`optimize_anything`: Unified Text Optimization can Outperform Specialized Systems

Anonymous Authors[1]

Abstract

Can a single LLM-based optimization system match specialized tools across fundamentally different domains? We show that when optimization problems are formulated as improving a text artifact evaluated by a scoring function, a single AI-based optimization system—supporting single-task search, multi-task search with cross-problem transfer, and generalization to unseen inputs—achieves state-of-the-art results across six diverse tasks. Our system discovers agent architectures that nearly triple Gemini Flash's ARC-AGI accuracy (32.5% $\rightarrow$ 89.5%), finds scheduling algorithms that cut cloud costs by 40%, generates CUDA kernels where 87% match or beat PyTorch, and outperforms AlphaEvolve's reported circle packing solution (n=26). Ablations across three domains reveal that actionable side information yields faster convergence and substantially higher final scores than score-only feedback, and that multi-task search outperforms independent optimization given equivalent per-problem budget through cross-task transfer, with benefits scaling with the number of related tasks. Together, we show for the first time 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

Large language models can serve as effective optimizers when paired with automated evaluation. FunSearch (Ellenberg et al., 2025) evolves Python functions to discover mathematical constructions that surpass known bounds. AlphaEvolve (Novikov et al., 2025) extends the idea to broader code optimization, improving a 56-year-old matrix multiplication bound and designing scheduling heuristics for Google's data centers, but it operates exclusively on code artifacts, in single-task mode (one problem at a time). GEPA (Agrawal et al., 2026) achieves state-of-the-art prompt optimization with generalization to unseen inputs, but is limited to prompts; MIPROv2 (Opsahl-Ong et al., 2024) similarly targets prompt and few-shot selection. Despite strong results within their artifact types, no existing system has been applied to agent architectures, numeric optimization, or image gen, and no single system has demonstrated effectiveness across fundamentally different domains simultaneously.

We observe that a wide range of problems can be formulated as optimizing a text artifact. Whether the artifact is a CUDA kernel, a cloud scheduling policy, an agent architecture, Scalable Vector Graphics (SVGs), or a system prompt, the structure is the same: serialize the artifact as a string, evaluate it, and let an LLM propose improvements based on diagnostic feedback. This observation suggests a much simpler interface and a uniform algorithm is possible.

We present `optimize_anything`, a declarative API that implements this insight. The user provides a seed artifact (or, in seedless mode, just a natural-language objective), an evaluator that returns a score and optional diagnostic feedback, and optionally a dataset. The system handles prompt construction, reflection, candidate selection, and search strategy. This declarative design, inspired by DSPy's (Khattab et al., 2023) principle of *programming—not prompting*, means the same API call works whether one is optimizing an LLM prompt, an agent architecture, or an image.

Our contributions are as follows:

1. **A single LLM-based Text Optimization system matches or surpasses domain-specific tools across six fundamentally different domains.** We are the first to show that a single system (our proposed `optimize_anything`) can optimize code, prompts, agent architectures, numerical configurations, and images, achieving state-of-the-art results in each. Our system discovers agent architectures that nearly triple

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

055 056 057 058 059 060 061 062 063 064 065 066 067 068 069 070 071 072 073 074 075 076 077 078 079 080 081 082 083 084 085 086 087 088 089 090 091 092 093 094 095 096 097 098 099 100 101 102 103 104 105 106 107 108 109

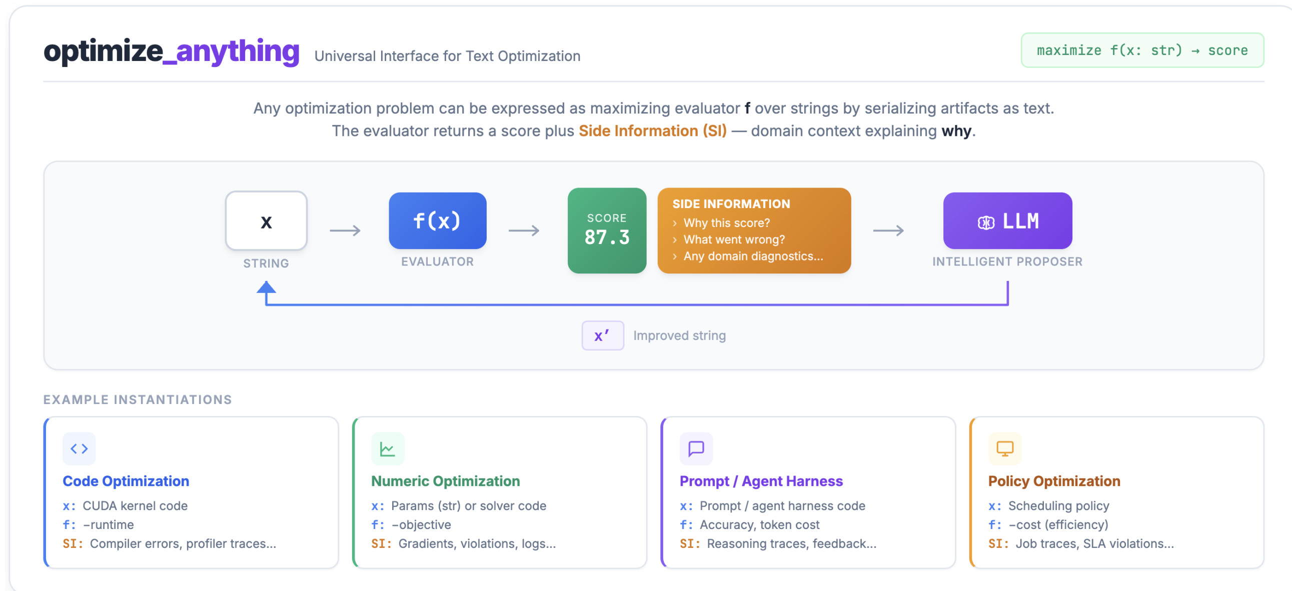


Figure 1. The `optimize_anything` loop: a text artifact x is passed to an evaluator $f(x)$ which returns a score plus diagnostic feedback (SI), which is consumed by an LLM proposer to produce an improved artifact. The same API instantiates across domains: code optimization, prompt tuning, agent architecture search, and policy discovery.

ARC-AGI accuracy (32.5% → 89.5%), finds scheduling algorithms that cut cloud costs by 40%, generates CUDA kernels where 87% match or beat PyTorch baselines, create custom solver code matching and outperforming Optuna in numerical optimization, and outperforms AlphaEvolve's solution on circle packing. This establishes LLM-based text optimization as a general-purpose problem-solving paradigm, not limited to code or prompts.

2. **Three optimization modes—single-task, multi-task, and generalization—unified under one interface, including the first multi-task mode.** Existing LLM-evolution systems each support exactly one mode. AlphaEvolve (Novikov et al., 2025), OpenEvolve (Sharma, 2025), and ShinkaEvolve (Lange et al., 2025) operate in single-task mode: optimizing one code artifact for one problem at a time. GEPA (Agrawal et al., 2026) and MIPROv2 (Opsahl-Ong et al., 2024) operate in generalization mode: optimizing a prompt to perform well on unseen inputs, but only for prompts. No prior system supports *multi-task search*, where solving a batch of related problems together enables cross-transfer of discovered optimization patterns. `optimize_anything` unifies all three modes under one interface: multi-task search on CUDA kernels outperforms independent single-task optimization given equivalent per-problem budget (§5.4), and generalization extends beyond prompts to agent architectures (§5.1) and scheduling policies (§A.2). All optimization modes are expressed through the same `optimize_anything` API.

3. **Side information as a first-class evaluator contract.** Prior frameworks support diagnostic feedback through ad-hoc, framework specific mechanisms. `optimize_anything` elevates it to a uniform API contract: any diagnostic—stack traces, profiler data, rendered images, structured error reports—flows to the proposer through one interface. Ablations across three domains (prompt optimization, circle packing, and CUDA kernels) show that actionable side information yields 4-6× faster convergence and substantially higher final performance versus score-only feedback (§5.5).

We achieve these results by extending the Pareto-based search of Agrawal et al. (2026) (originally studied only for prompt optimization) to arbitrary text artifacts, adding single-task and multi-task modes. Candidates are selected based on per-example or per-metric Pareto dominance rather than aggregate scores, preserving complementary strengths across iterations. Table 2 provides a detailed comparison.

We evaluate `optimize_anything` across six primary domains spanning all three optimization modes (Table 1), with two additional domains (blackbox mathematical optimization and 3D modeling) in the appendix as preliminary demonstrations. Key results include: (i) evolved agent architectures nearly triple Gemini Flash's ARC-AGI accuracy (32.5% → 89.5%); (ii) discovered cloud scheduling algorithms cut costs by up to 40%; (iii) 87% of generated CUDA kernels match or beat PyTorch baselines from KernelBench, with multi-task mode outperforming dedicated single-task optimization; (iv) prompt optimization improves GPT-4.1-

mini's AIME-2025 accuracy from 46.67% to 60.00%; and (v) our circle packing solution outperforms AlphaEvolve's published one, confirmed by a controlled rerun against OpenEvolve under matched conditions. Ablations across three domains show that actionable side information yields 4-6× faster convergence and substantially higher final performance versus score-only feedback, and that multi-task search benefits scale with the number of related tasks.

2. Related Work

LLM-based program evolution. AlphaEvolve (Novikov et al., 2025) pioneered the LLM-evolution paradigm, using Gemini models with island-based MAP-Elites (Mouret & Clune, 2015) to discover algorithms for Google's infrastructure. OpenEvolve (Sharma, 2025) provides an open-source reimplementation with model-agnostic support. ShinkaEvolve (Lange et al., 2025) extends the paradigm with novelty-based rejection sampling for sample efficiency and adaptive LLM ensemble selection for diversity. FunSearch (Ellenberg et al., 2025) applies evolutionary LLM search to mathematical discovery. EvoPrompting (Chen et al., 2023) evolves code for neural architecture search. All operate exclusively in single-task mode and expose framework-specific abstractions (island topologies, prompt samplers, evolve-block markers). `optimize_anything` strips the interface to its declarative essence, adds multi-task and generalization modes, and elevates diagnostic feedback to a first-class API concept.

Prompt optimization. GEPA (Agrawal et al., 2026) combines reflective mutation with a Pareto-based search technique for prompt optimization, outperforming both MIPROv2 (Opsahl-Ong et al., 2024) and GRPO (Shao et al., 2024). `optimize_anything` uses GEPA's evolutionary search algorithm as the optimization backend, extending it beyond prompts to arbitrary text artifacts. Other prompt optimization methods include OPRO (Yang et al., 2024), APE (Zhou et al., 2023), ProTeGi (Pryzant et al., 2023), and PromptBreeder (Fernando et al., 2023). TextGrad (Yuksekgonul et al., 2024) uses LLM-generated "gradients" for text optimization.

LLM self-improvement and reflection. Reflexion (Shinn et al., 2023) uses verbal reinforcement for agent self-correction. Self-Refine (Madaan et al., 2023) applies iterative self-feedback. Evolution through Large Models (Lehman et al., 2022) explores LLMs as mutation operators. `optimize_anything`'s SI mechanism generalizes these ideas by making diagnostic feedback a declarative evaluator contract rather than a hardcoded self-critique.

Agent architecture search. ADAS (Hu et al., 2024) and AFlow (Zhang et al., 2025) search over agent architectures. `optimize_anything`'s generalization mode subsumes these as special cases: the artifact is the agent code, the evaluator runs it on tasks, and the system evolves both architecture and prompts jointly.

3. The `optimize_anything` API

3.1. Core Interface

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) alongside an optional Side Information (SI) dictionary containing diagnostic feedback the proposer reads during reflection:

```
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 (VLM).

The full `optimize_anything` signature is:

```
def optimize\_anything(
   seed_candidate=None, # Starting artifact
   evaluator=..., # fn(candidate) -> Score + SI
   dataset=None, # Training examples
   valset=None, # Validation set
   objective=None, # Natural language goal
   background=None, # Domain knowledge
   config=None, # Engine settings
) -> OptimizationResult:
```

Specifically, `optimize_anything` doesn't require mutation prompts, task-specific templates, island configurations, or `EVOLVE-BLOCK` markers (all common in prior frameworks). The user declares the *what* (artifact, evaluator, domain knowledge), and `optimize_anything`, through its optimization backends, handles the *execution*.

Seedless mode. In domains where providing even a starting artifact is difficult, or where writing even a bad seed requires domain expertise (e.g., 3D modeling), the user can just provide a natural-language `objective` as an argument in place of the `seed_candidate` argument and the LLM bootstraps the first candidate from scratch. Seedless mode makes the system accessible to users who can *specify* what they want but not *implement* it. Appendix D demonstrates it on a 3D modeling task.

Table 1. Summary of experimental results across six domains. "Mode" indicates which optimization paradigm is used: S = single-task search, M = multi-task search, G = generalization. All results use `optimize_anything` with the indicated proposer LLM.

Domain	Mode	Proposer	Key Result
Agent Skills (§A.1)	G	Claude Opus 4.6	100% pass rate (+20.7pp)
Cloud Sched. (§A.2)	G	Gemini 3 Pro	40.2% cost savings
ARC-AGI (§5.1)	G	Gemini 3 Flash	89.5% test (+57pp)
AIME Prompts (§A.3)	G	GPT-5	60.0% test (+13.3pp)
CUDA Kernels (§5.2)	M	GPT-5	87% match baseline
Circle Pack. (§5.3)	S	GPT-5	Beats AlphaEvolve
Math Opt. (§C)	S	GPT-5	Wins 7/10 vs Optuna
3D Modeling (§D)	S	Claude Opus 4.6	Generates a 3D unicorn

3.2. Three Optimization Modes

Which mode is active depends solely on whether `dataset` and `valset` are provided:

Single-Task Search. No dataset. The candidate *is* the solution; the evaluator scores it directly. This is the mode that AlphaEvolve and OpenEvolve operate in. Example: in circle packing (§5.3), the artifact is the packing algorithm and the evaluator returns the packing score plus geometric diagnostics.

Multi-Task Search. A `dataset` of related tasks is provided; insights from solving one help solve the others. Example: in CUDA kernel generation (§5.2), each task is a PyTorch operation to accelerate. Multi-task mode discovers optimization patterns that transfer across problems, converging faster and solving more problems than single-task runs (§5.4). No prior LLM-evolution framework supports this mode. Architecturally, the Pareto frontier is shared across tasks for cross-transfer during proposal, but at output time each task independently selects its own best candidate from the frontier. This means multi-task search produces N specialized artifacts (one per task) that have benefited from shared optimization context, patterns discovered while optimizing task e_i are available as parents when proposing for task e_j, but each artifact can specialize to its task.

Generalization. Both `dataset` and `valset` are provided; the optimized artifact must perform well on unseen examples. This is the mode that GEPA's prompt optimization (Agrawal et al., 2026) operates in; `optimize_anything` generalizes the pattern to any text artifact. Example: in agent architecture discovery (§5.1), the artifact is the entire agent, and it must generalize to unseen ARC-AGI puzzles. The key distinction is that multi-task search yields N specialized artifacts while generalization yields one globally generalized artifact.

Table 2. Comparison of `optimize_anything` with prior LLM-based optimization frameworks across code evolution, prompt optimization, and agent architecture search systems. Only `optimize_anything` supports all three modes and provides diagnostic feedback as a first-class API concept.

Feature	AlphaEv.	OpenEv.	ShinkaEv.	GEPA	MIPROv2	TextGrad	ADAS	AFlow	`optimize_anything`
Code artifacts	✓	✓	✓						✓
Prompt artifacts				✓	✓	✓			✓
Agent artifacts							✓	✓	✓
Single-task search	✓	✓	✓			✓			✓
Multi-task search									✓
Generalization				✓	✓		✓	✓	✓
SI as API contract									✓
Image feedback									✓
Declarative interface									✓
No mutation templates				✓	✓	✓	✓	✓	✓
Open-source		✓	✓	✓	✓	✓	✓	✓	✓

4. Method

`optimize_anything` currently extends and manages information atop GEPA (Agrawal et al., 2026), an algorithm originally studied primarily in the context of prompt optimization and code search, as the optimization backend. The system overview is shown in Figure 1. While `optimize_anything`'s primary contribution is a unified interface, several concrete algorithmic modifications were necessary to generalize from prompts to arbitrary text artifacts: (1) new frontier types for single-task and multi-task search with distinct selection semantics (GEPA's Pareto-frontier selection relied on evaluation across multiple data points, whereas single-task search admits only one); (2) a refiner step that catches common LLM generation artifacts (malformed code blocks, import errors, syntax issues) before evaluation, essential for code and agent artifacts where minor formatting errors cause complete evaluation failure; (3) content-addressed evaluation caching to avoid redundant expensive rollouts; (4) SI as a first-class typed primitive enabling domain-portable proposer logic and multimodal feedback; and (5) an adapter layer between various optimization backends and the unified interface. We describe the two mechanisms that underpin effectiveness and contrast `optimize_anything` with prior frameworks.

4.1. Problem Formulation

We formalize the text optimization problem as follows. Let $\mathcal{X}$ denote the space of text artifacts (strings). An evaluator $f : \mathcal{X} \times \mathcal{E} \cup \{\perp\} \to \mathbb{R} \times \mathcal{I}$ maps an artifact $x \in \mathcal{X}$ and an (optional) example $e \in \mathcal{E} \cup \{\perp\}$ to a score $s(x, e) \in \mathbb{R}$ and actionable side information $\iota(x, e) \in \mathcal{I}$, i.e., $f(x, e) = (s(x, e), \iota(x, e))$. The three modes correspond to:

Single-task search: $\mathcal{E} = \emptyset$; maximize $s(x)$ directly. The artifact *is* the solution (e.g., a packing algorithm).

Multi-task search: Given a dataset $\mathcal{D} = \{e_1, \ldots, e_n\}$ of related problems, find an artifact $x \in \mathcal{X}$ (e.g., a kernel-generation prompt) maximizing $\frac{1}{n}\sum_{i=1}^{n} s(x, e_i)$. Cross-transfer arises because the Pareto frontier preserves patterns that work across problems.

Generalization: Given a training set $\mathcal{D}_{\text{train}}$ and a validation set $\mathcal{D}_{\text{val}} = \{e_1^{\text{val}}, \ldots, e_k^{\text{val}}\}$, find an artifact $x \in \mathcal{X}$ maximizing $\frac{1}{k}\sum_{j=1}^{k} s\big(x, e_j^{\text{val}}\big)$. Search uses feedback from $\mathcal{D}_{\text{train}}$, while $\mathcal{D}_{\text{val}}$ measures generalization to unseen examples. This generalizes classical machine learning: the artifact may be a prompt, an agent, or a policy.

4.2. Side Information (SI)

Popularly used numerical optimization methods like gradient descent reduce all diagnostic context to a single scalar. The optimizer knows *that* a candidate failed, but not *why*. For example, one cannot show a Bayesian optimizer a stack trace. LLM-evolution frameworks changed this by feeding execution results 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 an engineer iterating on a prototype than to blind evolution.

`optimize_anything` leans into this by making diagnostic feedback a first-class part of the evaluator contract. The evaluator returns both a score and a `side_info` dictionary containing any diagnostic the evaluator can produce:

- **Text:** compiler errors, runtime exceptions, profiler summaries, natural-language critiques.
- **Structured data:** per-test-case results, sub-scores for multiple objectives, execution traces.
- **Images:** rendered SVGs, 3D model screenshots, or chart visualizations, enabling VLM proposers to *see* what they are improving.

SI is the text-optimization analogue of the gradient. Where gradients tell a numerical optimizer which direction to move, SI can tell the LLM proposer *why* a candidate failed and *how* to fix it. During a dedicated reflection step, the proposer reasons over this signal to diagnose failures and propose targeted improvements.

Prior frameworks expose feedback through framework-specific mechanisms; SI provides a uniform interface that makes it trivial to surface any diagnostic. The key design choice is that SI is *opt-in but zero-friction*: evaluators that return only a score work fine, and existing `print()` statements can be captured automatically via `capture_stdio=True`.

4.3. Pareto-Based Search

Even when optimizing a single objective, evaluating candidates across multiple examples or metrics produces richer signal than a scalar aggregate. The naive approach collapses that signal into one average score and always selects the top candidate. This stalls fast: averaging hides which aspects are strong and which are weak, and the proposer tries to improve everything at once.

`optimize_anything` does two things differently. First, it tracks scores per task (from `dataset`) or per metric (from sub-scores in SI) individually and maintains a **Pareto frontier**: any candidate that is the best at *something* survives, even if its average is suboptimal. Second, each reflection step shows the proposer a minibatch of just 2–3 examples instead of all of them, enabling focused, targeted improvements on that subset.

Over iterations, the frontier accumulates complementary strengths. Candidates that excel at different tasks are preserved and their strategies recombined. This mechanism also powers multi-task search: when optimizing across related problems, the frontier preserves candidates that excel on different tasks, and strategies discovered for one problem transfer to others (§5.4).

Candidate selection. Following GEPA (Agrawal et al., 2026), candidates are selected for mutation in proportion to how often they appear on the Pareto front. Let J index the objectives used to form the Pareto scores (e.g., per-example tasks, per-metric scores, or both). Each candidate Φ induces a score $s_j(\Phi)$ for every $j \in J$. Let $\mathcal{P}$ denote the set of Pareto-nondominated candidates under these objectives. For each objective $j \in J$, let $\mathcal{B}[j]$ be the set of candidates in $\mathcal{P}$ that achieve the best score on j. We sample candidates with probability proportional to $|\{j \in J : \Phi \in \mathcal{B}[j]\}|$, focusing exploration on broadly effective solutions.

Reflection and mutation. Given a selected candidate Φ and a minibatch $\mathcal{M}$ of examples, the system executes Φ on $\mathcal{M}$, collects scores and SI, and presents them to the proposer LLM in a structured reflection prompt. The proposer diagnoses failures using the SI and produces an updated artifact Φ'. If Φ' improves on the minibatch, it is fully evaluated and added to the candidate pool.

5. Experiments

We evaluate `optimize_anything` across six primary domains spanning all three optimization modes. For each, we describe the artifact, evaluator, SI design, and results. Below we present the four flagship domains; three additional generalization-mode experiments—Coding Agent Skills on the Bleve repository (Appendix A.1), Cloud Scheduling

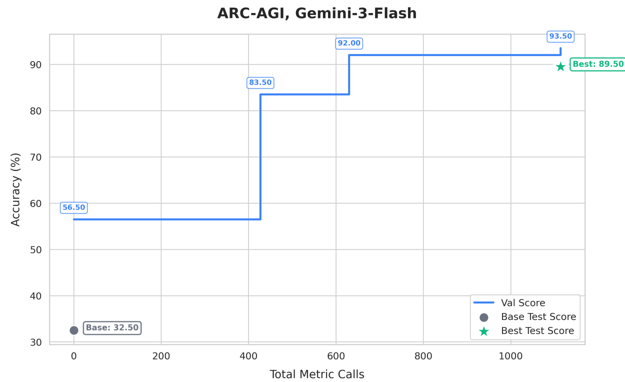


Figure 2. ARC-AGI agent architecture evolution with Gemini 3 Flash. Validation accuracy reaches 93.5%; test accuracy improves from 32.5% to 89.5%.

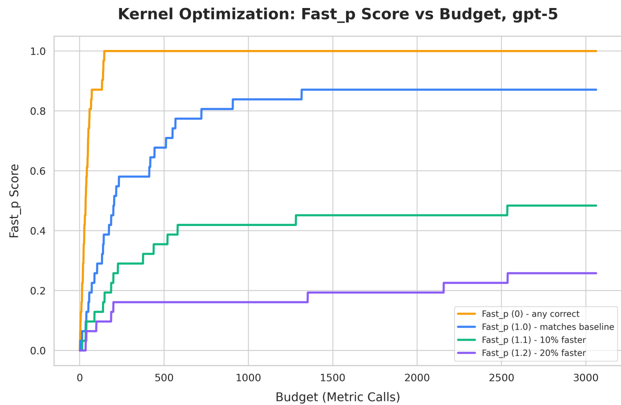


Figure 3. KernelBench results (GPT-5 as proposer). $\text{Fast}_p(s)$: fraction of kernels achieving speedup $\geq s$. 87% match baseline; 25% are 20%+ faster.

Algorithms (CloudCast, Can't Be Late; Appendix A.2), and AIME Prompt Optimization (Appendix A.3)—are detailed in Appendix A; headline numbers for all six domains are summarized in Table 1. We then present ablation studies on multi-task search (§5.4), SI (§5.5), and proposer sensitivity and cost (Appendix A.5), followed by an analysis of the optimization mechanisms (§6). Optimized solutions are presented in the Appendix K.

5.1. ARC-AGI Agent Architecture (Generalization)

Setup. Rather than optimizing a prompt, we optimize the *entire agent system*: code, sub-agent architecture, control flow, helper functions, and prompts are all treated as a single text artifact. The optimization objective is for the artifact to generalize to unseen ARC-AGI (Chollet, 2019) puzzles.

SI design. Training/test grid examples, per-puzzle scores, internal model outputs, LLM costs, error tracebacks, and code execution results.

Results. Using Gemini 3 Flash as both the proposer and the underlying agent model, `optimize_anything` starts for a naive 10-line agent seed (one LLM call) and iteratively designs it into a 300+ line system consisting of 4 components along with fallbacks. The test accuracy improves from 32.5% to **89.5%**, a 57 percentage point gain (Figure 2). The optimized architecture implements a 4-stage pipeline: (1) rule induction via pattern analysis, (2) code generation with `exec()`-based verification, (3) iterative debugging with up to 2 fix attempts, and (4) structured fallback from code-first to direct LLM prediction. This represents a qualitative leap: the system discovers architectural patterns (verify-then-fallback, iterative refinement) that typically require manual engineering iterations.

5.2. CUDA Kernel Generation (Multi-Task Search)

Setup. We generate CUDA kernels for 31 reference PyTorch operations from KernelBench (Ouyang et al., 2025), evaluated on a V100 32GB GPU. The 31 problems span diverse operations: matrix multiplications, convolutions, reductions, element-wise ops, and normalization layers. Under the hood, `optimize_anything` evolves the prompt that drives kernel generation; in multi-task mode, insights discovered for one problem (e.g., how to handle memory coalescing) transfer to others automatically through the shared Pareto frontier.

SI design. The evaluator compiles the generated kernel, runs correctness tests (max absolute error vs. PyTorch reference), and benchmarks wall-clock time. SI includes: (i) NVCC compiler errors with line numbers, (ii) correctness test failures with actual vs. expected outputs, (iii) relevant CUDA documentation snippets, and (iv) speedup ratio vs. the PyTorch baseline.

Results. 87% of generated kernels match or beat the PyTorch baseline performance; 48% achieve 10%+ speedups, and 25% achieve 20%+ speedups (Figure 3). The evolved kernels employ techniques such as float4 vectorization, two-pass algorithms (compute statistics, then normalize), warp shuffle reductions, and shared memory tiling. Multi-task mode's advantages are analyzed in §5.4.

5.3. Circle Packing (Single-Task Search)

Setup. The task is to pack n=26 circles while maximizing the sum of radii within a unit square. `optimize_anything` optimizes the packing algorithm code; the evaluator executes the proposed packing code, and returns the score plus geometric diagnostics.

SI design. Circle positions, radii, constraint violations, overlap distances, boundary violations, and a rendered visu-

Table 3. Controlled comparison of `optimize_anything` vs. OpenEvolve on circle packing (n=26), both using GPT-5.1 as proposer.

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

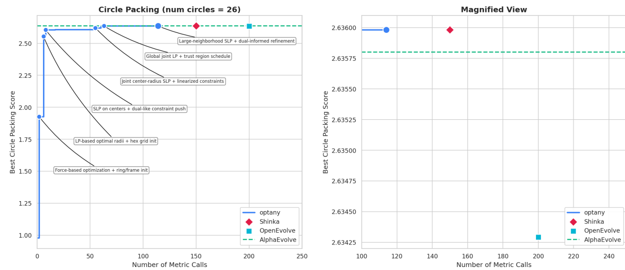


Figure 4. Circle packing (n=26). `optimize_anything` outperforms AlphaEvolve's, ShinkaEvolve's, and OpenEvolve's solution, reaching a higher score with fewer evaluations.

alization of the packing.

Results. `optimize_anything` reaches a score of 2.63598+, outperforming AlphaEvolve's, OpenEvolve's, and ShinkaEvolve's reported solution (Figure 4). The optimized algorithm is a bilevel optimizer: an LP over radii with dual-variable gradients for L-BFGS-B center optimization, augmented by CMA-ES exploration and diverse seeding strategies.

Controlled comparison with OpenEvolve. To address concerns about comparing against published rather than reproduced results, we ran OpenEvolve (open-source reimplementation of AlphaEvolve) under matched conditions using the same proposer LLM (GPT-5.1). As shown in Table 3, `optimize_anything` achieved a superior score (2.63598) in just 63 evaluations (costing $\tilde{\$}$3.18), while OpenEvolve failed to match this performance even when given over three times the evaluation budget (200 iterations, costing $6.85, reaching only 2.6307).

5.4. Ablation: Multi-Task vs. Single-Task Search

We re-optimize the 10 best multi-task problems from scratch in single-task mode with equivalent per-problem budget. Figure 8 (in Appendix A) shows that multi-task mode consistently outperforms single-task across all speedup thresholds, with the gap widening at higher thresholds ($\text{Fast}_p(1.2)$: single-task plateaus early while multi-task continues improving). The Image Generation domain (Appendix A.4) provides a complementary multi-task example over VLM-scored visual aspects.

The mechanism is cross-transfer via the Pareto frontier: optimization patterns discovered for one kernel (e.g., vectorized memory access, warp-level reductions) are preserved on the frontier and inform proposals for other kernels. In single-task mode, each problem must independently discover these patterns.

Scaling with number of tasks. Multi-task benefits scale with the number of related tasks: MT20 (20 problems) outperforms MT10 (10 problems), which outperforms single-task, with gains most pronounced at moderate speedup thresholds (Tables 6–7 in Appendix F). Frontier size does not bottleneck scaling, as candidates are sampled by frontier frequency (e.g., ARC-AGI used 200 tasks effectively).

5.5. Ablation: Side Information

We isolate SI's contribution by comparing `optimize_anything` with and without sub-scores on Facility Support Analysis prompt optimization (Figure 9 in Appendix A.6). With SI, validation reaches 0.80 in 100 rollouts vs. $\sim$600 without; final test score is 86.32 vs. 82.5—SI accelerates convergence and improves final performance. Cross-domain ablations on circle packing and CUDA kernels (Table 4, same appendix) confirm generalization: SI achieves the optimal circle packing solution (score-only reaches 94%) and enables 2.5–5$\times$ more kernels to exceed speedup thresholds. SI reveals *which* failure mode to address next; without it, the proposer can only observe that the score changed.

6. Why the Framework Works: Optimization Trajectory Analysis

Beyond final scores, trajectory analysis on circle packing reveals three key mechanisms driving `optimize_anything`'s effectiveness: (1) **SI-enabled targeted algorithmic shifts** (e.g., collapsed radii $\rightarrow$ LP, poor centers $\rightarrow$ SLP) instead of blind mutations; (2) **multi-module Pareto leapfrogging**, where the code artifact and refiner prompt alternately advance on a shared front, creating a coordination dynamic absent from single-artifact systems; and (3) **Pareto diversity preventing premature convergence** by retaining candidates across algorithmic families (greedy, LP, SLP, bilevel L-BFGS, CMA-ES). These mechanisms operate identically across domains because they arise from the `evaluate(candidate)` $\rightarrow$ `(score, side_info)` contract; full details are in Appendix G.

7. Discussion

Cross-task transfer is most beneficial when problems share underlying optimization patterns but differ in specifics (e.g., CUDA kernels share memory-coalescing, vectorized access,

385 and warp-level reduction strategies that multi-task mode discovers once and transfers across operations); it can degrade performance when tasks lack shared structure, as on circle packing for different N where optimal configurations change unpredictably (single-task 2.6360 vs. MT11 2.5973; Table 5 in Appendix A.6) (Graham & Lubachevsky, 1996; Galiev & Lisafina, 2013). Extended discussion of how SI's role varies across domains and the qualitative diversity of the artifacts discovered by `optimize_anything` appears in Appendix B.

8. Limitations

`optimize_anything` inherits limitations from LLM-based optimization. (1) The quality of proposals depends on the proposer LLM's capabilities; weaker models produce weaker candidates, as confirmed by our proposer sensitivity analysis (Table 8). (2) Evaluation cost can be high when the evaluator involves expensive operations (e.g., $144 for ARC-AGI, Table 9), however, it must be noted that LLM-based optimization is highly sample efficient and therefore calls evaluators less often. (3) The system assumes the artifact is representable as text; optimization of continuous parameters or binary artifacts requires a text-based proxy. (4) While multi-task search provides cross-transfer benefits on related problems, the degree of benefit depends on how related the problems are, for example, circle packing exhibits degradation with multi-task mode (Table 5). (5) designing effective SI still requires domain expertise; while evaluators returning only a score work, the demonstrated gains come from expert-designed SI (compiler errors, profiler traces, VLM scoring rubrics). That said, `optimize_anything` trades *optimization* expertise for *domain* expertise. The user, most often a domain expert, need not configure backends, tune algorithmic hyperparameters, or engineer prompting strategies, only surface the diagnostics they already understand.

9. Conclusion

`optimize_anything` demonstrates that a simple declarative interface (seed artifact, evaluator, and optional dataset) is sufficient to match or outperform purpose-built tools across diverse domains. The key ideas are (1) three unified optimization modes under one API, (2) Side Information as a first-class evaluator contract, and (3) Pareto-based search across metrics and examples. The API is backend-agnostic; as new optimization strategies emerge, they plug in without changing user code. `optimize_anything` will be open-sourced with multiple backends; the project URL is omitted here to preserve anonymity for review.

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.

Chen, A., Dohan, D., and So, D. EvoPrompting: Language models for code-level neural architecture search. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

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

Ellenberg, J. S., Fraser-Taliente, C. S., Harvey, T. R., Srivastava, K., and Sutherland, A. V. Generative modeling for mathematical discovery, 2025. URL https://arxiv.org/abs/2503.11061.

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.

Khattab, O., Singhvi, A., Maheshwari, P., Zhang, Z., Santhanam, K., Vardhamanan, S., Haq, S., Sharma, A., Joshi, T. T., Moazam, H., Miller, H., Zaharia, M., and Potts, C. Dspy: Compiling declarative language model calls into self-improving pipelines, 2023. URL https://arxiv.org/abs/2310.03714.

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.

Lehman, J., Gordon, J., Jain, S., Ndousse, K., Yeh, C., and Stanley, K. O. Evolution through large models, 2022. URL https://arxiv.org/abs/2206.08896.

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.

Ouyang, A., Guo, S., Arora, S., Zhang, A. L., Hu, W., Ré, C., and Mirhoseini, A. Kernelbench: Can llms write efficient gpu kernels?, 2025. URL https://arxiv.org/abs/2502.10517.

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.

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.

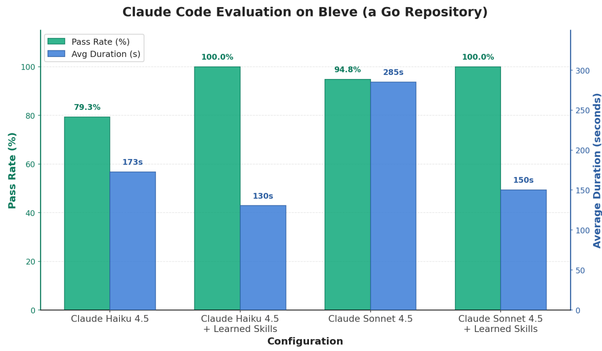
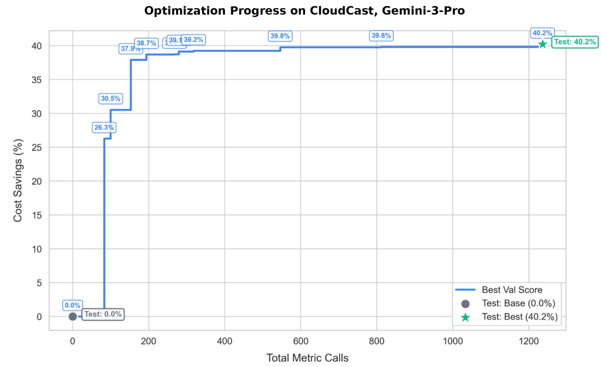
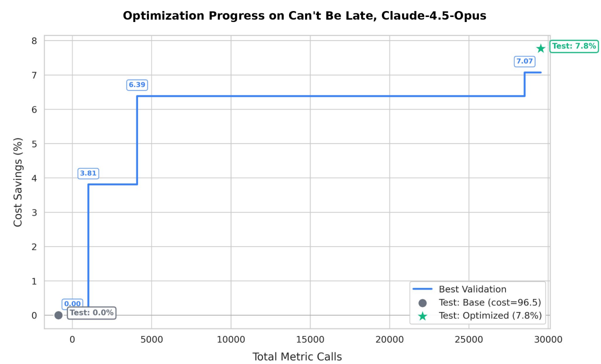


Figure 5. Claude Code on the Bleve repository. Optimized skills boost pass rates to near-perfect while reducing resolve time by 47%. Skills transfer across models without reoptimization.



(a) CloudCast: 40.2% cost savings.



(b) Can't Be Late: 7.8% savings.

Figure 6. Optimization trajectories for cloud scheduling. Both use generalization mode with train/val splits over infrastructure scenarios.

A. Additional Experimental Domains

This appendix presents three additional generalization-mode experiments referenced from Section 5: Coding Agent Skills, Cloud Scheduling Algorithms, and AIME Prompt Optimization. Section A.5 provides a complementary analysis of proposer sensitivity and optimization cost.

A.1. Coding Agent Skills (Generalization)

Setup. Skills are natural-language instructions and best practices for working with a specific codebase. The evaluator runs a coding agent on repository tasks and scores whether it resolves them; the optimized skills must generalize to unseen tasks. We optimize skills for the Bleve search library and evaluate transfer to Claude Code with both Haiku 4.5 and Sonnet 4.5.

SI design. The evaluator returns task descriptions, agent traces (tool calls, code edits, errors), test outcomes, and resolution time.

Results. Optimized skills boost Haiku 4.5's pass rate from 79.3% to 98.3% and Sonnet 4.5's from 94.8% to 100%, while cutting resolution time by 47% (Figure 5). Critically, skills discovered for one model transfer effectively to another without reoptimization, demonstrating the generalization mode's ability to learn model-agnostic repository knowledge.

A.2. Cloud Scheduling Algorithms (Generalization)

Setup. We optimize two cloud infrastructure algorithms from the ADRS benchmark. **CloudCast** discovers broadcast routing strategies for multi-cloud data transfer, minimizing data egress cost. **Can't Be Late** learns scheduling policies deciding when to use cheap preemptible `SPOT` instances versus reliable `ON_DEMAND` instances to meet deadlines. Both use generalization mode with training/validation splits over infrastructure scenarios.

SI design. For CloudCast: per-partition routing decisions, edge utilizations, cost breakdowns. For Can't Be Late: spot-availability patterns, instance-usage timelines, segment counts (`SPOT` vs. `ON_DEMAND` vs. restarts).

Results. CloudCast achieves 40.2% cost savings over Dijkstra routing (Figure 6a), evolving from a baseline shortest-path algorithm to a provider-aware Steiner tree approach that jointly optimizes for egress cost and transfer latency. Can't Be Late achieves 7.8% cost savings (Figure 6b), evolving a simple deadline-check heuristic into an adaptive strategy with state tracking for spot-unavailability patterns, break-even switching cost analysis, and graduated decision thresholds based on slack ratio. Both results top the ADRS leaderboard (`optimize_anything`: 96.6 aggregate score vs. 92.9 for OpenEvolve, 72.0 for ShinkaEvolve). The evolved artifacts are qualitatively different from their seeds: CloudCast discovers provider-aware Steiner tree routing (absent from the Dijkstra seed), while Can't Be Late learns persistent spot-unavailability tracking and overhead-aware switching costs (absent from the greedy seed).

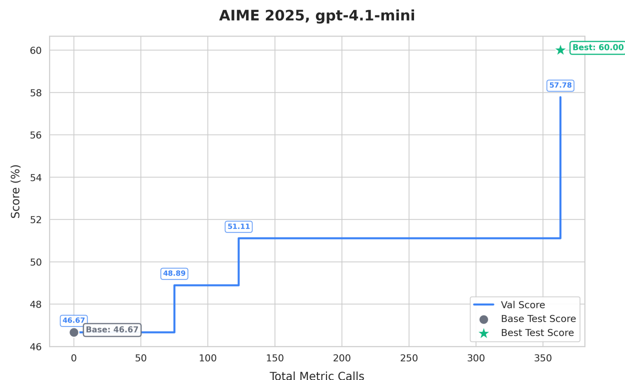


Figure 7. AIME prompt optimization for GPT-4.1-mini. Validation score improves from 46.67% to 57.78%; test score reaches 60.00%.

A.3. AIME Prompt Optimization (Generalization)

Setup. We optimize a system prompt for GPT-4.1-mini on AIME (American Invitational Mathematics Examination) competition problems. Training uses AIME 2022–2024; testing uses AIME 2025.

SI design. The evaluator returns each problem statement, the model's reasoning chain, extracted answer, ground truth, and a correct/incorrect flag.

Results. Prompt optimization improves GPT-4.1-mini from 46.67% to **60.00%** on AIME 2025 (Figure 7), a 13.3pp gain from changing only the system prompt. This outperforms MIPROv2 (Opsahl-Ong et al., 2024) (51.33% on the same benchmark). The optimized prompt (Appendix J) evolves from a single generic sentence into a structured 6-rule reasoning framework. This result matches the performance gains reported by Agrawal et al. (2026), demonstrating that exposing a prompt optimization algorithm through a general interface does not hurt performance on prompt optimization.

A.4. Image Generation (Multi-Task Search)

Setup. We generate SVG code and CAD models (via `build123d`) for four image goals (Table 10 in Appendix I). The evaluator renders the image and queries a VLM to rate individual visual aspects on a 0–100 scale; each evaluator call scores one aspect, making this a natural multi-task search over the Pareto frontier of visual properties.

Results. Five human evaluators unanimously preferred `optimize_anything`-optimized images over zero-shot baselines across all goals. Quantitatively, the "pelican riding a bicycle" task achieves a VLM score of 0.726 vs. 0.330 for the zero-shot baseline (2.2× improvement). Qualitative comparisons are shown in Appendix Figure 11.

A.5. Proposer Sensitivity and Optimization Cost

Comparing GPT-5.1 against the cheaper GPT-5-nano reveals a clear cost-performance tradeoff (Table 8 in Appendix H): the nano model reduces costs by over 90% on Circle Packing while still improving substantially over the seed, but consistently underperforms the larger model on final quality. Total optimization costs range from $1 (Numerical Blackbox) to $144.70 (ARC-AGI), with reflection cost minimal and total spend dominated by the evaluator (Table 9 in Appendix H).

A.6. Ablation Figures and Tables

Figure 8 below accompanies the multi-task vs. single-task ablation in §5.4. Figure 9 and Table 4 accompany the SI ablation in §5.5. Table 5 accompanies the discussion in §7 of when multi-task search hurts.

Table 4. SI vs. score-only ablation across three domains. SI provides substantial gains in all domains, confirming generalization beyond prompt optimization.

Domain	Metric	With SI	Score only
Circle Packing	Best score (% of best)	**100%**	93.96%
KernelBench (ST)	Kernels $\geq 1.1\times$ ($f_{1.1}$)	**32.3%**	12.9%
KernelBench (ST)	Mean speedup	**4.11×**	1.15×
KernelBench (MT)	Kernels $\geq 1.1\times$ ($f_{1.1}$)	**40%**	0%
KernelBench (MT)	Mean speedup	**1.15×**	1.03×

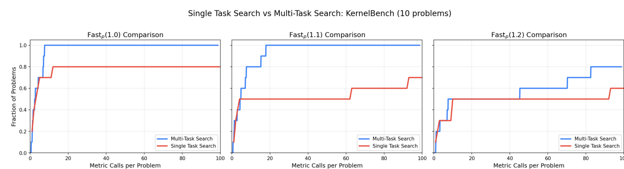


Figure 8. Single-task vs. multi-task mode on 10 selected KernelBench problems. Multi-task (blue) consistently outperforms single-task (red) at all speedup thresholds, converging faster and solving more problems.

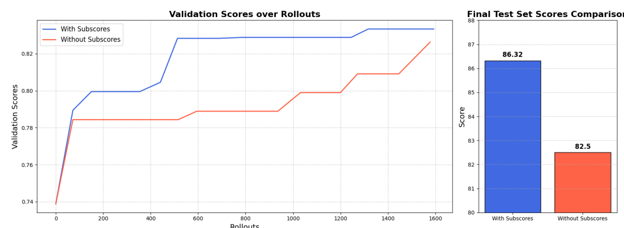


Figure 9. Ablation: prompt optimization with vs. without SI on the Facility Support Analysis dataset. SI accelerates convergence (left) and improves final test performance (right): 86.32 vs. 82.5.

B. Extended Discussion

These paragraphs accompany the Discussion in §7.

605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659

Table 5. Multi-task search on circle packing. Unlike CUDA kernels, circle packing problems for different N are fundamentally independent, and multi-task search introduces noise.

Config	Score
Single-task	**2.6360**
MT7	2.6313
MT11	2.5973

When does multi-task search help? Our experiments reveal that cross-task transfer is most beneficial when problems share underlying optimization patterns but differ in their specifics. CUDA kernel generation exemplifies this: memory coalescing, vectorized access patterns, and warp-level reductions are strategies that apply across operations but manifest differently for each kernel. Multi-task mode discovers these patterns once and transfers them, while single-task mode must rediscover them independently for each problem (Tables 6–7).

When does multi-task search hurt? Multi-task search can degrade performance when tasks lack shared transferable structure. We quantify this on circle packing, where optimizing different values of N jointly introduces noise rather than useful cross-transfer (Table 5 in Appendix A.6: single-task 2.6360 vs. MT7 2.6313 vs. MT11 2.5973). Circle packing problems for different N are fundamentally independent, optimal configurations change unpredictably with N, with no transferable structure (Graham & Lubachevsky, 1996; Galiev & Lisafina, 2013). In general, multi-task search helps when tasks share underlying patterns (e.g., CUDA kernels on the same hardware) and hurts when they are fundamentally independent.

The role of SI across domains. While the SI ablation (Table 4) confirms SI's value, the mechanism differs by domain: for code (CUDA, circle packing), SI surfaces compiler errors and runtime diagnostics pinpointing failures; for agents (ARC-AGI), per-puzzle traces reveal which components fail; for cloud scheduling, SI exposes temporal decision structure. In each case, SI converts a scalar signal into actionable diagnostics.

Artifacts optimized by `optimize_anything`. The optimized artifacts range from structured prompts (AIME) and agent architectures (ARC-AGI) to 900+ line bilevel algorithms (circle packing), demonstrating that the system discovers qualitatively novel strategies—multi-stage pipelines (ARC-AGI), provider-aware Steiner trees (CloudCast), break-even cost analysis (Can't Be Late)—arising from the interaction between LLM reasoning and diagnostic feedback.

C. Blackbox Mathematical Optimization

We additionally evaluate `optimize_anything` in single-task search mode on blackbox mathematical optimization, using the 56-problem EvalSet benchmark (McCourt, 2016) against Optuna (Akiba et al., 2019). Rather than tuning parameters within a fixed algorithm, `optimize_anything` optimizes the solver code itself, discovering bespoke algorithms for each problem.

With a budget of 8,000 evaluations per problem, `optimize_anything` ties Optuna on 40 problems, wins 7, and loses 9. On 10 selected problems where Optuna struggles with lower budgets (2,000 evaluations), `optimize_anything` finds better solutions on 7 out of 10. The mechanism: Optuna's fixed TPE-CMA-ES pipeline fails in predictable, structural ways (e.g., TPE's per-dimension sampling converges to trap basins; CMA-ES assumes smooth unimodal landscapes). `optimize_anything` tailors the solver to each problem—discovering L-BFGS-B for boundary optima and multi-start search for deceptive traps.

D. Seedless Mode: 3D Unicorn

Every main experiment starts from a seed artifact. Seedless mode (`seed_candidate=None`) instead provides only a natural-language objective and lets the LLM bootstrap the first candidate. We demonstrate this on a 3D modeling task: generating a Python script (build123d + pyrender) that produces a 3D unicorn. The evaluator renders multi-view PNGs and asks a VLM to score them, passing images back as SI. Starting from no code, `optimize_anything` iteratively refines geometry, proportions, and anatomical detail, producing a recognizable 3D unicorn that improves substantially over the zero-shot baseline.

E. Detailed Algorithm

Algorithm 1 `optimize_anything`: Core optimization loop

Require: Artifact Φ_0, evaluator f, dataset $\mathcal{D}$, budget B
Require: Minibatch size b, Pareto set size n
1: Initialize candidates $\mathcal{P} \leftarrow [\Phi_0]$
2: Evaluate Φ_0 on $\mathcal{D}$; record per-example scores S
3: **while** budget B not exhausted **do**
4: $\quad k \leftarrow$ PARETOSELECT$(\mathcal{P}, S)$ {Select based on frontier}
5: $\quad \mathcal{M} \leftarrow$ minibatch of size b from $\mathcal{D}$
6: $\quad$ Execute Φ_k on $\mathcal{M}$; collect scores and SI
7: $\quad \Phi' \leftarrow$ REFLECT$(\Phi_k$, scores, SI) {LLM proposes fix}
8: $\quad$ **if** Φ' improves on $\mathcal{M}$ **then**
9: $\quad\quad$ Evaluate Φ' on full $\mathcal{D}$
10: $\quad\quad \mathcal{P} \leftarrow \mathcal{P} \cup \{\Phi'\}$
11: $\quad\quad$ Update S; prune dominated candidates
12: $\quad$ **end if**
13: **end while**
14: **return** $\Phi^* \in \mathcal{P}$ maximizing average score

Algorithm 1 presents the core loop. For single-task search, the "dataset" is a singleton and per-example tracking reduces to per-metric tracking. For multi-task search, each dataset element is an independent problem. For generalization, scores on $\mathcal{D}$ guide search while a held-out `valset` measures generalization. The PARETOSELECT subroutine follows GEPA (Agrawal et al., 2026): it identifies non-dominated candidates and samples proportionally to their frontier frequency.

F. Multi-Task Scaling Tables

Table 6. Multi-task scaling on 10 KernelBench problems. $f_{1.x}$: fraction of kernels achieving $\geq x\%$ speedup over PyTorch baseline.

Setting	$\mathbf{f_{1.0}}$	$\mathbf{f_{1.1}}$	$\mathbf{f_{1.2}}$
ST	60.0%	40.0%	**20.0%**
MT10	**90.0%**	40.0%	**20.0%**
MT20	**90.0%**	**50.0%**	**20.0%**

Table 7. Single-task vs. MT20 on 20 randomly sampled KernelBench problems.

Setting	$\mathbf{f_{1.0}}$	$\mathbf{f_{1.1}}$	$\mathbf{f_{1.2}}$
ST	50.0%	25.0%	**15.0%**
MT20	**90.0%**	**40.0%**	**15.0%**

G. Optimization Trajectory Analysis: Full Details

Mechanism 1: SI enables targeted algorithmic shifts. SI works because it reveals *which* failure mode to address next, not merely that performance changed. 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 can only observe that the score changed, not why, and resorts to undirected mutations. The cross-domain SI ablation (Table 4) confirms this mechanism generalizes: on KernelBench with multi-task search, SI enables 40% of kernels to exceed $1.1\times$ speedup vs. 0% with score-only feedback.

Mechanism 2: Multi-module Pareto leapfrogging. `optimize_anything` optimizes both the code artifact and a refiner prompt, both tracked on the shared Pareto front. In circle packing, this creates a productive leapfrogging dynamic: the refiner discovers LP-based optimization while the code module is still a weak heuristic (code=0.98, refiner=1.93). The code module then absorbs the LP approach, catching up ($\rightarrow$2.61). The refiner pushes further with SLP ($\rightarrow$2.63). The code module absorbs SLP and reaches the world record. Each module's advances become the foundation for the other's next improvement—a coordination mechanism absent from single-artifact systems like AlphaEvolve. Even broken code mutations (score=0.0) are recovered by the refiner and retained on the front, acting as a safety net that preserves exploration.

Mechanism 3: Pareto diversity prevents premature convergence. At convergence, the Pareto front retains candidates from multiple algorithmic families simultaneously (greedy, LP, SLP, bilevel L-BFGS, CMA-ES) across quality dimensions (max score, mean score, EMA stability, improvement rate). This ensures the proposer has access to structurally diverse parents when generating new candidates, rather than being locked into refining a single approach. The preservation of diverse strategies is what enables the algorithmic shifts described above: even when LP dominates on raw score, greedy and CMA-ES candidates survive on stability metrics and can seed novel hybrid approaches.

H. Proposer Sensitivity and Optimization Cost

Table 8. Proposer LLM sensitivity. GPT-5-nano reduces cost significantly but underperforms GPT-5.1 on final achieved performance. Both models improve substantially over the seed.

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

I. Image Generation Details

For SVG tasks, the evaluator renders the image and queries a VLM for feedback. For each goal, we define several

Table 9. Total optimization cost per experiment. Reflection cost is minimal; total spend is dominated by the evaluator.

Task	Total Cost
Numerical Blackbox	$1
Circle Packing	$6
KernelBench (31 kernels)	$140
AIME (Generalization)	$6.44
SVG Optimization	$18
Agent Skills (Generalization)	$50
Cloud Scheduling	$52.42
ARC-AGI (Generalization)	$144.70

Goal	# Aspects	Repr.	Model
A pelican riding a bicycle	12	SVG	Gemini Flash 3.
A high-quality 3D unicorn	4	CAD	Claude Opus 4.
An octopus on a pipe organ	13	SVG	Claude Opus 4.
A sloth steering an excavator	13	SVG	Claude Opus 4.

Table 10. Image generation goals. A VLM evaluator scores one visual aspect per call; multi-task search explores the Pareto frontier of visual properties.

natural language properties which ask a VLM to rate on a scale of 0 to 100 how well the image aligns with that aspect. During each evaluator call, the VLM rates one aspect (not all at once), making this a natural multi-task search over the Pareto frontier. In the CAD setting, since we are dealing with 3D objects, the evaluator takes 3 screenshots equidistant apart and asks the VLM to provide feedback using those images.

J. Optimized AIME Prompt

Optimized Prompt for AIME

```
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 (e.g., 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
and what assumptions define it.  - If you
invoke a known theorem (e.g., Ptolemy,
Power of a Point, similarity, Vieta), name
it and show exactly how it applies in this
context.

4.  Handle dead ends correctly.  - If you
realize a line of reasoning leads to a
contradiction or dead end, explicitly say
so.  - Then restart from the last correct
point; do not guess or hand-wave.

5.  Keep the reasoning focused and minimal
while still being rigorous.  - Avoid
unnecessary numerical approximations if
an exact approach is available.  - Do not
approximate exact values unless the problem
explicitly asks for a decimal.  - Prefer
algebraic or structural arguments over
trial-and-error or random guessing.  -
You may test candidate values only after
deriving strong constraints that sharply
limit the possibilities.

6.  At the end, clearly isolate the answer:
- Provide the final answer as a single
number or expression on its own line.  -
Do not include any extra words, symbols, or
explanation on that final line.
```

K. Discovered solutions

We present excerpts of the final optimized artifacts discovered by `optimize_anything` for each domain.

K.1. Coding Agent Skills: Bleve Repository

The following is the optimized `SKILL.MD` excerpt discovered by `optimize_anything` for the Bleve search library:

Optimized Bleve Skills (excerpt)

```
4) Run tests early and iterate from
failures (tests are the bug report) - Start
broad when feasible:  'cd /testbed && go
test ./...' (or project equivalent).  -
Narrow quickly:  - package:  'go test
./path/to/pkg' - single test:  'go test
./path/to/pkg -run TestName -count=1' (add
-v only if needed) - For panics:  follow
the stack trace top frame in repo code
first.  - For mismatches:  use \expected vs
got" to locate the producing function and
invariants.

...

7) Make minimal, reviewable changes and
verify continuously - Change one behavior
at a time; rerun the smallest reproducing
test after each change.  - Add focused
unit tests when coverage is missing; keep
them in the same package and table-driven
where sensible (include short words +
accented/Unicode edge cases).  - Avoid
scratch main.go files in repo root.
```

K.2. ARC-AGI Agent Architecture

The optimized agent grew from a 10-line seed to a 300+ line system implementing a 4-stage pipeline: rule induction via pattern analysis, code generation with `exec()`-based verification, iterative debugging with up to 2 fix attempts, and structured fallback from code-first to direct LLM prediction.

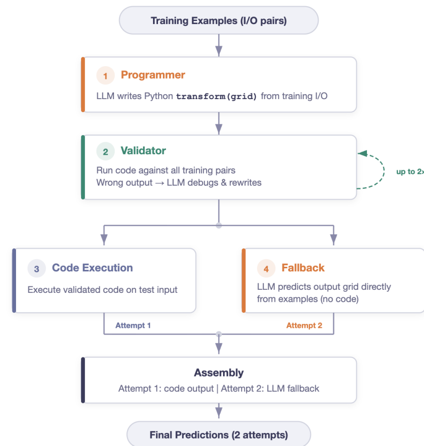


Figure 10. Architecture of the optimized ARC-AGI agent. The system discovers a 4-stage pipeline with verify-then-fallback logic, starting from a naive single-call seed.

K.3. CloudCast Routing Algorithm

The optimized CloudCast algorithm (178 lines) discovers provider-aware Steiner tree routing with egress cost optimization, a qualitative departure from the Dijkstra seed.

We show the main `search_algorithm` function; the full artifact is available in the supplementary material.

Optimized CloudCast Algorithm (excerpt)

```
def search_algorithm(src, dsts, G, num_partitions):
    """Optimized Broadcast Routing Algorithm v3.
    Key Optimizations:
    1. Provider-Aware Weighting: biases path finding
        towards
      intra-provider links to minimize egress.
    2. Pareto-Frontier Candidate Selection: Explicitly
        keeps
      candidates that offer distinct cost/time
          tradeoffs.
    3. Diverse Steiner Strategies: Includes MST-like
      approximations for cost and bottleneck-widest
      paths for throughput.
    4. Robust Greedy Allocation: Accurately models
        bandwidth
      contention across partitions.
    """
    # --- Constants & Configuration ---
    EST_DATA_VOL_GB = 300.0
    EST_INSTANCE_COST_PER_HR = 10.0
    PARTITION_VOL_GB = EST_DATA_VOL_GB / max(1,
        num_partitions)
    # Sweep parameters for Cost vs Time tradeoff
    alphas = [0.0, 1e-5, 0.001, 0.01, 0.05, 0.1, 0.5,
        2.0]
    bw_thresholds = [0.0, 0.5, 5.0, 20.0]
    strategies = ['prim', 'prim', 'furthest', 'random']
    # ... (remaining 140 lines: provider extraction,
        graph
    # preprocessing, Steiner tree construction, greedy
    # allocation, and Pareto-frontier selection)
```

K.4. Can't Be Late Scheduling Policy

The optimized scheduling policy (110 lines) starts from a simple deadline-check heuristic and discovers three key behaviors absent from the seed: (1) break-even switching cost analysis that avoids costly SPOT→ON_DEMAND transitions when remaining work is small, (2) persistent spot-unavailability tracking via a counter that detects when SPOT is unlikely to return, and (3) graduated decision thresholds based on slack ratio that become increasingly aggressive as the deadline approaches. We show the core `_step` method; the full artifact includes `reset()` and additional edge-case guards.

Optimized Can't Be Late Policy (excerpt)

```
from sky_spot.strategies.strategy import Strategy
from sky_spot.utils import ClusterType

class EvolveSingleRegionStrategy(Strategy):
   def __init__(self, args):
      super().__init__(args)
      self.spot_unavailable_count = 0
      self.consecutive_short_spot_windows = 0

   def _step(self, last_cluster_type, has_spot) ->
       ClusterType:
      remaining_task_time = self.task_duration - sum(
          self.task_done_time)
      remaining_time = self.deadline - self.env.
          elapsed_seconds
      slack = remaining_time - remaining_task_time -
          self.restart_overhead

      # Track persistent spot unavailability
      if not has_spot:
         self.spot_unavailable_count += 1
      else:
         self.spot_unavailable_count = 0

      # Critical deadline: must use ON_DEMAND
      if remaining_task_time + self.restart_overhead
          >= remaining_time - 0.5:
         return ClusterType.ON_DEMAND

      slack_ratio = slack / max(remaining_task_time, 1
          e-6)

      if has_spot:
         if last_cluster_type == ClusterType.ON_DEMAND:

            # Break-even analysis: is switching to SPOT
                worth it?
            switch_cost = self.restart_overhead * 1.0
            savings_per_hour = 0.7 # OD(1.0) - SPOT
                (0.3)
            break_even = switch_cost / savings_per_hour

            if remaining_task_time < break_even * 1.5:
               return ClusterType.ON_DEMAND
            if slack < self.restart_overhead * 3:
               return ClusterType.ON_DEMAND
         return ClusterType.SPOT
      else:
         if last_cluster_type == ClusterType.ON_DEMAND:
```

```
            return ClusterType.ON_DEMAND
        # Graduated thresholds based on slack ratio
        if slack_ratio < 0.1:
            return ClusterType.ON_DEMAND
        if slack_ratio < 0.25 and self.
            spot_unavailable_count > 10:
            return ClusterType.ON_DEMAND
        if slack_ratio < 0.4 and self.
            spot_unavailable_count > 20:
            return ClusterType.ON_DEMAND
        return ClusterType.NONE # Wait for spot
    # ... (remaining 15 lines: reset(), _from_args(),
    # and additional small-task / tight-deadline guards)
```

K.5. CUDA Kernel: LayerNorm

We show the best individual kernel discovered for LayerNorm, which achieves a 3.32× speedup over the PyTorch baseline. The kernel employs three key techniques absent from the naive implementation: (1) `float4` vectorization that loads four values per memory transaction, cutting memory overhead by ∼4×; (2) a two-pass algorithm (compute statistics, then normalize) that lets the GPU optimize each phase independently; and (3) warp shuffle reductions (`__shfl_down_sync`) for direct register-to-register partial sum accumulation, bypassing slower shared memory paths. This kernel was discovered in multi-task mode, where optimization patterns transfer across the 31 KernelBench problems via the shared Pareto frontier.

Optimized LayerNorm CUDA Kernel (excerpt)

```
__inline__ __device__ float warp_sum(float v) {
    unsigned mask = 0xffffffffu;
    for (int offset = KB_WARP_SIZE / 2; offset > 0;
        offset >>= 1)
        v += __shfl_down_sync(mask, v, offset);
    return v;
}

__global__ void rowwise_stats_kernel(
    const float* __restrict__ x, float* __restrict__
        mean,
    float* __restrict__ inv_std, int64_t B, int64_t M,
        float eps) {
    int64_t row = blockIdx.x;
    if (row >= B) return;
    const float* row_ptr = x + row * M;

    float thread_sum = 0.0f, thread_sumsq = 0.0f;
    // float4 vectorized loads when aligned
    const float4* row_v4 = reinterpret_cast<const float4
        *>(row_ptr);
    for (int64_t j = threadIdx.x; j < (M >> 2); j +=
        blockDim.x) {
        float4 v = row_v4[j];
        thread_sum += (v.x + v.y + v.z + v.w);
        thread_sumsq += (v.x*v.x + v.y*v.y + v.z*v.z + v.
            w*v.w);
    }
    // Warp shuffle reduction + cross-warp shared memory
        reduce
    thread_sum = warp_sum(thread_sum);
    thread_sumsq = warp_sum(thread_sumsq);
    // ... (shared memory cross-warp reduction, mean/
        inv_std output)
}

__global__ void layernorm_affine_kernel(
    const float* __restrict__ x, const float*
        __restrict__ weight,
    const float* __restrict__ bias, const float*
        __restrict__ mean,
    const float* __restrict__ inv_std, float*
        __restrict__ y,
    int64_t B, int64_t M) {
    int64_t row = blockIdx.x;
    float m = mean[row], inv = inv_std[row];
    // Vectorized normalize + affine transform
    const float4* x_v4 = reinterpret_cast<const float4
        *>(x + row*M);
    float4* y_v4 = reinterpret_cast<float4*>(y + row*M);

    for (int64_t j = threadIdx.x; j < (M >> 2); j +=
        blockDim.x) {
        float4 xv = x_v4[j], wv = w_v4[j], bv = b_v4[j];
        y_v4[j] = {((xv.x-m)*inv)*wv.x+bv.x, ((xv.y-m)*
            inv)*wv.y+bv.y,
                ((xv.z-m)*inv)*wv.z+bv.z, ((xv.w-m)*inv)
                    *wv.w+bv.w};
    }
}
// ... (remaining 80 lines: host function, input
    validation,
// thread/block configuration, Python module wrapper)
```

K.6. Circle Packing Algorithm

The evolved circle packing algorithm (480+ lines) is a bilevel optimizer that jointly optimizes circle centers and radii for n=26 circles in a unit square. Starting from a simple greedy packing seed, the system discovers a multi-stage architecture: (1) an LP over radii with dual-variable sensitivities that provide exact gradients for center optimization, (2) L-BFGS-B over centers using these LP-derived gradients, (3) block SLP trust-region boosts targeting the worst-performing circles, (4) CMA-ES global exploration with automatic restarts, and (5) aggressive relocation of smallest circles to edges and corners. The algorithm also employs six diverse seeding strategies (hexagonal, uniform, edge-ring, farthest-point, corner-spokes, and edge-biased hex) to avoid local optima. We show the main entry point and key optimization components.

Evolved Circle Packing Algorithm (excerpt)

```
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 constraints: r_i <= min(x_i, 1-x_i,
            y_i, 1-y_i)
        # Pairwise constraints: 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: trust-region moves on worst circles
def block_slp_boost(centers, rounds=4, k=10, delta
     =0.18):
    # Jointly optimize dx, dy, r via linearized LP
    # Focus on smallest radii + highest gradient
         circles

# CMA-ES global exploration over center positions
# 6 seeding strategies: hex, uniform, edge-ring,
# farthest-point, corner-spokes, edge-biased hex

# Pipeline: seed evaluation -> L-BFGS bilevel ->
# block SLP boost -> CMA-ES -> relocate worst ->
# evolutionary exploration -> final SLP polish
# ... (remaining 350 lines: seeding, CMA-ES,
     relocation,
# evolutionary exploration, final polish)
```

K.7. Artifact Availability

`optimize_anything` will be open-sourced; specific repository URLs are withheld here to preserve anonymity for review. A reproduction artifact accompanying this paper will be released alongside the camera-ready version. Each evaluation domain has its own subdirectory under `domains/` with runnable `optimize_anything` code, a `README.md` mapping the folder to the relevant section of this paper, and the saved optimizer-state checkpoint from the paper run. See the top-level `README.md` for the reproduction guide.

Hardware notes. Most domains run on a single CPU host with API access to the proposer and refiner LLMs (the paper used GPT-5/5.1, Gemini 3 Flash, and Claude Opus 4.6 depending on domain; exact identifiers are documented per domain). The KernelBench domain requires an NVIDIA V100 32GB GPU with CUDA 12.1+.

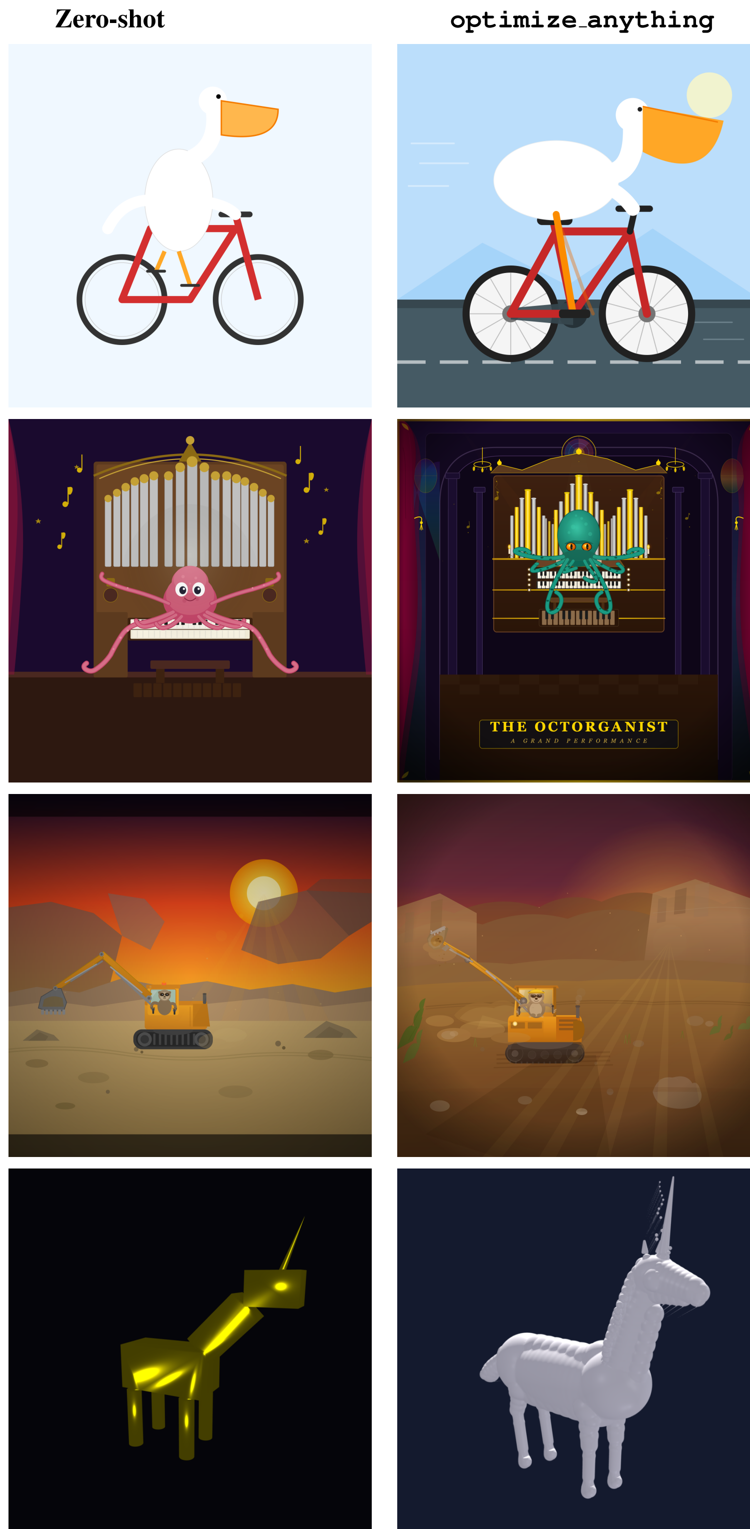


Figure 11. Qualitative comparison between zero-shot generations (left) and optimize_anything candidates (right) across four example tasks. Optimization consistently improves many visual aspects including composition, structure, detail, and overall visual quality.