optimize_anything: A Unified Framework for LLM-Based Mathematical and Scientific Optimization

Lakshya A Agrawal [*1] **Donghyun Lee** [*1] **Shangyin Tan** [*1] **Wenjie Ma** [1] **Karim Elmaaroufi** [1] **Rohit Sandadi** [1] **Sanjit A. Seshia** [1] **Koushik Sen** [1] **Dan Klein** [1] **Ion Stoica** [1] **Joseph E. Gonzalez** [1] **Omar Khattab** [2] **Alexandros G. Dimakis** [1] **Matei Zaharia** [1]

Abstract

LLM-driven optimization for mathematical and scientific discovery is fragmented across specialized systems for prompt optimization, evolutionary code search, and agent-architecture search. The underlying problem in each case is the same, improving a text artifact under a scoring function, yet no existing system spans these categories under a single interface. We present `optimize_anything`, a declarative framework that does. A single framework raises GPT-4.1-mini on AIME-2025 from 46.7% to 60.0% (beating MIPROv2), discovers a circle-packing algorithm for $n{=}26$ exceeding AlphaEvolve under a matched rerun, evolves a 10-line ARC-AGI agent into a $300+$ line architecture lifting Gemini 3 Flash from 32.5% to 89.5%, and improves on Optuna in 7 of 10 hard blackbox problems. Together, we show that text optimization with LLM-based search is a general-purpose problem-solving paradigm, unifying tasks traditionally requiring domain-specific algorithms under a single framework.

1. Introduction

LLM-based optimization has produced strong but narrowly scoped systems. Prompt optimizers such as GEPA (Agrawal et al., 2026) and MIPROv2 (Opsahl-Ong et al., 2024) target system prompts. Evolutionary code-search systems such as FunSearch (Romera-Paredes et al., 2024) and AlphaEvolve (Novikov et al., 2025) target Python algorithms. Agent-architecture search systems such as ADAS (Hu et al., 2024) and AFlow (Zhang et al., 2025) target agent graphs. Classical hyperparameter optimizers such as Optuna (Akiba et al., 2019) target numerical configurations. Each framework imposes its own abstractions (island topologies, evolve-block markers, framework-specific feedback hooks) and does not extend to artifact types outside its design scope.

The underlying optimization problem is the same across these categories. The goal is to improve a text artifact x under a scoring function $f(x)$. The artifact may be a system prompt, a Python program, an agent, or a numerical solver, and the scoring function may be exact answer matching, a sum of radii, held-out accuracy, or wall-clock benchmarking. A single algorithm should drive all of them.

We present `optimize_anything`, a declarative framework that subsumes these specialties under one interface. The user supplies a seed artifact (or, in seedless mode, a natural-language objective), an evaluator that returns a score and optional diagnostic feedback (Side Information, SI), and optionally a dataset. The framework handles candidate selection, reflection, search, and proposal. The same call applies whether the artifact is a prompt, an algorithm, an agent, or an image.

Contributions.

1. **A unified system across four optimization categories.** `optimize_anything` matches or exceeds the corresponding purpose-built tool on four mathematical and scientific tasks: AIME-2025 prompt evolution ($46.67\% \rightarrow 60.0\%$ on GPT-4.1-mini, against MIPROv2 at 51.3%, §4.1), circle packing for $n{=}26$ (exceeding AlphaEvolve's published solution under a matched-proposer rerun against OpenEvolve, §4.2), an ARC-AGI agent evolved from a 10-line seed to 89.5% test accuracy (§4.3), and numerical solver synthesis that improves on Optuna in 7 of 10 hard problems (§4.4).

2. **Two optimization modes under one interface.** Single-task search (the artifact is the solution) and generalization (the artifact must perform on unseen inputs) are both selected purely by what the user passes in, not by switching frameworks.

[*]Equal contribution [1]UC Berkeley, USA [2]MIT, USA. Correspondence to: Lakshya A Agrawal <lakshyaaagrawal@berkeley.edu>.

The 3rd *AI for Math Workshop* at the 43rd International Conference on Machine Learning (ICML), Seoul, South Korea, 2026.
Copyright 2026 by the author(s).

3. **Side information as a first-class evaluator contract.** Prior frameworks expose diagnostic feedback through ad-hoc, framework-specific mechanisms. `optimize_anything` elevates it to a uniform API contract through which any diagnostic (compiler errors, geometric constraint violations, per-puzzle traces, rendered images) flows to the proposer. An ablation on circle packing isolates this contribution (Appendix B).

We achieve these results by extending the Pareto-based search of Agrawal et al. (2026) (originally studied for prompt optimization only) to arbitrary text artifacts. Section 3 details the algorithm, and Section 4 the experiments. The appendix collects ablations and discovered artifacts.

2. Related Work

LLM-based evolution for math and science. FunSearch (Romera-Paredes et al., 2024) pioneered LLM-driven search over Python functions to discover mathematical constructions, including new bounds for the cap-set problem. AlphaEvolve (Novikov et al., 2025) extended the paradigm with Gemini models and island-based MAP-Elites (Mouret & Clune, 2015), improving a 56-year-old matrix multiplication bound and tackling Google infrastructure problems. OpenEvolve (Sharma, 2025) provides a model-agnostic open-source reimplementation, and ShinkaEvolve (Lange et al., 2025) adds novelty-based rejection sampling. All operate exclusively on code artifacts in single-task mode, and expose framework-specific abstractions (island topologies, prompt samplers, evolve-block markers). `optimize_anything` extends this paradigm to non-code artifacts under a declarative interface that spans prompts, code, and agents, and supports both single-task and generalization modes.

Prompt optimization for mathematical reasoning. GEPA (Agrawal et al., 2026) combines reflective mutation with a Pareto-based search technique for prompt optimization, outperforming MIPROv2 (Opsahl-Ong et al., 2024) and GRPO (Shao et al., 2024), and is the optimization backend that `optimize_anything` extends. Related prompt-optimization methods include OPRO (Yang et al., 2024), APE (Zhou et al., 2023), ProTeGi (Pryzant et al., 2023), PromptBreeder (Fernando et al., 2023), and TextGrad (Yuksekgonul et al., 2024), which uses LLM-generated "gradients". None apply to non-prompt artifacts such as agent code or numerical algorithms.

Self-improving and reflective agents. Reflexion (Shinn et al., 2023) and Self-Refine (Madaan et al., 2023) treat reflection as an inner loop within a single trajectory. ADAS (Hu et al., 2024) and AFlow (Zhang et al., 2025) search over agent architectures. `optimize_anything` treats reflection as part of an outer evolutionary loop and treats the entire agent (code, control flow, prompts) as one optimizable text artifact (§4.3).

Mathematical constructions and benchmarks. Circle packing in a unit square is a classical packing problem with published bounds and configurations for many n (Graham & Lubachevsky, 1996; Galiev & Lisafina, 2013); AlphaEvolve reports a strong solution for $n{=}26$ that we use as the comparison point. AIME problems are a standard NL mathematical reasoning benchmark. ARC-AGI (Chollet, 2019) probes abstract reasoning. EvalSet (McCourt, 2016) is a 56-problem blackbox optimization benchmark used in the Optuna (Akiba et al., 2019) ecosystem; we compare `optimize_anything`-evolved solvers against Optuna on it.

3. Method

3.1. The `optimize_anything` API

At its simplest, `optimize_anything` requires a seed artifact and an evaluator. The evaluator takes a candidate string and returns a score (higher is better) together with an optional Side Information (SI) dictionary containing diagnostic feedback the proposer reads during reflection.

```python
import optimize_anything as oa

def evaluate(candidate: str) -> tuple[float, dict]:
    result = execute_code(candidate)
    return result.score, {
        "Error": result.stderr,
        "Output": result.stdout,
        "Runtime": f"{result.time_ms:.1f}ms",
    }

result = oa.optimize_anything(
    seed_candidate="<your artifact>",
    evaluator=evaluate,
)
```

SI can include open-ended text, structured data, multiple sub-scores, or images (via `oa.Image`) for vision-capable LLMs. The full signature additionally accepts `dataset`, `valset`, a natural-language `objective` (for seedless bootstrap), and `background` for domain knowledge. The user declares the *what*; `optimize_anything` handles the *execution*.

Two optimization modes. Which mode is active depends solely on whether `dataset` and `valset` are provided. In **single-task search** (no dataset), the artifact *is* the solution; circle packing is one example. In **generalization**, both `dataset` and `valset` are given and the artifact must perform on unseen inputs (e.g., AIME 2025 evaluated after training on AIME 2022 through 2024).

3.2. Problem Formulation

Let $\mathcal{X}$ denote the space of text artifacts. An evaluator $f : \mathcal{X} \times (\mathcal{E} \cup \{\bot\}) \to \mathbb{R} \times \mathcal{I}$ maps an artifact $x \in \mathcal{X}$ and optional example e to a score $s(x, e) \in \mathbb{R}$ and side information $\iota(x, e) \in \mathcal{I}$. **Single-task search.** $\mathcal{E} = \emptyset$ and we maximize $s(x)$. **Generalization.** Given $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{val}}$, we search using $\mathcal{D}_{\text{train}}$ feedback while measuring performance on $\mathcal{D}_{\text{val}}$.

3.3. Side Information as the Text Gradient

Gradient descent reduces diagnostic context to a single scalar. The optimizer knows *that* a candidate failed, but not *why*. LLM-evolution frameworks changed this by feeding execution outputs into LLM proposers, but when an LLM reads a compiler error, diagnoses a logic bug, and proposes a targeted fix, the process is closer to a mathematician iterating on a proof than to blind evolution.

`optimize_anything` elevates this to a first-class API contract. The evaluator returns both a score and a `side_info` dictionary containing any diagnostic the evaluator can produce, including compiler errors and runtime exceptions, per-test-case or per-puzzle sub-scores, constraint violations (e.g., for circle packing, overlap distances and boundary violations), and rendered images for VLM proposers. SI is optional. Evaluators that return only a score remain valid, and existing `print()` statements can be captured automatically. SI is the text-optimization analogue of the gradient. Where gradients tell a numerical optimizer which direction to move, SI tells the LLM proposer *why* a candidate failed and *how* to fix it.

3.4. Backend

`optimize_anything` is backend-agnostic. Any LLM-based optimizer that consumes candidates and produces revised candidates can be plugged in through the adapter layer. The experiments in this paper use GEPA (Agrawal et al., 2026) as the backend, which combines reflective LLM mutation with Pareto-based candidate selection. The Pareto frontier is built over per-example scores in generalization mode and over per-metric sub-scores (drawn from SI) in single-task mode; in either case GEPA maintains a frontier of non-dominated candidates and samples candidates for mutation in proportion to their frequency on the frontier. Each reflection step presents the proposer with a minibatch of 2 to 3 examples (or sub-scores, in single-task mode) and their associated SI, and the proposer returns a revised candidate that the system evaluates and inserts into the frontier if it improves on the minibatch.

To carry GEPA from prompt-only search to arbitrary text artifacts, `optimize_anything` adds (i) a refiner step that catches common LLM generation artifacts (malformed code blocks, syntax errors) before evaluation, which is essential when minor formatting causes total evaluation failure on code artifacts, (ii) content-addressed caching of evaluator outputs, (iii) SI as a typed primitive in the API contract rather than an ad-hoc string, and (iv) the adapter layer that isolates backend-specific selection logic from the user-facing interface.

Table 1. Summary across domains. Mode: S = single-task, G = generalization.

Domain	Mode	Proposer	Key Result
AIME (§4.1)	G	GPT-5.1	46.7% → 60.0%
Circle Pack. (§4.2)	S	GPT-5.1	beats AlphaEvolve
ARC-AGI (§4.3)	G	Gemini 3 F	32.5% → 89.5%
Blackbox Math (§4.4)	S	GPT-5.1	7/10 hard vs. Optuna

4. Experiments

We evaluate `optimize_anything` on four mathematical and scientific tasks spanning both optimization modes. Table 1 gives the headlines.

4.1. AIME 2025 Prompt Evolution

Setup. We optimize a system prompt for GPT-4.1-mini on the American Invitational Mathematics Examination. Training uses AIME 2022 through 2024 (90 problems), and the held-out test uses AIME 2025 (30 problems). The proposer is GPT-5.1, decoding is greedy, and the metric is exact final-answer match. SI returns each problem statement, the model's reasoning chain, extracted answer, ground truth, and a correct/incorrect flag.

Results. Prompt evolution improves GPT-4.1-mini from 46.67% to $\mathbf{60.00}\%$ on AIME 2025 (Figure 1), a $+13.3$pp gain from changing only the system prompt. This exceeds MIPROv2 (Opsahl-Ong et al., 2024) (51.33% on the same benchmark). The evolved prompt (Appendix E) is a structured 6-rule reasoning framework (restate the problem, define variables, justify each step, handle dead ends explicitly, prefer exact algebraic reasoning, isolate the final answer) obtained without manual prompt engineering.

4.2. Circle Packing (Mathematical Construction)

Setup. Pack n=26 non-overlapping circles inside the unit square to maximize $\sum_i r_i$. `optimize_anything` evolves the packing *algorithm* (Python code), executed by the evaluator, which returns the sum of radii plus geometric SI (circle positions, radii, pairwise overlap distances, boundary violations, and a rendered visualization). This is single-task search, in which the artifact *is* the solution.

Results. `optimize_anything` reaches a score of $\mathbf{2.63598}+$, exceeding AlphaEvolve's published solution and OpenEvolve's and ShinkaEvolve's reported solutions

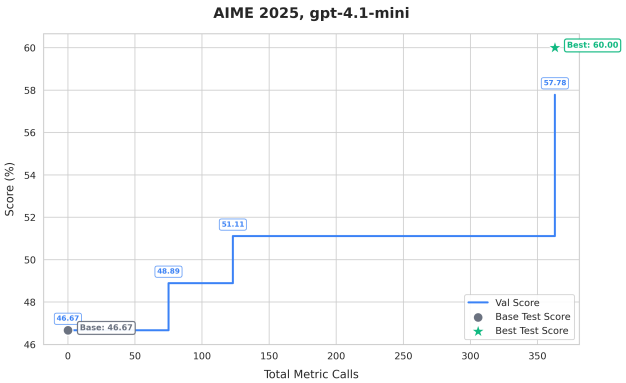

Figure 1. AIME prompt evolution for GPT-4.1-mini. Validation improves from 46.67% to 57.78%; held-out AIME 2025 reaches 60.00%.

Table 2. Controlled circle packing (n=26); both systems use GPT-5.1.

Metric	`optimize_anything`	OpenEv.@100	OpenEv.@200
Evaluations	63 / 100	100	200
Best sum of radii	**2.63598**	2.4583	2.6307
Cost	$3.18	$1.98	$6.85

for the same n (Figure 2). The discovered algorithm is a bilevel optimizer combining an LP over radii (with dual-variable sensitivities providing exact gradients for L-BFGS-B optimization over centers) with block SLP trust-region boosts on the worst circles, CMA-ES exploration, and six diverse seeding strategies (hexagonal, edge-ring, farthest-point, corner-spokes; see Appendix F.1).

Controlled comparison with OpenEvolve. To rule out proposer-LLM confounds in comparing against published numbers, we reran OpenEvolve under matched conditions using the same proposer LLM (GPT-5.1). Table 2 shows `optimize_anything` achieves 2.63598 in 63 evaluations ($3.18), while OpenEvolve at 200 iterations ($6.85) only reaches 2.6307. OpenEvolve converges to local refinement of its initial heuristic, while `optimize_anything`'s search retains diverse algorithmic families (LP, SLP, CMA-ES, bilevel L-BFGS) across iterations and ultimately produces the bilevel-LP hybrid.

An SI ablation isolating the contribution of diagnostic feedback to this result appears in Appendix B.

4.3. ARC-AGI Agent Architecture

Setup. We treat the *entire agent* (code, sub-agent architecture, control flow, helper functions, and prompts) as one text artifact. The artifact must generalize to unseen ARC-AGI (Chollet, 2019) puzzles (generalization mode). The seed is a naive 10-line agent (one LLM call). The proposer and underlying agent model are both Gemini 3 Flash. SI

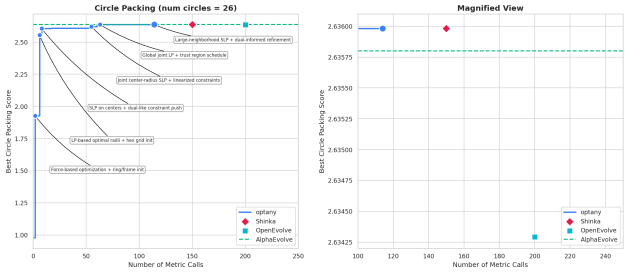

Figure 2. Circle packing (n=26). `optimize_anything` reaches a higher sum of radii in fewer evaluations than AlphaEvolve, ShinkaEvolve, and OpenEvolve.

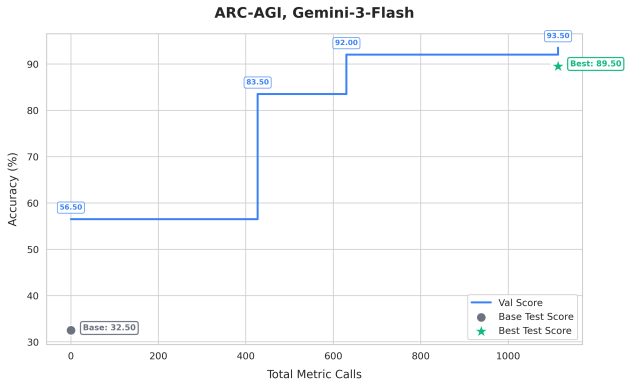

Figure 3. ARC-AGI agent evolution with Gemini 3 Flash. Validation reaches 93.5% and held-out test improves 32.5% $\rightarrow$ 89.5%.

returns training and test grid examples, per-puzzle scores, internal model outputs, LLM costs, error tracebacks, and code-execution results.

Results. `optimize_anything` evolves the seed into a 300+ line system implementing a 4-stage pipeline. The stages are (i) rule induction via pattern analysis, (ii) code generation with `exec()`-based verification, (iii) iterative debugging with up to two fix attempts, and (iv) structured fallback from code-first to direct LLM prediction. Held-out test accuracy improves from 32.5% to **89.5**%, a +57pp gain (Figure 3). The discovered architecture instantiates verify-then-fallback and iterative refinement, patterns that typically require manual engineering.

4.4. Blackbox Mathematical Optimization

Setup. Single-task search on the 56-problem EvalSet (McCourt, 2016) benchmark against Optuna (Akiba et al., 2019). Rather than tuning parameters within a fixed algorithm, `optimize_anything` evolves the *solver code* itself, discovering bespoke algorithms per problem.

Results. With 8,000 evaluations per problem, `optimize_anything` ties Optuna on 40 of 56 problems, wins on 7, and loses on 9. We further evaluate on a hard subset of 10 problems, identified as the cases

where Optuna's solution under a reduced 2,000-evaluation budget falls well below its 8,000-evaluation result (i.e., problems whose landscape is most resistant to Optuna's default TPE-CMA-ES schedule). Under the same 2,000-evaluation budget on this subset, `optimize_anything` produces better solutions on 7 of 10. Optuna's fixed TPE-CMA-ES pipeline fails in predictable structural ways (per-dimension TPE sampling converges to trap basins, CMA-ES assumes smooth unimodal landscapes), whereas `optimize_anything` selects per-problem solvers, for example L-BFGS-B for boundary optima and multi-start for deceptive traps.

5. Limitations

`optimize_anything` inherits four limitations from LLM-based optimization. First, proposal quality depends on the proposer LLM, and weaker models produce weaker candidates (proposer-sensitivity analysis in Appendix D). Second, evaluation cost can be high when the evaluator is expensive ($144 for ARC-AGI). LLM-based optimization is sample-efficient relative to gradient-free search, but absolute cost still matters. Third, the artifact must be representable as text, and optimization of continuous parameters or binary artifacts requires a text-based proxy. Fourth, designing effective SI still requires domain expertise, although `optimize_anything` trades *optimization* expertise (no backends to tune, no hyperparameters) for *domain* expertise, since the user surfaces the diagnostics they already understand. For mathematical and scientific discovery this is often the more natural division of labor.

6. Conclusion

We presented `optimize_anything`, a declarative interface (seed artifact, evaluator, and optional dataset) that matches or exceeds purpose-built tools across AIME, circle packing, ARC-AGI agent synthesis, and blackbox mathematical optimization. The framework contributes (i) two optimization modes (single-task and generalization) selected by the user's data inputs, (ii) Side Information as a first-class evaluator contract, and (iii) a backend-agnostic adapter layer; the experiments here use GEPA as the search backend. Code accompanies this submission under an anonymized identifier.

References

Agrawal, L. A., Tan, S., Soylu, D., Ziems, N., Khare, R., Opsahl-Ong, K., Singhvi, A., Shandilya, H., Ryan, M. J., Jiang, M., Potts, C., Sen, K., Dimakis, A. G., Stoica, I., Klein, D., Zaharia, M., and Khattab, O. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2026.

Akiba, T., Sano, S., Yanase, T., Ohta, T., and Koyama, M. Optuna: A next-generation hyperparameter optimization framework, 2019. URL https://arxiv.org/abs/1907.10902.

Chollet, F. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

Fernando, C., Banarse, D., Michalewski, H., Osindero, S., and Rocktäschel, T. Promptbreeder: Self-referential self-improvement via prompt evolution, 2023. URL https://arxiv.org/abs/2309.16797.

Galiev, S. I. and Lisafina, M. S. Linear models for the approximate solution of the problem of packing equal circles into a given domain. *European Journal of Operational Research*, 230(3):505–514, 2013.

Graham, R. L. and Lubachevsky, B. D. Dense packings of equal disks in an equilateral triangle: from 22 to 34 and beyond. *The Electronic Journal of Combinatorics*, 2, 1996.

Hu, S., Lu, C., and Clune, J. Automated design of agentic systems. In *arXiv preprint arXiv:2408.08435*, 2024.

Lange, R. T., Imajuku, Y., and Cetin, E. Shinkaevolve: Towards open-ended and sample-efficient program evolution, 2025. URL https://arxiv.org/abs/2509.19349.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

McCourt, M. Optimization test functions. https://github.com/sigopt/evalset, 2016. URL https://github.com/sigopt/evalset.

Mouret, J.-B. and Clune, J. Illuminating search spaces by mapping elites, 2015. URL https://arxiv.org/abs/1504.04909.

Novikov, A., Vũ, N., Eisenberger, M., Dupont, E., Huang, P.-S., Wagner, A. Z., Shirobokov, S., Kozlovskii, B., Ruiz, F. J. R., Mehrabian, A., Kumar, M. P., See, A., Chaudhuri, S., Holland, G., Davies, A., Nowozin, S., Kohli, P., and Balog, M. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

Opsahl-Ong, K., Ryan, M. J., Purtell, J., Broman, D., Potts, C., Zaharia, M., and Khattab, O. Optimizing instructions and demonstrations for multi-stage language model programs, 2024. URL https://arxiv.org/abs/2406.11695.

Pryzant, R., Iter, D., Li, J., Lee, Y. T., Zhu, C., and Zeng, M. Automatic prompt optimization with "gradient descent" and beam search. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2023.

Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M. P., Dupont, E., Ruiz, F. J. R., Ellenberg, J. S., Wang, P., Fawzi, O., Kohli, P., and Fawzi, A. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024. doi: 10.1038/s41586-023-06924-6.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y. K., Wu, Y., and Guo, D. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

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

Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366.

Yang, C., Wang, X., Lu, Y., Liu, H., Le, Q. V., Zhou, D., and Chen, X. Large language models as optimizers, 2024. URL https://arxiv.org/abs/2309.03409.

Yuksekgonul, M., Bianchi, F., Boen, J., Liu, S., Huang, Z., Guestrin, C., and Zou, J. Textgrad: Automatic "differentiation" via text, 2024. URL https://arxiv.org/abs/2406.07496.

Zhang, J., Xiang, J., Yu, Z., Teng, F., Chen, X., Chen, J., Zhuge, M., Cheng, X., Hong, S., Wang, J., Zheng, B., Liu, B., Luo, Y., and Wu, C. Aflow: Automating agentic workflow generation, 2025. URL https://arxiv.org/abs/2410.10762.

Zhou, Y., Muresanu, A. I., Han, Z., Paster, K., Pitis, S., Chan, H., and Ba, J. Large language models are human-level prompt engineers, 2023. URL https://arxiv.org/abs/2211.01910.

A. Use of Generative AI

The authors made use of generative-AI tools (ChatGPT, Gemini, Claude, and Cursor) to draft sections of this paper, including text, tables, graphs, and code. Experiment design and the technical content were the authors' own.

B. Side Information Ablation (Circle Packing)

To isolate the contribution of SI, we re-run the circle packing procedure of §4.2 without diagnostic feedback. With SI, `optimize_anything` reaches the global best ($\sum_i r_i = 2.63598$). Without SI (score only), the same procedure stalls at 93.96% of best. SI reveals which constraint is failing (overlap, boundary, collapsed radius), and the proposer targets that specific failure mode rather than resorting to undirected mutations.

Table 3. SI vs. score-only on circle packing (n=26).

Metric	With SI	Score only
Sum of radii (% of best)	**100%**	93.96%

C. Optimization Trajectory Analysis (Circle Packing)

Mechanism 1: SI enables targeted algorithmic shifts. SI works because it reveals *which* failure mode to address next. In circle packing, SI-driven reflection produces a characteristic pattern: collapsed radii $\rightarrow$ switch to LP; poor center placement $\rightarrow$ switch to SLP; local saturation $\rightarrow$ switch to bilevel L-BFGS. Without SI, the proposer observes only that the score changed and resorts to undirected mutations.

Mechanism 2: Multi-module Pareto leapfrogging. `optimize_anything` optimizes both the code artifact and a refiner prompt on the shared Pareto front. In circle packing this creates a productive leapfrogging dynamic: the refiner discovers LP while the code module is still a weak heuristic (code$= 0.98$, refiner$= 1.93$); the code module then absorbs the LP approach ($\rightarrow 2.61$); the refiner pushes further with SLP ($\rightarrow 2.63$); the code module absorbs SLP and reaches the record. Even broken-code mutations (score $= 0$) are recovered by the refiner and retained as a safety net.

Mechanism 3: Pareto diversity. At convergence the front retains candidates from multiple algorithmic families (greedy, LP, SLP, bilevel L-BFGS, CMA-ES) across quality dimensions (max score, mean score, EMA stability, improvement rate), ensuring structurally diverse parents for proposals.

D. Proposer Sensitivity and Cost

Table 4. Proposer LLM sensitivity. GPT-5-nano reduces cost but underperforms.

Model	Task	Performance	Cost
GPT-5.1	AIME	46.67% $\rightarrow$ **60.0%**	\$6.44
GPT-5-nano	AIME	46.67% $\rightarrow$ 50.0%	\$3.71
GPT-5.1	Circle Pack.	0.98 $\rightarrow$ **2.636**	\$6.00
GPT-5-nano	Circle Pack.	0.98 $\rightarrow$ 2.512	\$0.50

Table 5. Total optimization cost per experiment.

Task	Total Cost
Numerical Blackbox	\$1
Circle Packing (controlled run, §4.2)	\$3.18
Circle Packing (proposer sensitivity sweep)	\$6.00
AIME (Generalization)	\$6.44
ARC-AGI (Generalization)	\$144.70

E. Optimized AIME Prompt

Optimized Prompt for AIME (GPT-4.1-mini)

```
Solve the math problem carefully and thoroughly.  Your goal is to produce a correct, well-structured
solution that leads unambiguously to the requested final result.
Follow these rules:
1.  Restate the problem briefly in your own words.
2.  Set up notation and equations cleanly before manipulating them.  - Define variables explicitly.  -
State all constraints (integrality, ranges, geometric conditions) before using them.
3.  Show clear, logically ordered reasoning.  - Justify each important algebraic or geometric step.  -
When you split into cases, state why each case is necessary.  - If you invoke a known theorem (Ptolemy,
Power of a Point, similarity, Vieta), name it and show how it applies.
4.  Handle dead ends correctly.  - If you realize a line of reasoning leads to a contradiction,
explicitly say so.  - Then restart from the last correct point; do not guess.
5.  Keep the reasoning focused and minimal while still being rigorous.  - Avoid unnecessary numerical
approximations if an exact approach is available.  - Prefer algebraic or structural arguments over
trial-and-error.  - Test candidate values only after deriving strong constraints.
6.  At the end, isolate the answer:  - Provide the final answer as a single number or expression on its
own line.  - Do not include any extra words on that final line.
```

F. Discovered Solutions

We present excerpts of the optimized artifacts.

F.1. Circle Packing Algorithm (excerpt)

The evolved algorithm (480+ lines) is a bilevel optimizer that jointly optimizes circle centers and radii for n=26 in a unit square, starting from a simple greedy seed.

Evolved Circle Packing Algorithm (excerpt)

```python
def main(timeout, current_best_solution):
    """Bilevel L-BFGS with exact LP sensitivities +
    SLP block boosts + CMA/Evolution fallback."""
    n = 26
    # LP for optimal radii given centers; returns duals.
    def solve_radii_lp(centers, need_duals=False):
        # Boundary: r_i <= min(x_i, 1-x_i, y_i, 1-y_i)
        # Pairwise: r_i + r_j <= ||c_i - c_j||
        # Objective: maximize sum(r)
        res = linprog(c_obj, A_ub=A_ub, b_ub=b_ub, ...)
        return r, success, {'dual': res.ineqlin.marginals}

    # Gradient from LP duals: sensitivity of sum(r) to centers.
    def gradient_from_duals(centers, dual_vec):
        # Boundary: g[i,0] += (dual_left - dual_right)
        # Pairwise: g[i] += lambda_k * unit_vec(i->j)
        return g

    # Bilevel: L-BFGS-B over centers, LP over radii.
    def lbfgs_bilevel(centers_init, max_iters=300):
        def f_and_g(flat):
            r, _, info = solve_radii_lp(centers, need_duals=True)
            g = gradient_from_duals(centers, info['dual'])
            return -score, -g.reshape(-1)
        minimize(f_and_g, method='L-BFGS-B', bounds=bounds)

    # Block SLP, CMA-ES, 6 seeding strategies, relocation,
    # evolutionary exploration, final polish. See artifact.
```

F.2. ARC-AGI Agent Architecture

The optimized agent grew from a 10-line seed to a 300+ line system implementing rule induction, code generation with `exec()` verification, iterative debugging with up to two fix attempts, and structured fallback from code-first to direct LLM prediction.

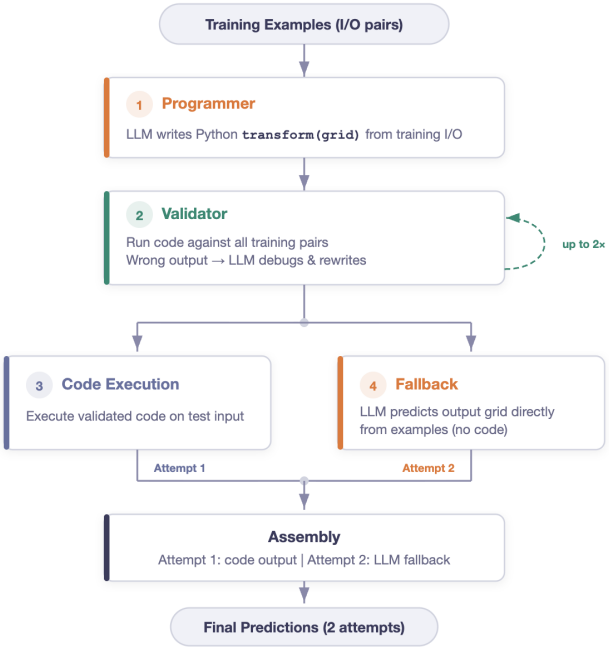

Figure 4. Architecture of the optimized ARC-AGI agent: a 4-stage pipeline with verify-then-fallback logic, evolved from a naive single-call seed.

G. Artifact Availability

Code and reproduction artifacts accompany this submission as supplementary material under an anonymized identifier. Each evaluation domain has its own subdirectory with runnable code, a `README.md` mapping the folder to the relevant section of this paper, and the saved checkpoint from the paper run.

Hardware notes. All domains run on a single CPU host with API access to the proposer LLM. The paper used GPT-5/5.1, Gemini 3 Flash, and Claude Opus 4.6 depending on domain, with exact identifiers documented per domain.