t = A_tG_tB_t, \\ D_t &= \mathcal{P}\big(\mathcal{N}_{\beta_t}(\widetilde{M}_t, \widetilde{G}_t)\big). \end{aligned} \tag{23}$$

KronGate-Sink-A (E10; `seed1/128_precond`) computes agreement before preconditioning:

$$U_t = \mathcal{H}_{0.25}(D_t; M_t, G_t). \tag{24}$$

KronGate-Sink-B computes agreement after preconditioning:

$$U_t = \mathcal{H}_{\gamma}(D_t; \widetilde{M}_t, \widetilde{G}_t). \tag{25}$$

E11 (`seed1/147_geometry`) uses $\gamma = 0.25$, and E12 (`seed1/154_gating`) uses $\gamma = 0.30$.

KronGate-Sink-LA (E13; `seed1/154_geometry`) uses the E11 construction, but computes the preconditioners from

$$L_t^+ = L_t + 0.05 G_t G_t^\top, \qquad R_t^+ = R_t + 0.05 G_t^\top G_t. \tag{26}$$

The additional current-gradient contribution affects preconditioning without replacing the stored EMA states.

E14: Cosine-adaptive lookahead. CosLookahead-Sink (E14; `seed0/027_momentum`) is the selected candidate from a Generation-3 search arm using gate v1:

$$\begin{aligned} \beta_t &= 0.85 + 0.10\ \mathrm{clip}(t/300, 0, 1), \\ c_t &= \mathrm{cossim}(M_t, G_t), \\ \alpha_t &= \beta_t + 0.05 c_t, \\ X_t &= \alpha_t M_t + (1-\alpha_t) G_t, \\ U_t &= \mathcal{P}\big(\mathcal{S}^2(X_t)\big). \end{aligned} \tag{27}$$

The program applies α_t directly, without an additional clipping operation.

E15: Annealed momentum with explicit debiasing. DebiasAnneal-RowNorm (E15; `seed0/043_norm`) originates from the no-knowledge ablation:

$$\begin{aligned} \beta_t &= 0.80 + 0.17\,\sigma((t-200)/100), \\ \widehat{M}_t &= \mathrm{ema_debias}(M_t, \beta_t, t), \\ U_t &= \mathcal{Q}\left(\mathcal{P}\big(\mathcal{N}_{\beta_t}(\widehat{M}_t, G_t)\big)\right). \end{aligned} \tag{28}$$

The `ema_debias` primitive retains its implementation semantics; it is not replaced here by a constant-coefficient correction formula.

E16: Pre-polar normalization using RMS primitives. PreSink-Muon (E16; `seed1/048_geometry`) also originates from the no-knowledge ablation. It uses $\beta_t = 0.95\ \mathrm{clip}(t/200, 0, 1)$ and $X_t = \mathcal{N}_{\beta_t}(M_t, G_t)$. Define

$$\begin{aligned} \widetilde{\mathcal{R}}(X) &= X \odot \mathrm{rsqrt}\left(\mathrm{row_rms}(X) + \epsilon\right), \\ \widetilde{\mathcal{C}}(X) &= X \odot \mathrm{rsqrt}\left(\mathrm{col_rms}(X) + \epsilon\right). \end{aligned} \tag{29}$$

Its update is

$$U_t = \mathcal{P}\left((\widetilde{\mathcal{C}} \circ \widetilde{\mathcal{R}})^2(X_t)\right). \tag{30}$$

These RMS primitive calls are preserved explicitly: an inverse square root of an RMS statistic differs from an inverse square root of a mean-square statistic.

E17: AnnealSink outside the calibrated LR band. AnnealSink-S (E17; `single-agent/seed0/062`) was discovered by the single-agent ablation. It declares a learning rate of 0.025 and uses

$$\begin{aligned} \beta_t &= 0.85 + 0.10\ \mathrm{clip}(t/400, 0, 1), \\ D_t &= \mathcal{P}\big(\mathcal{N}_{0.95}(M_t, G_t)\big), \\ U_t &= \mathcal{F}\big(\mathcal{S}^2(D_t); D_t\big). \end{aligned} \tag{31}$$

Its positive target-scale result is retained despite its rejection by the v2 learning-rate constraint.

E18: Soft pre-polar balancing. SoftPreSink-Muon (E18; `roster-4/seed0/056_norm`) originates from the four-expert ablation. It uses $\beta_t = 0.85 + 0.10\min(t/300, 1)$ and $X_t = \mathcal{N}_{\beta_t}(M_t, G_t)$. Define

$$\begin{aligned} \mathcal{R}_{\mathrm{soft}}(X) &= X \odot \left(\mathrm{rowmean}(X \odot X) + \epsilon\right)^{-1/4}, \\ \mathcal{C}_{\mathrm{soft}}(X) &= X \odot \left(\mathrm{colmean}(X \odot X) + \epsilon\right)^{-1/4}. \end{aligned} \tag{32}$$

The update is

$$U_t = \mathcal{P}\left((\mathcal{C}_{\mathrm{soft}} \circ \mathcal{R}_{\mathrm{soft}})^2(X_t)\right). \tag{33}$$

The fourth-root factors provide weaker normalization than the inverse-square-root mean-square factors in Eq. 12.

Table 6: **CIFAR-100 ViT results for selected designs.** Top-1 accuracy is reported as mean ± sample standard deviation in percent over three training seeds at the selected learning rate.

Optimizer	Learning rate	Top-1 accuracy (%)
AnnealSink	0.0050	63.57 ± 0.28
Kronecker×Muon	0.0025	63.12 ± 0.23
SignGate-Sink	0.0025	62.69 ± 0.35
NormGrad-Sink	0.0025	63.67 ± 0.25
FactorWhite-Sink	0.0025	62.90 ± 0.29
PreSink-Muon	0.0025	63.29 ± 0.34

Transfer to vision. Table 6 reports CIFAR-100 ViT results for six selected designs. Each optimizer uses its individually selected learning rate, with results aggregated over three training seeds. The uncertainty is the sample standard deviation. These experiments assess transfer with learning-rate retuning, rather than zero-shot hyperparameter transfer.

B Experimental Configuration and Evaluation Protocols

B.1 Research Agents, Baselines, and Search Budgets

Both OPTScientist and OpenEvolve use GLM-5.2 as the proposal model. Within an OPTScientist run, eight specialists work asynchronously, combining domain-specific priors with shared research knowledge. Each specialist independently proposes and implements candidates, obtains experimental feedback, and records lessons for subsequent research. Independent search runs start from the same initial conditions.

The OpenEvolve-over-DSL comparison shares the DSL, compiler, initial optimizer program, three-stage evaluation cascade, evaluation cache, and proxy-score definition with OPTScientist. OpenEvolve-over-code is reported separately as a code-space baseline. Each configuration is evaluated over three independent runs, with a budget of 60 candidate training evaluations per run. OPTScientist specialists can make multiple proposals within a session; OpenEvolve makes one language-model call per iteration. The comparison aligns candidate-evaluation budgets, not language-model calls, generated tokens, or total research cost.

A candidate consumes one evaluation slot once it enters training. Stage-1 failures and divergent training runs count toward the budget. Compilation failures are returned before submission, and gate-rejected submissions do not enter training; neither consumes a training-evaluation slot. Counts follow scored evaluation records rather than unique textual experiment identifiers, including when identifiers collide.

For each run, Best@60 is the highest combined proxy score among the 60 evaluated records. The main table reports its mean and sample standard deviation across runs. Wins@60 counts evaluated records whose proxy scores strictly exceed the Muon reference score, and is averaged across runs. The v2 pass fraction uses training-evaluated records as its denominator, not all proposals. Consequently, a pass fraction of one in the gated configuration indicates enforcement of admission rules; it does not measure how often raw proposals are admissible or how often candidates improve at the target scale.

B.2 Proxy and Full-Fidelity Training

Table 7 summarizes the training configurations. The proxy cascade uses GPT-2-style models trained on climbmix. The target evaluation uses nanochat d21 trained on FineWeb shards prepared for the approximately 20-tokens-per-parameter recipe. Both settings use the nanochat tokenizer with vocabulary size 32,768 and sequence length 2,048.

Training-token counts are computed as optimizer steps times effective batch size. Each proxy stage uses approximately 20 training tokens per parameter. Completing all three proxy stages requires approximately 4.3 H100 GPU-hours per candidate. A full-fidelity run takes approximately 14 hours

Table 7: **Training configurations.** Batch sizes are effective tokens per optimizer step. A candidate progresses sequentially through the proxy stages.

Setting	Proxy 1	Proxy 2	Proxy 3	Full fidelity
Parameters	49.8M	96.0M	201.1M	≈1.1B
Depth	10	11	16	21
Width	640	768	1024	—
Dataset	climbmix	climbmix	climbmix	FineWeb
Training steps	1,900	3,663	7,672	21,000
Sequence length	2,048	2,048	2,048	2,048
Effective batch	2^{19}	2^{19}	2^{19}	2^{20}
Device batch (sequences)	128	64	16	16
Gradient accumulation	2	4	16	8
H100 GPUs per candidate	1	1	1	4
Training tokens	0.996B	1.920B	4.022B	22.020B

on four H100s, or 56 GPU-hours. These are approximate training costs, not total search costs including language-model inference and all auxiliary evaluations.

The proxy-to-target comparison changes model size, training horizon, effective batch size, and training corpus. It therefore evaluates transfer across the stated protocols, rather than isolating model width as the only intervention. The mechanistic interventions instead keep the target trainer configuration fixed while modifying the specified optimizer component.

B.3 Cascade Promotion, Scoring, and Confirmation

A stage-1 validation bpb of at most 1.12 permits advancement to stage 2; a stage-2 validation bpb of at most 1.05 permits advancement to stage 3. Stage 3 has no further promotion threshold. Approximately 93% of candidates in the recorded proxy evaluations complete all three stages.

For a candidate u, let $\ell_i(u)$ denote validation bpb and $I_i(u)$ indicate completion of stage i. The score is

$$S(u) = \sum_{i=1}^{3} w_i I_i(u)\big(2.5 - \ell_i(u)\big), \qquad (w_1, w_2, w_3) = (0.2, 0.3, 0.5). \tag{34}$$

Uncompleted stages contribute zero. Training failures remain part of the evaluation budget irrespective of their assigned score. Stage losses, completion status, and failure reasons are retained alongside the combined score.

Evaluation caching binds outcomes to the candidate program, cascade configuration, and compiler version. Selected candidates are subsequently evaluated at full fidelity against matched-seed Muon runs. Target-scale outcomes do not enter the online ranking of the same proxy-search run; they support subsequent analysis and later research iterations.

B.4 Optimizer Interface and Reference Recipe

Candidate programs specify matrix updates. Embeddings and other designated auxiliary parameters retain the trainer's AdamW handling. Proxy matrix weight decay is zero. At full fidelity, all designs are evaluated through the same trainer and its common parameter-group, scheduling, and scaling machinery. Candidate-added parameter terms are distinct from weight decay applied by the trainer.

The Muon reference retains the nanochat recipe: Nesterov updates, a trainer-supplied momentum coefficient increasing from 0.85 to 0.95 over the first 300 steps, and the default Newton–Schulz iteration setting. Embedding, unembedding, and scalar parameter groups use nominal AdamW learning

Table 8: **Nominal learning rates in the baseline configurations.** Matrix learning rates are subsequently transformed by the applicable trainer scaling and schedule.

Optimizer	Nominal learning rate
Muon	0.02
Muon-KJ	0.02
Muon-MuP	0.02
NorMuon	0.02
AdamW	3×10^{-4}
Lion	10^{-4}
RMSProp	10^{-3}
SGD	0.02
Sophia	3×10^{-4}

rates of 0.3, 0.004, and 0.5, respectively. The learning-rate schedule has no warmup and decays linearly over the final half of training. Trainer-side matrix weight decay is normalized to the token budget and decreases linearly to zero, starting at approximately 0.057 in the 21,000-step configuration. These settings are not independently retuned in the optimizer comparison.

For effective batch size B measured in tokens, the trainer applies the matrix learning-rate multiplier

$$\eta_{\text{base}} = \eta_{\text{decl}} \sqrt{B/2^{19}}. \tag{35}$$

The multiplier is one in the proxy setting and $\sqrt{2}$ at full fidelity. Thus, a declared learning rate of 0.02 corresponds to approximately 0.0283 before the target learning-rate schedule and any other update scaling are applied. Declared learning rates must therefore be distinguished from the trainer-scaled rate.

Most candidate programs declare a learning rate of 0.02. Exceptions in the diagnostic experiments include the additional-parameter-term candidate at 0.035, TwoPhase-A/B at 0.045, and AnnealSink-S at 0.025. Their original settings are retained when reporting the corresponding failures or screening exceptions.

Additional shape-scaling comparisons include Muon-KJ, with factor $\sqrt{\max(1, d_{\text{out}}/d_{\text{in}})}$, and Muon-MuP. The former is evaluated on three training seeds and the latter on one. These variants do not improve on the plain Muon reference in the reported comparison.

B.5 Learning-Rate Tuning and Static Admission

The main benchmark reports the primary declared-rate settings. A separate matched tuning experiment scans the one-dimensional matrix learning-rate grid

$$\eta_{\text{decl}} \in \{0.010, 0.014, 0.020, 0.028\} \tag{36}$$

for Muon and AnnealSink, with all other recipe settings fixed. On training seed 0, Muon's best grid point is 0.014, with validation bpb 0.73742, compared with 0.73776 at the default 0.020. The tuned comparison evaluates both methods at 0.014 on five matched training seeds. Default-setting and tuned results are reported separately.

Checker v2 combines structural validity, scale-risk screening, and a declared-learning-rate constraint:

$$\begin{aligned} A(u) &= V(u)\,\mathbf{1}[R_{\text{scale}}(u) = \varnothing]\,\mathbf{1}[\eta_{\text{decl}}(u) \in \mathcal{H}_{\text{cal}}]\,, \\ \mathcal{H}_{\text{cal}} &= (0, 0.02]. \end{aligned} \tag{37}$$

Here $V(u)$ denotes compilation validity and $R_{\text{scale}}(u)$ the reported scale-risk set. The learning-rate test operates on the declared rate, before the batch multiplier in Eq. 35; it is an empirically calibrated constraint, not a dimensional property. Absolute-step schedules trigger warnings rather than rejection. Diagnostic sweeps and retrospective evaluation of historical candidates are distinct from admission to a gated search run.

In the retrospective cohort of 28 fully evaluated designs, 17 of 18 admitted designs and 1 of 10 rejected designs improve over Muon. The rejected success, AnnealSink-S, declares a rate of 0.025.

Table 9: **Matched five-seed comparison.** Values are mean ± sample standard deviation. CORE is reported on a 0–100 scale. Δ is the mean paired difference (AnnealSink minus Muon), in the displayed metric's units. Wins counts seeds favoring AnnealSink. Tests are two-sided paired t-tests with four degrees of freedom; p-values are unadjusted.

Setting	Metric	Muon	AnnealSink	Δ	Wins	$t(4)$	p
Default, LR 0.020	Val bpb ↓	0.73787 ± 0.00199	0.73526 ± 0.00034	-0.00260	5/5	-3.44	0.026
Tuned, LR 0.014	Val bpb ↓	0.73735 ± 0.00239	0.73469 ± 0.00037	-0.00266	5/5	-2.91	0.044
Tuned, LR 0.014	CORE ↑	26.658 ± 0.368	27.169 ± 0.527	$+0.510$	4/5	1.81	0.145

Table 10: **Per-seed results after matched learning-rate tuning.** Both optimizers use declared matrix learning rate 0.014. CORE is reported on a 0–100 scale.

	Validation bpb ↓		CORE ↑	
Seed	Muon	AnnealSink	Muon	AnnealSink
0	0.73742	0.73455	26.592	27.678
1	0.74112	0.73535	27.150	27.323
2	0.73541	0.73452	26.304	27.591
3	0.73517	0.73444	26.333	26.452
4	0.73764	0.73458	26.913	26.799

FISMO-lite is an admitted design that does not improve. Because historical outcomes informed the rules, this analysis is a calibration diagnostic rather than an independent prospective estimate of success on unseen candidates.

B.6 Matched Learning-Rate Tuning and Repeated-Seed Results

Muon and AnnealSink receive the same one-dimensional search over declared matrix learning rates $\{0.010, 0.014, 0.020, 0.028\}$ on training seed 0. Both select 0.014, an interior point of the search grid. We then evaluate this setting on five matched training seeds using the 1.1B-parameter model and 21,000-step target protocol, with all other trainer settings fixed.

Table 9 separates the tuned comparison from the default-setting validation results. AnnealSink achieves lower validation bpb on every evaluated seed under both settings. Its mean CORE score also improves at the tuned setting, although the paired test does not establish a statistically significant downstream improvement.

Because seed 0 is used for learning-rate selection, the five-seed comparison includes the selection seed. AnnealSink also improves validation bpb on all four remaining seeds. The paired tests summarize the observed training variability for the selected configurations; they do not account for the broader optimizer-discovery and hyperparameter-selection process.

B.7 Mechanistic Intervention Protocol

Interventions use training seed 0 and hold the remaining target trainer configuration fixed. The tested edits remove an additional parameter-dependent update term, reduce an excessive declared learning rate, or replace AnnealSink's norm-correction numerator with the parameter norm. The learning-rate edits include 0.045 to 0.020 and, for the parameter-term candidate with that term retained, 0.035 to 0.020.

The dose-response experiment varies the coefficient c of an additional candidate term according to

$$U_c = U_{\text{base}} + c\,\omega W, \qquad c \in \{0, 0.5, 1, 2\}, \tag{38}$$

where ωW denotes the candidate's original additional parameter term. The declared learning rate remains 0.035, and trainer-side weight decay remains unchanged. Thus, $c = 0$ removes the candidate-added term without removing the trainer's regularization. The observed degradation under parameter-norm injection is reported as a finite validation result, not as numerical divergence.

Table 11: **Language-model architecture transfer without optimizer-specific retuning.** Validation bpb is reported as mean ± sample standard deviation over three matched seeds. Wins counts seeds with lower bpb for AnnealSink.

Architecture	Muon	AnnealSink	Wins
LLaMA-style, 599M	0.82389 ± 0.00033	0.82348 ± 0.00014	3/3
Faithful GPT-2, 1.04B	0.83105 ± 0.00018	0.83264 ± 0.00040	0/3

Table 12: **Per-seed architecture-transfer results.** Entries are validation bpb; lower is better.

	LLaMA-style		Faithful GPT-2	
Seed	Muon	AnnealSink	Muon	AnnealSink
0	0.82354	0.82348	0.83096	0.83299
1	0.82392	0.82334	0.83126	0.83220
2	0.82420	0.82361	0.83094	0.83273

B.8 Training Seeds and Statistical Reporting

The main language-model benchmark and the single-seed design catalog use training seed 0 throughout. Its Muon reference has validation bpb 0.73776 and CORE 27.349. Repeated-seed results are kept separate: AnnealSink uses seeds 0–4, while AnnealResidual, AnnealMuon, and Kronecker×Muon use seeds 0–2. Muon validation bpb for seeds 0–4 is 0.73776, 0.74106, 0.73613, 0.73625, and 0.73813, respectively.

For each candidate, the mean improvement uses the same subset of Muon training seeds. Reported standard deviations use the sample denominator $n - 1$. A strict win requires lower candidate validation bpb than its matched-seed reference. AnnealSink wins on all five evaluated seeds, and the other three designs each win on all three. These counts describe observed repeatability; they are not significance tests.

Downstream task scores, Avg(9), and CORE are displayed on a 0–100 scale, whereas validation bpb remains in its original units. Avg(9) is computed from the nine task scores before display rounding. CORE is the centered aggregate over the full 22-task suite and is not the same statistic as Avg(9).

B.9 Transfer Across Language-Model Architectures

We evaluate AnnealSink on two additional language-model architectures without optimizer-specific retuning. The LLaMA-style model has approximately 599M parameters, 21 layers, and hidden dimension 1,408. The faithful GPT-2 model has approximately 1.04B parameters, 28 layers, hidden dimension 1,664, and 13 attention heads, with learned positional embeddings, biased LayerNorm, and GELU activations.

Both configurations train on FineWeb-Edu for 3,000 steps with approximately one million tokens per step using four H100 GPUs. They share the tokenizer and sequence length of 2,048. The optimizer programs retain their declared hyperparameters, including learning rate 0.02. Trainer-side batch-dependent learning-rate scaling, momentum overrides, and weight-decay handling remain shared between Muon and AnnealSink. Each comparison uses three matched training seeds.

AnnealSink yields a small validation improvement on the LLaMA-style model across all three seeds, while performing worse on the faithful GPT-2 configuration. These results show that its advantage is architecture-dependent. Unlike the vision experiments, these comparisons reuse optimizer hyperparameters without retuning.

B.10 Vision Transfer

Vision experiments directly reuse the discovered optimizer programs on CIFAR-100 with ViT-S/4, trained for 200 epochs with batch size 512. Eligible matrix parameters use the DSL optimizer with zero matrix weight decay. Patch embeddings, the class token, positional embeddings, the prediction

Table 13: **Vision-transfer results.** CIFAR-100 top-1 accuracy (%), mean ± sample SD over three training seeds at the selected learning rate.

Optimizer	Learning rate	Top-1 accuracy (%)
Muon	0.0025	62.89 ± 0.15
AnnealSink	0.0050	63.57 ± 0.28
Kronecker×Muon	0.0025	63.12 ± 0.23
SignGate-Sink	0.0025	62.69 ± 0.35
NormGrad-Sink	0.0025	$\mathbf{63.67} \pm 0.25$
FactorWhite-Sink	0.0025	62.90 ± 0.29
PreSink-Muon	0.0025	63.29 ± 0.34

head, LayerNorm parameters, and biases are handled by AdamW with learning rate 10^{-3}. Thus, parameter assignment follows the explicit trainer groups rather than tensor rank alone.

Each optimizer is evaluated over three training seeds at its selected learning rate. Muon uses 0.0025; AnnealSink uses 0.005; Kronecker×Muon, SignGate-Sink, NormGrad-Sink, FactorWhite-Sink, and PreSink-Muon use 0.0025. Results are mean top-1 accuracy with sample standard deviation. The optimizer update rules are unchanged, but the learning rate is selected for the vision setting. These experiments therefore assess cross-domain transfer with learning-rate retuning, not zero-shot transfer of all hyperparameters.

B.11 Optimizer-State Memory Accounting

Optimizer-state memory is estimated statically using the same `estimate_state_memory_mb` function for every design. The estimate uses the compiled DSL state declarations, the matrix-parameter shapes of the 1.1B target model, and FP32 state storage. It measures persistent optimizer state rather than total training memory, activations, or temporary operator workspaces.

Muon maintains one momentum buffer, with an estimated state footprint of 1905.76 MB. AnnealSink, AnnealResidual, AnnealMuon, SignGate-Sink, and NormGrad-Sink have the same estimated footprint. Kronecker×Muon uses 5240.83 MB, approximately $2.75\times$ the Muon footprint, while the KronGate family uses 8575.94 MB, approximately $4.5\times$. FactorWhite-Sink adds row and column state vectors, giving an estimated footprint of 1907.80 MB.

These estimates support comparisons of persistent state requirements under a common accounting procedure. They should not be interpreted as measured peak GPU memory or as measurements of optimizer throughput.

C AI Use Statement

Large language models were used as research agents within OPTScientist and the OpenEvolve baselines, as described in the main paper. These agents proposed optimizer hypotheses, implemented candidate programs, and recorded experimental lessons using GLM-5.2. Generative AI tools also assisted with manuscript drafting, translation, language revision, and feedback on mathematical exposition and experimental presentation. The authors take responsibility for the final manuscript, experimental claims, and accompanying artifacts.