Agentic Optimization: A Framework for Autonomous Coding-Agent Search

Junye Pan
junyep2@illinois.edu

Abstract

Large language models are increasingly capable at code understanding, program synthesis, and debugging, but many automated discovery systems still place them inside rigid controller-owned search loops. We present Agentic Optimization, a server-first framework that moves search policy into autonomous coding agents while keeping task definition, evaluator access, candidate snapshotting, and policy enforcement under a server-owned control plane. The framework exposes task-agnostic `verify` and `submit` interfaces, plus semantic access to incumbents, artifacts, findings, traces, and shared tools, without prescribing a fixed evolutionary workflow. We evaluate Agentic Optimization on circle packing and LLM inference kernel optimization. On CP26, both linear and multi-agent runs exceed previously reported scores, reaching 2.63598356975 and 2.63598366253. On end-to-end Qwen3-1.7B SGLang serving optimization, agents produce locally plausible changes but fail to obtain reliable gains, revealing a mismatch between public verification, local microbenchmarks, and hidden serving-level hard gates. When narrowed to a decode-attention microkernel on the same RTX 4090 target, the best candidate achieves a 2.063× operator-level speedup over the official SGLang implementation.

Code availability. `https://github.com/Junye-Pan/AgenticOptimization`.

1 Introduction

Executable feedback has become a central mechanism for turning language-model proposals into reliable scientific and algorithmic progress. AlphaTensor demonstrated that search procedures coupled to exact evaluators can discover nontrivial algorithms for matrix multiplication [1]. FunSearch showed that an LLM paired with a systematic evaluator and program archive can improve mathematical constructions [2]. More recent systems broaden this pattern from specialized generators to coding agents and automated research loops: AlphaEvolve evolves code under evaluator feedback [3]; ShinkaEvolve improves the sample efficiency of LLM-driven program evolution [4]; AVO treats autonomous coding agents as variation operators [5]; CORAL studies autonomous multi-agent evolution with persistent memory [6]; and Evaluation-driven Scaling argues that evaluator-mediated loops are a primary scaling axis for scientific discovery [7].

This trajectory raises a systems question. If a modern coding agent can inspect code, run commands, compare traces, write analysis scripts, construct local tools, and decide which artifact to revise, how much of the optimization policy should remain hard-coded in the outer loop? Existing systems often retain a strong controller: the controller selects parents, samples prompts, chooses mutation or crossover operators, manages archives, decides which context enters the model input, and schedules official evaluation. Such scaffolding is useful, but it also makes it difficult to observe whether the agent itself can learn to conduct search. A fixed outer loop may suppress precisely the

behaviors that coding agents are beginning to exhibit: selective retrieval, opportunistic debugging, local experiment design, tool creation, and cross-session knowledge reuse.

We propose Agentic Optimization, a framework for giving coding agents direct responsibility for inner-loop search while keeping measurement and reproducibility under server control. In Agentic Optimization, the outer loop does not implement a fixed evolutionary algorithm. It creates experiments and assignments, starts disposable worker sessions, records events, enforces policy, snapshots artifacts, and owns official evaluation. The worker agent decides which prior artifacts to inspect, whether to exploit an incumbent or explore a new direction, what local checks to run, when to publish findings or tools, and when a candidate merits authoritative scoring.

This paper makes three contributions.

1. We formulate *agentic optimization* as a separation between agent-owned search behavior and server-owned experimental authority. This separation makes the coding agent the inner-loop optimizer without giving it direct control over benchmark assets, official scoring, budgets, or provenance records.

2. We introduce a server-first architecture that represents experiments through semantic resources—tasks, assignments, sessions, attempts, artifacts, evaluations, incumbents, findings, traces, environments, and shared tools—rather than through ad hoc workspace directories or growing chat transcripts.

3. We define a task-agnostic evaluation interface that separates hard-constraint verification and authoritative submission. This interface gives agents useful feedback while preserving hidden evaluators, benchmark integrity, and reproducible candidate snapshots.

The intended contribution is therefore not a particular optimizer, prompt, or mutation operator. It is a systems substrate for measuring and amplifying the search behavior of coding agents while retaining the authority boundaries required for scientific claims.

2 Related Work

Evaluation-guided discovery. A common pattern in recent automated discovery systems is to search over executable artifacts rather than over natural-language answers. AlphaTensor casts matrix multiplication discovery as a reinforcement-learning game over tensor decompositions [1]. FunSearch searches over programs with a systematic evaluator and maintains an archive of high-performing code [2]. Eureka applies LLM-generated code to reward design, using execution feedback to improve reward functions [8]. Voyager shows that executable skill libraries can support long-horizon agent learning in an open-ended environment [9]. These works motivate the premise that discovery systems should treat executable feedback and reusable artifacts as first-class objects.

Program evolution with language models. AlphaEvolve, ShinkaEvolve, CodeEvolve, AVO, CausalEvolve, CORAL, and Evaluation-driven Scaling all study variants of evaluator-guided program improvement with LLMs or coding agents [3–7, 10, 11]. Their controllers differ in archive design, parent selection, novelty filtering, multi-agent scheduling, memory, and selection strategy. Agentic Optimization shares their emphasis on executable feedback, but it changes the systems boundary: rather than improving the controller's mutation heuristic, it minimizes controller-side search policy and exposes durable state through semantic tools. The resulting question is complementary: how should a platform be organized when the coding agent, not the controller, is expected to decide the local search procedure?

Coding-agent interfaces and harnesses. The performance of an LLM agent depends strongly on the interface and harness through which it acts. SWE-agent demonstrates that an agent-computer interface can substantially affect software-engineering performance [12]. AutoHarness studies automatic synthesis of code harnesses that prevent invalid actions [13]. Meta-Harness searches over model-harness code using prior source, scores, and traces as optimization context [14]. These works motivate Agentic Optimization's focus on the worker-facing interface: the agent should receive capabilities that are semantically meaningful, auditable, and policy-gated, rather than an unstructured pile of files or a rigid workflow script.

Automated research systems. The AI Scientist, AutoSOTA, AIDE, AutoKaggle, and MLE-bench study broader automated research or machine-learning engineering loops [15–19]. These systems emphasize paper-to-code grounding, experiment execution, data-science workflows, replication, or benchmarked ML engineering. Agentic Optimization targets a narrower but more controlled substrate: task packages expose candidate contracts and evaluator boundaries, while the control plane records the provenance required to interpret optimization trajectories. This narrower scope is intentional; it avoids conflating agentic search behavior with paper-writing, idea generation, dataset procurement, or uncontrolled environment repair.

3 Problem Setting and Design Principles

3.1 Autonomous optimization tasks

We consider optimization tasks in which progress is expressed through executable artifacts. A task is specified by

$$\tau = (\mathcal{C}, \mathcal{X}_{\text{pub}}, \mathcal{X}_{\text{hid}}, V, F),$$

where $\mathcal{C}$ is the candidate space, $\mathcal{X}_{\text{pub}}$ is public task context, $\mathcal{X}_{\text{hid}}$ denotes hidden benchmark assets, V is a hard verifier, and F is the authoritative scoring function. A candidate $c \in \mathcal{C}$ may be a source file, a directory of code, a configuration, or any executable artifact satisfying the task's public contract.

The worker can observe public context and server-exposed optimization history, but it does not observe hidden benchmark assets or official evaluator internals. The official score is defined only through server-mediated submission:

$$s(c) = F(\sigma(c); \mathcal{X}_{\text{hid}}),$$

where $\sigma(c)$ is an immutable snapshot of the candidate at submission time. Verification provide search feedback.

$$V(c) \to \{\text{valid}, \text{invalid}\}$$

This separation lets the agent use executable feedback during search while keeping the measured result tied to a snapshotted artifact and hidden official evaluator.

3.2 From controller-owned search to agent-owned search

Many evaluator-guided discovery systems can be viewed as a controller-owned search loop. At step t, the outer algorithm selects prior candidates, constructs a model context, specifies a variation operator, and asks the model to generate a new artifact:

$$c_{t+1} = \text{Generate}_\theta(\text{Mutate}_\psi(\text{Select}_\psi(\mathcal{P}_t))),$$

where $\mathcal{P}_t = \{(c_i, s_i)\}_{i=1}^{t}$ is the accumulated archive, θ denotes the model, and ψ denotes the hand-coded outer-loop policy. In this formulation, the model participates in candidate generation, but the search policy is largely outside the agent.

Agentic Optimization instead treats the coding agent as the inner-loop optimizer:

$$c_{t+1} = \text{Agent}_\theta(\mathcal{P}_t, \mathcal{K}_t, \mathcal{T}, V, P),$$

where $\mathcal{K}_t$ is server-exposed semantic memory, including artifacts, evaluations, incumbents, findings, traces, shared tools, and task knowledge, and $\mathcal{T}$ is the agent's tool interface. The agent decides which artifacts to inspect, whether to exploit an incumbent or explore a new direction, what local diagnostics to run, what tools to build, and when to submit a candidate for official scoring.

The outer loop remains necessary, but its role changes. It no longer defines a fixed evolutionary workflow such as parent selection, mutation, reflection, and replacement. Instead, it provides the task, starts worker sessions, allocates budgets, exposes semantic state, snapshots submitted candidates, runs official evaluation, and records provenance. Thus search behavior is agent-owned, while experimental authority is server-owned.

3.3 Design principles

This formulation leads to three design principles.

Agent-owned search. The coding agent should not merely fill in a mutation template chosen by a controller. It should be able to retrieve prior artifacts, inspect evaluation history, run local experiments, create helper tools, publish findings, and decide when a candidate is worth official evaluation.

Server-owned authority. The agent may request evaluation, but it should not control hidden benchmark assets, official scoring code, candidate snapshots, budget counters, leaderboard updates, or incumbent records. These authority-bearing operations belong to the server.

Semantic state instead of transcript state. Long-horizon optimization should not depend on a single growing conversation or on ad hoc workspace directories. The server should expose durable semantic resources—artifacts, evaluations, incumbents, findings, traces, task knowledge, and shared tools—that future agents can retrieve and build on.

4 Agentic Optimization Framework

4.1 Architectural overview

Agentic Optimization has three active roles: a task package, a control plane, and disposable worker sessions. The task package defines the public contract and private evaluation boundary. The control plane is the persistent authority for semantic state. A worker session is a bounded execution of a CLI coding agent inside a managed workspace.

A typical run proceeds as follows. The server creates an experiment for a task, materializes a workspace from public task context and selected server records, starts a coding-agent session, receives semantic tool calls and candidate submissions, records evaluations, updates the leaderboard and incumbent, and optionally starts a fresh session over the same durable state. The repeated session structure is important: continuity comes from server-visible resources rather than from an indefinitely growing conversation.

Table 1: Primary framework boundaries. The controller provides authority-bearing infrastructure but does not prescribe the worker's search algorithm.

Boundary	**Responsibility**
Task package	Public instructions, candidate contract, runtime requirements, public seed artifacts, verification, and authoritative evaluation.
Control plane	Experiments, assignments, sessions, attempts, artifacts, evaluations, leaderboards, incumbents, environments, findings, traces, shared tools, telemetry.
Worker session	Local candidate edits, local scratch execution, context inspection, use of semantic tools, publication of findings and shared tools, and decisions about when to submit.
Artifact store	Durable bytes for candidates, traces, replay bundles, shared tools, logs, and exported state; semantic meaning is assigned by control-plane records.
Runtime provider	Immutable base environments and policy-controlled overlays for verification and official evaluation.

Table 2: Evaluation surfaces exposed to workers.

Surface	**Role**
`verify`	Checks hard constraints: candidate shape, dependency sanity, compilation, public interface compatibility, static restrictions, and cheap correctness tests.
`submit`	Snapshots the candidate, evaluates it with the authoritative scorer, records the result, and, for valid improvements, updates leaderboard and incumbent state.

The boundary in Table 1 is the central methodological claim. The framework is deliberately "thin" only with respect to search policy. It is not thin with respect to measurement, provenance, or safety-critical control.

4.2 Task API

A task package exposes a public contract and private authority-bearing entry points. The public contract specifies the candidate layout, allowed files, input-output interface, public seed, public task knowledge, and any public diagnostics. The private task implementation supplies two entry points: `verify` and `submit`. This split addresses a recurring failure mode in open-ended optimization: if the only feedback is official scoring, agents waste budget on malformed candidates; if the evaluator is fully exposed, agents can overfit to private assets or exploit scoring artifacts. `verify` provide public-safe feedback, while `submit` remains the source of official claims. For path-based submissions, the server snapshots the submitted candidate before evaluation so that the scored object is immutable and replayable.

4.3 Server-first control plane

The control plane represents the experiment as typed resources rather than as a directory tree. Core resources include tasks, experiments, assignments, worker sessions, attempts, candidate artifacts, evaluations, leaderboard entries, incumbents, runtime environments, findings, traces, and shared tools. The database is the semantic source of truth: a file becomes a candidate, trace, replay bundle, or shared tool only when registered as such by the server. This design supports both auditability and agent autonomy. From the server side, every official claim can be linked to a task version, candidate artifact, runtime environment, evaluator request, result record, and trace context. From the worker side, the agent can retrieve the current incumbent, inspect relevant evaluations, search findings, check out shared tools, and compare prior artifacts without guessing paths or parsing arbitrary logs.

4.4 Semantic worker interface

The worker-facing command surface is the interface through which autonomy is bounded. It exposes semantic capabilities such as context retrieval, verification, official submission, finding publication, shared-tool publication, and trace inspection. The key design choice is that these commands are not a hidden workflow. The agent may call them in any order consistent with policy. For example, a worker may start from the incumbent, inspect failed attempts, write a local analyzer, publish the analyzer as a shared tool, then submit; another worker may ignore the incumbent and begin from a public seed. The controller records and constrains these actions but does not decide the local search path.

4.5 Memory across disposable sessions

A single ever-growing agent conversation is a weak substrate for scientific optimization: it is hard to audit, hard to reproduce, and eventually dominated by stale or irrelevant context. Agentic Optimization instead treats worker sessions as disposable and stores continuity in server-visible resources. Agent-created memory is split by function. Findings store durable claims, observations, failures, or hypotheses. Notebook checkpoints preserve local worklogs without forcing them into the prompt. Shared tools package reusable helper code. Traces record the raw interaction stream and command history for audit, but traces are not treated as conclusions; conclusions should be promoted into findings or notebooks. Attempts and artifacts capture candidate lineage and evaluated bytes. This separation reduces context bloat and gives future workers structured retrieval targets.

4.6 Runtime and policy control

Runtime environments are control-plane resources. A task declares a base runtime specification; the server resolves it into an immutable environment record. Worker-requested dependencies are represented as overlays and may be denied, approved for local diagnostics, or allowed for official evaluation only under explicit policy. Official scoring should prefer immutable environment identities such as container image digests over mutable host environments.

Network policy separates control-plane access from external internet access. Semantic tools require access to the control plane even when public internet is denied. Therefore, a backend that cannot distinguish these channels must mark the session as policy-weakened rather than silently treating prompt-level instructions as enforcement. This distinction is essential for benchmark integrity: preventing leakage is a systems property, not merely an instruction-following property.

4.7 Multi-agent exploration

The framework supports multi-agent breadth without hard-coding a multi-agent search algorithm. Assignments may carry task-defined directions, hypotheses, or exploitation targets. Workers in different assignments share durable state through the control plane but retain independent workspaces and local decisions. This enables controlled comparisons between exploration and exploitation: an exploiter converging on the incumbent is expected, while independent explorers collapsing onto the same method family may indicate insufficient diversity.

Because all official submissions, artifacts, findings, tools, and traces are typed records, multi-agent behavior can be analyzed after the run. Useful analyses include lineage graphs, artifact-diff clusters, direction-level score trajectories, finding reuse, shared-tool adoption, and entropy over active research directions. These analyses are external to the worker and therefore do not require the controller to impose a fixed search policy during the run.

5 Experimental Tasks

We evaluate Agentic Optimization on three independent task instances. Each task exposes a public verifier for repeated use during search and a separate evaluator for official scoring. The verifier is designed to reject malformed or incorrect candidates and to provide public-safe feedback. The evaluator is server-owned and defines the leaderboard score. This section describes the task definitions, evaluation design, and empirical results.

5.1 Circle Packing in a Unit Square

Task definition. Circle packing in a unit square asks for a configuration of non-overlapping circles that maximizes the total radius. In this task, the instance size is fixed to $n = 26$. A candidate construction outputs circle centers and radii.

Task: Circle Packing in a Unit Square.
Given $n = 26$, find circle centers

$$P = (p_1, \ldots, p_n), \qquad p_i = (x_i, y_i) \in [0, 1]^2,$$

and radii

$$r = (r_1, \ldots, r_n), \qquad r_i \geq 0,$$

such that every circle is contained in the unit square and no two circles overlap. Equivalently, for all $1 \leq i \leq n$,

$$r_i \leq x_i \leq 1 - r_i, \qquad r_i \leq y_i \leq 1 - r_i,$$

and for all $1 \leq i < j \leq n$,

$$r_i + r_j \leq \|p_i - p_j\|_2.$$

The objective is

$$\max_{P,r} \sum_{i=1}^{n} r_i.$$

This task is used as a compact mathematical optimization benchmark. The candidate surface is deliberately minimal: the observable output is a proposed geometric construction, not a proof or a natural-language answer. This makes the evaluation boundary unambiguous: feasibility is determined by geometric constraints, and quality is determined by the total safe radius.

Table 3: Circle packing results for $n = 26$. Higher is better. Prior scores are reported with the precision used in the corresponding papers; our entries are official strict-safe scores from the evaluator.

Method	**Score**
AlphaEvolve [3]	2.635862
ShinkaEvolve [4]	2.635982
TTT-Discover [20]	2.635983
Agentic Optimization (linear)	2.63598356975
Agentic Optimization (multi-agent)	**2.63598366253**

Verifier. The public verifier is a hard-constraint checker. It validates the output contract: the candidate must produce exactly 26 centers, 26 radii, and a finite declared objective value. It also checks that all coordinates and radii are finite, that all radii are non-negative, and that the declared objective is consistent with the returned radii.

The verifier then evaluates the geometric constraints. A construction passes verification only if every circle is contained in the unit square and every pair of circles is non-overlapping, up to the verifier's numerical tolerance.

Evaluator. The evaluator is server-owned and defines the official score. For a candidate that passes the verifier, the evaluator fixes the submitted centers P and recomputes the radii used for scoring. Thus the official score is determined by the quality of the submitted layout of centers, not by the candidate's self-reported radii.

This strict fixed-center scoring rule is the main difference between verification and official evaluation. The verifier answers whether the returned construction is acceptable under public tolerances. The evaluator recomputes a strictly safe set of radii and uses that value as the official score. This prevents candidates from gaining score by placing radii exactly on, or slightly beyond, the numerical tolerance boundary.

Results. We evaluate two usage patterns of Agentic Optimization on the $n = 26$ circle packing task for both linear run and multi-agent run. In the linear run, optimization proceeds as a single sequential trajectory. In the multi-agent run, multiple workers explore the same task through different directions while sharing state through the framework. Some workers search for strong basins from fresh starts; others focus on non-row layouts, topology changes, incumbent repair, or tolerance-aware polishing. The framework acts as the collaboration layer: workers inspect incumbents, compare evaluated artifacts, publish findings, and reuse shared tools rather than restarting from the initial program. This division of labor produces a small but measurable improvement over the linear run.

The runs also illustrate agent-owned search behavior. Workers created local search scripts, promoted reusable scripts into shared tools, checkpointed findings, compared incumbents, and used server-side probes to distinguish raw radius sums from strict-safe scores. This behavior was not encoded as a fixed outer-loop mutation rule. Instead, it emerged from the worker's access to semantic state: artifacts, incumbents, findings, evaluations, traces, and shared tools.

Table 3 reports the final scores. Both Agentic Optimization runs exceed the previously reported CP26 scores at the precision available in prior papers. The multi-agent run obtains the best score, improving over the linear run by 9.28×10^{-8}.

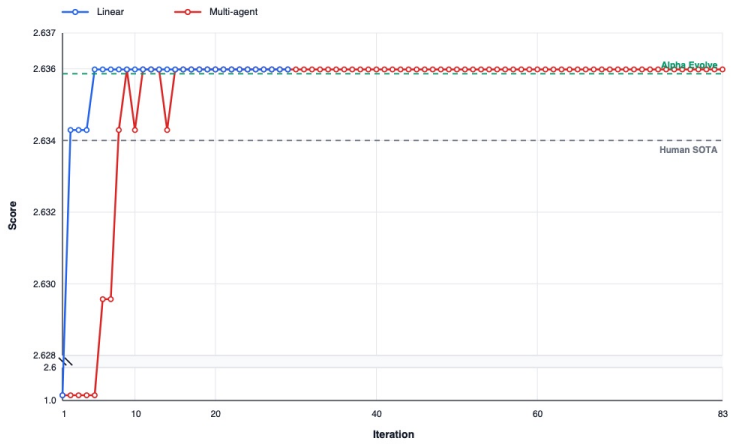

Figure 1: Score trajectory for the $n = 26$ circle-packing task. The lower portion of the vertical axis is compressed so that the early rise and the high-score optimization regime are visible in the same plot. Dashed lines mark the human state of the art reported before AlphaEvolve and the AlphaEvolve reference score [3].

5.2 End-to-End LLM Inference Kernel Optimization

Task definition. The end-to-end kernel optimization task studies whether a model serving run can be accelerated by replacing a fixed set of kernel and backend implementations while preserving the behavior of the original serving system. The model, tokenizer, request semantics, serving stack, scheduling policy, and benchmark workloads are fixed by the task. A valid candidate may only alter the exposed kernel/backend surface. It may not change the model architecture, model weights, tokenizer, request batching policy, or global serving logic.

The task includes both prefill and decode workloads. A prefill workload stresses prompt processing and KV-cache construction. A decode workload stresses autoregressive generation after a prefix has already been processed. All end-to-end serving experiments use Qwen3-1.7B in BF16 on a single NVIDIA RTX 4090. The baseline M_0 is the unmodified official SGLang implementation, and the candidate system M_c is evaluated under the same task-owned serving configuration on the same hardware. Therefore, the reported speedups are measured relative to official SGLang, rather than to a custom hand-optimized baseline.

Task: End-to-End Kernel Optimization for LLM Serving.
Let M_0 be the unmodified baseline serving system and M_c be the candidate-injected serving system. For each request workload w, the candidate must preserve the observable behavior of M_0 while reducing latency. Informally, the task is

$$\min_{c} \ \ell_c(w) \quad \text{subject to} \quad Y_c(w) \equiv Y_0(w) \quad \text{for all hard-gate workloads } w,$$

where $Y_c(w)$ denotes the candidate response and $\ell_c(w)$ denotes its measured serving latency.

Verifier. The public verifier has three parts.

First, it checks that the candidate remains inside the fixed optimization surface. This prevents candidates from modifying the model, tokenizer, scheduler, global serving modules, or other uncontrolled parts of the system.

Second, it performs component-level correctness checks. Deterministic kernels are compared against reference outputs using numerical tolerances. The verifier rejects invalid shapes, wrong dtypes, non-finite values, and outputs whose absolute or relative error exceeds the allowed tolerance.

Third, it runs a public server-level smoke test. The smoke test compares a candidate-injected server against an unmodified baseline server on public requests. For deterministic requests, the verifier compares generated text, output token ids, output token logprobs, and top-logprobs. It also records dispatch evidence to ensure that the declared kernel domains were actually exercised. This smoke test is not an official performance measurement; it is a public correctness and integration check.

TVD for stochastic sampling. Sampling cannot be verified by requiring a single sampled token to match the baseline. For stochastic sampling checks, the verifier compares distributions. Let q be the reference token distribution after applying the same sampling constraints, such as top-k or top-p filtering. Let $\widehat{q}_c$ be the empirical distribution obtained by repeatedly running the candidate sampler. The total variation distance is

$$\mathrm{TVD}(\widehat{q}_c, q) = \frac{1}{2} \sum_{v \in \mathcal{V}} \left| \widehat{q}_c(v) - q(v) \right|,$$

where $\mathcal{V}$ is the vocabulary. The stochastic component check accepts the candidate only if

$$\mathrm{TVD}(\widehat{q}_c, q) \leq \tau_{\mathrm{TVD}}.$$

Thus deterministic kernels are checked by numerical equivalence, while stochastic sampling kernels are checked by distributional equivalence.

Evaluator. The server-owned evaluator evaluates the candidate on hidden prefill and decode workloads. It launches two isolated serving systems: the unmodified baseline and the candidate-injected system. Both are evaluated under the same request set and launch configuration.

Correctness is a hard gate. For hard-gate workloads, the candidate must match the baseline response under the official equivalence checks: deterministic text, output token ids, output logprobs, top-logprobs, required dispatch counters, and absence of candidate exceptions or disallowed fallback behavior. Invalid candidates receive score 0.

For a valid candidate, the official score is the geometric mean of baseline-to-candidate latency ratios over the hard-gate hidden workloads:

$$S_{\mathrm{E2E}}(c) = \left(\prod_{w \in \mathcal{H}_{\mathrm{E2E}}} \frac{\ell_0(w)}{\ell_c(w)} \right)^{1/|\mathcal{H}_{\mathrm{E2E}}|},$$

where $\mathcal{H}_{\mathrm{E2E}}$ is the hidden hard-gate workload set, $\ell_0(w)$ is the baseline latency, and $\ell_c(w)$ is the candidate latency. Diagnostic timing workloads may be recorded for analysis, but they do not define the official score.

Results and failure analysis. In this end-to-end serving task, Agentic Optimization did not obtain a reliable positive result. The agents were able to produce locally plausible kernel and backend changes, and many candidates passed the public verifier. However, these candidates often failed the official hard-gate evaluation or reduced the final serving score. This makes the task a useful negative result: it exposes a gap between local kernel optimization and reliable end-to-end LLM serving optimization.

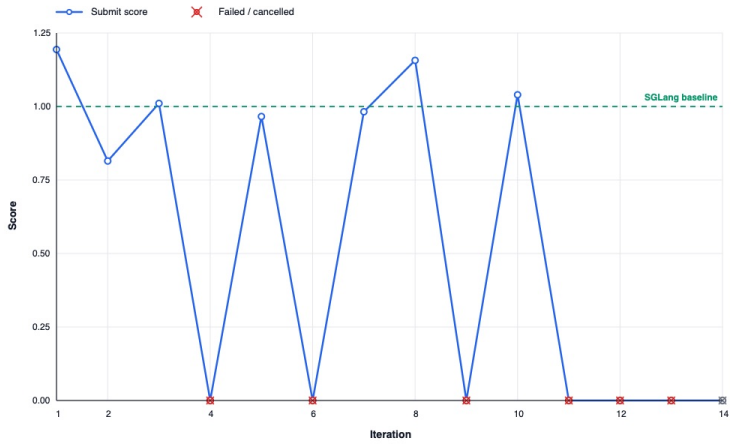

Figure 2: Official submit score progression for the end-to-end Qwen3-1.7B SGLang optimization task. The dashed line marks the unmodified SGLang baseline. Valid submissions are shown by the blue curve, while failed or cancelled leaderboard attempts are plotted at zero with cross markers. The best valid submission reaches 1.19×, and the repeated zero-score outcomes illustrate the brittleness of full serving-stack optimization.

The main failure was the mismatch between public feedback and official end-to-end scoring. Several candidates passed public verification but received an official score of 0.0, because hidden hard-gate workloads exposed text, token-id, logprob, or top-logprob mismatches. Local microbenchmarks were also weak predictors of serving performance: changes that improved individual kernels often failed full-server correctness or reduced the final geometric-mean score once batching, sampling, and logprob computation were included.

The best candidate improved prefill latency but degraded decode, showing that the official objective cannot be optimized by making a single local component faster. We also observed sensitivity in official evaluation: a semantically unchanged incumbent resubmission, differing only by trailing whitespace, failed a decode hard gate. These results indicate that the end-to-end task couples too many correctness and performance factors for reliable local agent search.

A final practical limitation is evaluation cost. Each public verification or official evaluation requires substantially more wall-clock time than the circle packing verifier and evaluator. Consequently, the agents could not run many rapid trial-and-error iterations. This reduced the effective search bandwidth: unlike the circle packing task, where agents could quickly test repairs, diagnose failures, and polish candidates, the end-to-end serving task provided slower and noisier feedback.

Moreover, this task is not simply a kernel-writing problem. It requires joint optimization over kernel behavior, serving integration, batching effects, sampling behavior, logprob correctness, and hidden workload latency. Current coding agents in our setup could propose locally sensible engineering changes and use public verifier feedback to filter candidates, but they could not reliably predict how those changes would affect the complete SGLang serving system. This observation motivates the more constrained decode-attention task in Section 5.3, where the optimization surface is reduced to a deterministic tensor operator with local correctness and local timing feedback.

5.3 Decode-Attention Kernel Optimization

Task definition. The decode-attention kernel optimization task isolates one performance-critical operator from the end-to-end serving system. During autoregressive decoding, each new query

Table 4: Main score improvements in the decode-attention optimization run.

Change	**Score**
Triton online-softmax path for single-split workloads	$\sim 1.37\times$
Direct path extended to `max_kv_splits` ≤ 8	$\sim 1.76\times$
Two-stage GQA path for long-context split-16 workloads	$\sim 1.93\times$
Split-16 reducer launch/shape tuning	$2.006\times$
Batch-4 split-8 tail-mask specialization	$\mathbf{2.063}\times$

attends to the accumulated KV cache. The task asks the candidate to optimize this decode-attention operator directly under synthetic but serving-shaped tensor workloads.

This task is narrower than end-to-end kernel optimization. It does not start a model server, does not evaluate HTTP generation latency, and does not expose the full model runner. The candidate is evaluated only as an implementation of the decode-attention operator.

The hardware and software target is kept consistent with the end-to-end serving task: Qwen3-1.7B, BF16 tensors, and a single NVIDIA RTX 4090. The reference baseline is the official SGLang Triton decode-attention implementation. Thus, for each workload, $t_{\text{ref}}(w)$ denotes the latency of the official SGLang decode-attention operator on the same hardware, and $t_c(w)$ denotes the latency of the candidate operator.

Difference from end-to-end optimization. Decode-attention optimization is a simpler task surface than end-to-end serving optimization. The input-output contract is a deterministic tensor operator, the reference oracle is local, the timing measurement is local, and a kernel change maps directly to a per-workload latency change. By contrast, the end-to-end task must preserve full serving behavior across model execution, scheduling, sampling, logprob reporting, and request-level latency measurement. The decode-attention task therefore isolates local kernel optimization, while the end-to-end task evaluates whether such optimization can be integrated into a complete serving system without breaking observable behavior.

Results. The decode-attention task produced a much stronger optimization result than the end-to-end serving task. The best candidate achieved an official score of $2.063\times$, measured as the geometric mean of reference-to-candidate decode-attention latency ratios across the official workloads. The optimization trajectory shows a clear sequence of incremental improvements.

This result explains why narrowing the task surface was effective. Unlike the end-to-end SGLang serving task, this task exposes a deterministic tensor-level operator with local correctness and local timing feedback. The verifier and evaluator report per-workload reference latency, candidate latency, speedup, error statistics, and fallback counts. As a result, the agent could perform a stable shape-by-shape hill climb.

By contrast, the end-to-end task coupled kernel changes with server integration, batching, sampling, logprob correctness, prefill/decode tradeoffs, and hidden hard-gate failures. In the decode-attention task, most of those confounding factors are removed. A candidate change maps more directly to an operator-level latency change, so the optimization framework provides a much cleaner feedback loop for current coding agents.

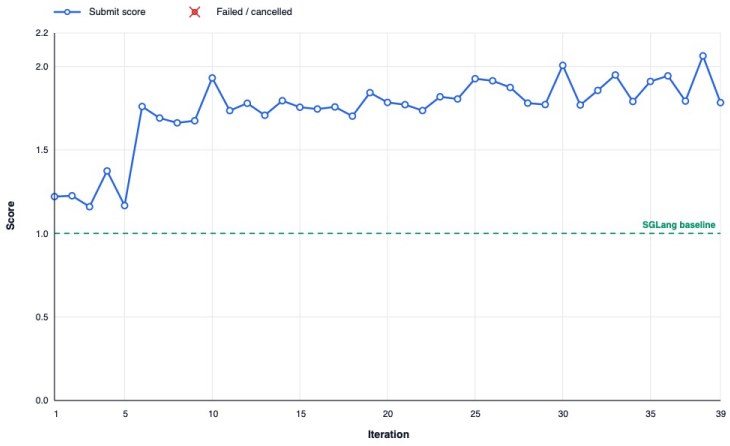

Figure 3: Official submit score progression for the narrowed Qwen3-1.7B decode-attention microkernel task. The dashed line marks the SGLang baseline at 1.0. All official submissions in this run passed validity checks, and the best candidate reaches 2.06× speedup by incrementally specializing the decode-attention path across split and batch regimes.

6 Conclusion

We presented Agentic Optimization, a framework that moves optimization policy into autonomous coding agents while keeping evaluation authority, leaderboard updates, and benchmark boundaries under server control. Across the three task instances, the results show both the promise and the limits of this design. On circle packing, agents used verifier feedback, official submissions, shared tools, and persistent findings to discover configurations that exceeded previously reported CP26 scores. On end-to-end LLM serving optimization, the same agentic interface exposed a harder systems problem: local kernel improvements did not reliably translate into official serving gains because correctness, batching, sampling, logprob reporting, and prefill/decode tradeoffs were coupled inside the full SGLang stack. The decode-attention microkernel task showed that narrowing the task surface can recover a much cleaner optimization loop, producing a 2.063× operator-level speedup over the official SGLang implementation on an RTX 4090.

These experiments suggest that agentic optimization is most effective when the candidate surface, verifier feedback, and evaluator score are closely aligned. When feedback is cheap, local, and directly connected to the official objective, coding agents can perform stable iterative search and reuse accumulated artifacts across sessions. When the evaluator involves a large integrated system with sparse, expensive, or brittle feedback, current agents can still propose plausible engineering changes, but they struggle to predict official outcomes reliably. This distinction motivates future work on better public surrogates for hidden evaluators, more robust serving-level correctness contracts, and task decompositions that preserve realistic objectives while giving agents feedback signals sharp enough for long-horizon search.

References

[1] Alhussein Fawzi et al. “Discovering Faster Matrix Multiplication Algorithms with Reinforcement Learning”. In: *Nature* 610.7930 (2022), pp. 47–53. DOI: 10.1038/s41586-022-05172-4. URL: https://www.nature.com/articles/s41586-022-05172-4.

[2] Bernardino Romera-Paredes et al. "Mathematical Discoveries from Program Search with Large Language Models". In: *Nature* 625.7995 (2024), pp. 468–475. DOI: 10.1038/s41586-023-06924-6. URL: https://www.nature.com/articles/s41586-023-06924-6.

[3] Alexander Novikov et al. *AlphaEvolve: A Coding Agent for Scientific and Algorithmic Discovery*. Tech. rep. Google DeepMind, 2025. arXiv: 2506.13131. URL: https://arxiv.org/abs/2506.13131.

[4] Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. "ShinkaEvolve: Towards Open-Ended and Sample-Efficient Program Evolution". In: *arXiv preprint arXiv:2509.19349* (2025). arXiv: 2509.19349. URL: https://arxiv.org/abs/2509.19349.

[5] Terry Chen et al. "AVO: Agentic Variation Operators for Autonomous Evolutionary Search". In: *arXiv preprint arXiv:2603.24517* (2026). arXiv: 2603.24517. URL: https://arxiv.org/abs/2603.24517.

[6] Ao Qu et al. "CORAL: Towards Autonomous Multi-Agent Evolution for Open-Ended Discovery". In: *arXiv preprint arXiv:2604.01658* (2026). arXiv: 2604.01658. URL: https://arxiv.org/abs/2604.01658.

[7] Haotian Ye et al. "Evaluation-driven Scaling for Scientific Discovery". In: *arXiv preprint arXiv:2604.19341* (2026). arXiv: 2604.19341. URL: https://arxiv.org/abs/2604.19341.

[8] Yecheng Jason Ma et al. "Eureka: Human-Level Reward Design via Coding Large Language Models". In: *arXiv preprint arXiv:2310.12931* (2023). arXiv: 2310.12931. URL: https://arxiv.org/abs/2310.12931.

[9] Guanzhi Wang et al. "Voyager: An Open-Ended Embodied Agent with Large Language Models". In: *arXiv preprint arXiv:2305.16291* (2023). arXiv: 2305.16291. URL: https://arxiv.org/abs/2305.16291.

[10] Henrique Assumpção et al. "CodeEvolve: An Open Source Evolutionary Coding Agent for Algorithm Discovery and Optimization". In: *arXiv preprint arXiv:2510.14150* (2025). arXiv: 2510.14150. URL: https://arxiv.org/abs/2510.14150.

[11] Yongqiang Chen et al. "CausalEvolve: Towards Open-Ended Discovery with Causal Scratchpad". In: *arXiv preprint arXiv:2603.14575* (2026). arXiv: 2603.14575. URL: https://arxiv.org/abs/2603.14575.

[12] John Yang et al. "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering". In: *Advances in Neural Information Processing Systems*. 2024. URL: https://arxiv.org/abs/2405.15793.

[13] Xinghua Lou et al. "AutoHarness: Improving LLM Agents by Automatically Synthesizing a Code Harness". In: *arXiv preprint arXiv:2603.03329* (2026). arXiv: 2603.03329. URL: https://arxiv.org/abs/2603.03329.

[14] Yoonho Lee et al. "Meta-Harness: End-to-End Optimization of Model Harnesses". In: *arXiv preprint arXiv:2603.28052* (2026). arXiv: 2603.28052. URL: https://arxiv.org/abs/2603.28052.

[15] Chris Lu et al. "The AI Scientist: Towards Fully Automated Open-Ended Scientific Discovery". In: *arXiv preprint arXiv:2408.06292* (2024). A later version appeared as "Towards end-to-end automation of AI research" in Nature, 2026. arXiv: 2408.06292. URL: https://arxiv.org/abs/2408.06292.

[16] Yu Li et al. "AutoSOTA: An End-to-End Automated Research System for State-of-the-Art AI Model Discovery". In: *arXiv preprint arXiv:2604.05550* (2026). arXiv: 2604.05550. URL: https://arxiv.org/abs/2604.05550.

[17] Zhengyao Jiang et al. "AIDE: AI-Driven Exploration in the Space of Code". In: *arXiv preprint arXiv:2502.13138* (2025). arXiv: 2502.13138. URL: https://arxiv.org/abs/2502.13138.

[18] Ziming Li et al. "AutoKaggle: A Multi-Agent Framework for Autonomous Data Science Competitions". In: *arXiv preprint arXiv:2410.20424* (2024). arXiv: 2410.20424. URL: https://arxiv.org/abs/2410.20424.

[19] Jun Shern Chan et al. "MLE-bench: Evaluating Machine Learning Agents on Machine Learning Engineering". In: *arXiv preprint arXiv:2410.07095* (2024). arXiv: 2410.07095. URL: https://arxiv.org/abs/2410.07095.

[20] Mert Yuksekgonul et al. "Learning to Discover at Test Time". In: *arXiv preprint arXiv:2601.16175* (2026). arXiv: 2601.16175. URL: https://arxiv.org/abs/2601.16175.