SwiftSolve: A Self-Iterative, Complexity-Aware Multi-Agent Framework for Competitive Programming

Adhyayan Veer Singh
Algoverse Research

Aaron Shen
Algoverse Research

Brian Law
Algoverse Research

Ahmed Ismail
UC Berkeley

Jonas Rohweder
TU Darmstadt
Zuse School ELIZA

Sean O'Brien
Algoverse Research

Kevin Zhu
Algoverse Research

Abstract

Correctness alone is insufficient: LLM-generated programs frequently satisfy unit tests while violating contest time or memory budgets. We present *SwiftSolve*, a complexity-aware multi-agent system for competitive programming that couples algorithmic planning with empirical profiling and complexity-guided repair. We frame competitive programming as a software environment where specialized agents act as programmers, each assuming roles such as planning, coding, profiling, and complexity analysis. A Planner proposes an algorithmic sketch; a deterministic Static Pruner filters high-risk plans; a Coder emits ISO C++17; a Profiler compiles and executes candidates on a fixed input-size schedule to record wall time and peak memory; and a Complexity Analyst fits log–log growth (slope s, R^2) with an LLM fallback to assign a complexity class and dispatch targeted patches to either the Planner or Coder. Agents communicate via typed, versioned JSON; a controller enforces iteration caps and diminishing returns, stopping. Evaluated on 26 problems (16 BigO(Bench), 10 Codeforces Div. 2), three seeds each ($N = 78$ runs) in a POSIX sandbox (2 s / 256–512 MB), SwiftSolve attains $\text{PASS@}1 = 61.54\%$ (16/26) on the first attempt and $\text{SOLVED@}{\le}3 = 80.77\%$ with marginal latency change (mean 11.96 s $\rightarrow$ 12.66 s per attempt). Aggregate run-level success is 73.08% at 12.40 s mean. Failures are predominantly resource-bound, indicating inefficiency rather than logic errors as the principal barrier. Against a Claude Opus 4 single-agent baseline, SwiftSolve improves run-level success (73.1% vs. 52.6%) at $\sim 2\times$ runtime overhead (12.4 s vs. 6.8 s). Beyond correctness (PASS@K), we report efficiency metrics (EFF@K for runtime/memory, incidence of TLE / MLE, and complexity fit accuracy on BigO (Bench), demonstrating that profiling and complexity-guided replanning reduce inefficiency while preserving accuracy. Future studies would integrate comparisons against other multi-agent frameworks and include ablation studies. Our implementation is available at `https://github.com/jonasrohw/swiftsolve/`.

1 Introduction

Code generation is primarily used to automate software development by generating functional and context-aware code from natural language inputs [5]. Automated code generation can lead to enhanced productivity and a reduction in manual coding efforts [16], resulting in more efficient systems. In recent years, multi-agent systems have emerged as a feasible alternative to single-agent frameworks in the domain of LLM-based code generation [7, 8].

Recently, researchers have proposed multi-agent frameworks that allow LLMs to analyze, generate, and iteratively patch code [6, 7, 11, 13, 17]. CodeCoR [13] and CodeSIM [9] are such examples. CodeCoR consists of four LLM-based agents for prompting, task understanding, test case generation, as well as code repairing. Meanwhile, CodeSIM involves three LLM-based agents for planning, code generation, and debugging, with the inclusion of a verification step for the planning agent, mimicking how humans understand, visualize, and refine algorithms.

However, significant gaps remain. While many studies focus on optimizing the correctness of code [7, 8, 19], few studies focus on improving runtime or memory complexity, resulting in inefficient code that could lead to time limit exceeded (TLE) or memory limited exceeded (MLE) errors. Additionally, previous multi-agent frameworks iteratively test and patch code [6, 13], but never consider asymptotic performance, causing slow code generation.

To address these gaps, we introduce SwiftSolve, a novel self-iterative multi-agent framework that can generate correct, complete, and efficient code. Specifically, SwiftSolve consists of 4 LLM-based agents: *1) Planner Agent*, which creates an algorithmic sketch based off of a natural language prompt; *2) Coder Agent*, which generates ISO C++17 code from the algorithmic sketch;*3) Profiler Agent*, which profiles candidates on a fixed, deterministic input-size schedule; and *4) Complexity Analyst Agent*, which will infer time and memory complexity, compares the estimate to contest constraints, and dispatches either a minor fix to the coder agent or an algorithmic overhaul to the planner agent.

In addition, we introduce the use of a lightweight static pruner [13] designed to filter out inefficient *plans* from the Planner before code generation and any expensive LLM calls. This pruner communicates through typed JSON messages and determines whether to return to the Planner (for large errors) or the Coder to allow for cheap, efficient iterations.

We evaluate our framework on two datasets—BigO(Bench) [1] and Codeforces Div. 2 [2]. On a snapshot covering $N{=}78$ task–seed trials (26 tasks × 3 seeds), SwiftSolve attains PASS@1 defined on the first attempt (`replan_0`) of 61.54% $(16/26)$. We report PASS@K and efficiency (EFF@K) metrics from our sandboxed profiler and include a GPT-4.1 Single-Agent baseline.

In summary, our contributions are two-fold:

- We propose SwiftSolve, a novel self-iterative multi-agent framework consisting of 4 distinct agents (i.e, planner agent, coder agent, profiler agent, and complexity analyst) for complete, correct, and effective code generation. A lightweight static pruner filters out inefficient *plans* from the Planner before code generation, routing back to the Planner for large errors or allowing the Coder to proceed.
- We demonstrate the effectiveness of SwiftSolve against GPT-4.1 and Claude 4 Opus through extensive comparison. Experimental results show that SwiftSolve significantly outperforms the baseline.

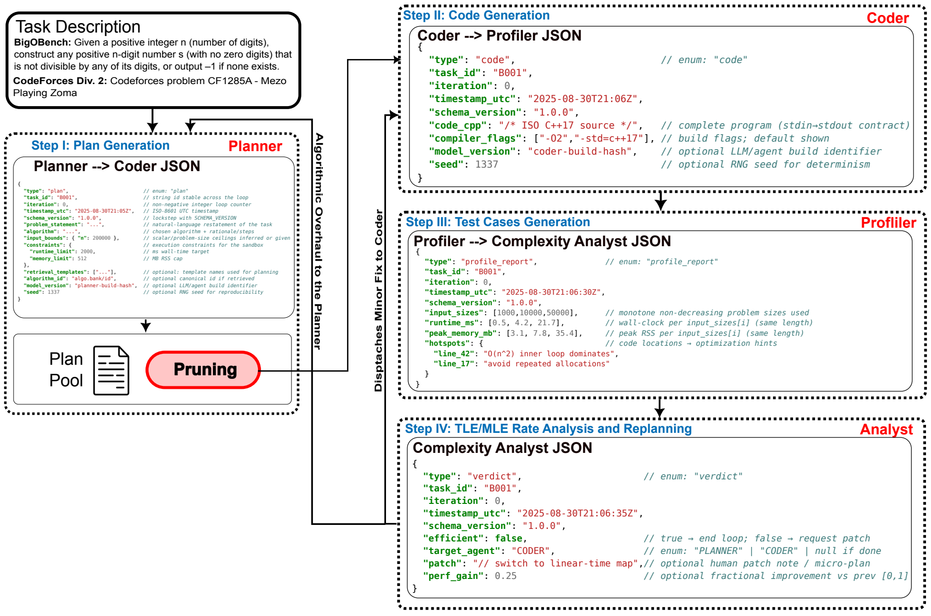
2 Related work

Self-Reflective and Multi-Agent Code Generation There have been multiple advancements in self-iterative multi-agent frameworks for code generation and correctness [4, 5, 12, 14, 16, 18]. For example, Effilearner [6], a self-optimizing framework for code generation, consists of 3 main parts: code generation, overhead profiling, and code refinement. This framework also incorporates a feedback loop designed to decrease code complexity and improve performance. Extensive experiments have corroborated Effilearner's effectiveness on Effibench, HumanEval, and MBPP, reducing overhead and increasing the efficiency of code generation. Unlike Effilearner, SwiftSolve employs four distinct agents, each assigned a specialized role to ensure expertise and efficiency in its designated task.

Similarly, subsequent works such as Liu et. al [11] used a team of LLM agents to learn from each other's successes and failures to improve code optimization tasks. The knowledge gained from each iteration is then demanded and deposited in a bank to be accessed by other agents, fostering a collaborative learning environment [18]. This lays the groundwork for future collaborative LLMs that don't consider an agent's complementary strengths a priori, but still contribute to the overall knowledge and effectiveness of the framework. Runtime speedups and resource efficiency metrics are then calculated to demonstrate code optimization. SwiftSolve, though mirroring the collaborative environment, additionally offers insight into runtime and memory complexity, ensuring framework robustness.

Role Specific LLMs The domain of role-based multi-agent frameworks, in which each agent has a specialized task, has become more prominent in recent years. For example, MapCoder [10], mimicking the human iteration cycle utilizes four agents for retrieval, planning, coding, and debugging. This state-of-the-art method has led to groundbreaking accuracy in code correctness and generation. Similarly, frameworks such as MetaGPT [4] and AgentCoder [7] provide a name, profile, goal, and constraints for each agent to ensure speciality and specific context for their roles. In addition to their specializations, MetaGPT has each individual agent monitor the environment, collecting important observations such as messages from other agents. This ensures the collaborative nature of the framework, allowing agents to assist each other when needed.

3 Methodology



Task Description
BigOBench: Given a positive integer n (number of digits), construct any positive n-digit number s (with no zero digits) that is not divisible by any of its digits, or output –1 if none exists.
CodeForces Div. 2: Codeforces problem CF1285A - Mezo Playing Zoma
Step I: Plan Generation
Planner
Planner --> Coder JSON
Plan Pool
Pruning
Algorithmic Overhaul to the Planner
Disptaches Minor Fix to Coder
Step II: Code Generation
Coder
Coder --> Profiler JSON
Step III: Test Cases Generation
Profliler
Profiler --> Complexity Analyst JSON
Step IV: TLE/MLE Rate Analysis and Replanning
Analyst
Complexity Analyst JSON

```
{
  "type": "plan",                          // enum: "plan"
  "task_id": "B001",                       // string id stable across the loop
  "iteration": 0,                          // non-negative integer loop counter
  "timestamp_utc": "2025-08-30T21:05Z",    // ISO-8601 UTC timestamp
  "schema_version": "1.0.0",               // lockstep with SCHEMA_VERSION
  "problem_statement": "...",              // natural-language restatement of the task
  "algorithm": "...",                      // chosen algorithm + rationale/steps
  "input_bounds": { "n": 200000 },         // scalar/problem-size ceilings inferred or given
  "constraints": {                         // execution constraints for the sandbox
    "runtime_limit": 2000,                 // ms wall-time target
    "memory_limit": 512                    // MB RSS cap
  },
  "retrieval_templates": ["..."],          // optional: template names used for planning
  "algorithm_id": "algo.bank/id",          // optional canonical id if retrieved
  "model_version": "planner-build-hash",   // optional LLM/agent build identifier
  "seed": 1337                             // optional RNG seed for reproducibility
}
```

```
{
  "type": "code",                            // enum: "code"
  "task_id": "B001",
  "iteration": 0,
  "timestamp_utc": "2025-08-30T21:06Z",
  "schema_version": "1.0.0",
  "code_cpp": "/* ISO C++17 source */",      // complete program (stdin→stdout contract)
  "compiler_flags": ["-O2","-std=c++17"], // build flags; default shown
  "model_version": "coder-build-hash",       // optional LLM/agent build identifier
  "seed": 1337                               // optional RNG seed for determinism
}
```

```
{
  "type": "profile_report",             // enum: "profile_report"
  "task_id": "B001",
  "iteration": 0,
  "timestamp_utc": "2025-08-30T21:06:30Z",
  "schema_version": "1.0.0",
  "input_sizes": [1000,10000,50000],    // monotone non-decreasing problem sizes used
  "runtime_ms": [0.5, 4.2, 21.7],       // wall-clock per input_sizes[i] (same length)
  "peak_memory_mb": [3.1, 7.8, 35.4],   // peak RSS per input_sizes[i] (same length)
  "hotspots": {                         // code locations → optimization hints
    "line_42": "O(n^2) inner loop dominates",
    "line_17": "avoid repeated allocations"
  }
}
```

```
{
  "type": "verdict",                         // enum: "verdict"
  "task_id": "B001",
  "iteration": 0,
  "timestamp_utc": "2025-08-30T21:06:35Z",
  "schema_version": "1.0.0",
  "efficient": false,                        // true → end loop; false → request patch
  "target_agent": "CODER",                   // enum: "PLANNER" | "CODER" | null if done
  "patch": "// switch to linear-time map",// optional human patch note / micro-plan
  "perf_gain": 0.25                          // optional fractional improvement vs prev [0,1]
}
```

Figure 1: **SwiftSolve pipeline.** Natural-language prompt → Planner → Static Pruner → Coder → Profiler → Complexity Analyst with JSON feedback to the Coder (and optional Planner) until an efficient solution is reached.

Our system is a modular, message-passing pipeline that co-optimizes code *correctness* and *asymptotic efficiency*. Figure 1 (overview) shows five components orchestrated by a controller: **Planner** → **Static Pruner** → **Coder** → **Profiler** → **Complexity Analyst** (with iterative feedback). Agents communicate solely via typed JSON envelopes backed by Pydantic v2 [3]. All messages carry a frozen header with `type`, `task_id`, `iteration`, `timestamp_utc`, and `schema_version` `"1.0.0"`.

3.1 System architecture

Typed interfaces. Agents exchange only versioned, typed JSON envelopes validated with Pydantic v2. Every envelope includes an immutable header—`type`, `task_id`, `iteration`, `timestamp_utc`, `schema_version`—and a schema-specific payload. Headers provide idempotence and ordering through the (`task_id`, `iteration`) key, and the controller rejects messages with unknown fields or missing required fields to guarantee forward compatibility and auditability. All messages are durably logged and are treated as read-only artifacts once emitted.

PlanMessage is the Planner's machine-consumable contract to the rest of the system. It carries the problem statement in natural language, an algorithm-family label that specifies the intended approach at the level of asymptotic behavior (e.g., two-pointers vs divide–and–conquer), declared input bounds and contest constraints (time in milliseconds and memory in megabytes), and optional retrieval hints for templated subroutines. It also records provenance (`model_version`, `seed`) and an optional algorithm identifier for downstream traceability. The Static Pruner consumes *PlanMessage* to deterministically reject obviously risky plans under the stated bounds before any code generation; plans that pass are the only inputs the Coder is permitted to realize.

CodeMessage transports a complete ISO C++17 translation unit along with the compiler flags required to reproduce the binary that will be profiled. The Coder emits exactly one translation unit per attempt; a thin sanitizer may normalize headers and formatting to ensure successful compilation while preserving semantics. Provenance fields mirror the planner's to enable per-seed determinism studies. The Profiler is the sole consumer of *CodeMessage*; no other agent mutates code payloads, and any subsequent edits must be expressed as a new *CodeMessage* at a higher iteration index.

ProfileReport is the Profiler's measurement artifact. It returns the deterministic input schedule used for probing, the corresponding vectors of wall-clock runtime and peak resident memory for each input size, and a lightweight hotspot map with textual notes on potential bottlenecks. Failures are represented in-place: timeouts or nonzero exits are encoded as missing or $+\infty$ entries for the affected sizes while allowing the remainder of the schedule to complete. The report additionally includes the slope and R^2 of a least-squares fit in log–log space and the resulting complexity label when available; these quantities are consumed by the Complexity Analyst and the controller for routing decisions, patch synthesis, and termination checks. Full schema definitions are deferred to the appendix.

Execution target and toolchain. All candidate programs are single-translation-unit ISO C++17. The Profiler compiles with `g++ -O2 -std=c++17 -march=native -ffast-math` (two attempts) and executes the binary under `/usr/bin/time -v` to obtain elapsed wall-clock time and maximum resident set size (RSS). Per-run process timeouts enforce liveness (default: **2 s**).

Controller. The controller invokes agents, logs all JSON payloads, and enforces safety (agent-failure budget, iteration cap, diminishing-returns stop). Outcomes are one of: `success`, `static_prune_failed`, or `agent_failure` (budget exhausted or insufficient gain).

3.2 Agent Roles

3.2.1 Planner

The Planner (Claude 4 Opus) converts the natural-language task into a minimal, machine-consumable plan: `algorithm` (e.g., "two_pointers"), integer `input_bounds` (e.g., `{"n": 100000}`), and integer `constraints` (e.g., `{"runtime_limit": 2000, "memory_limit": 512}`). On re-planning, the system prompt *requires a different algorithmic family* and to address performance feedback.

3.2.2 Static Pruner

Before any LLM generation is expended on code, a deterministic gate rejects plans that are provably risky given the declared bounds. Let $n =$ `input_bounds["n"]` (default 0 if absent) and $\alpha =$ `algorithm` lowercased. We reject if any of the following triggers fire:

1. **While-loop multiplicity at large n:** $\texttt{count}(\alpha, \text{"while"}) > 2$ and $n \geq 10^5$.
2. **Unbounded recursion at large n:** the substring "recursion" occurs in α and $n \geq 10^4$.
3. **Sort-inside-loop pattern at moderate n:** a regex approximating "`for ...: ... sort(...)`" matches α and $n \geq 10^3$.

If any rule triggers, the controller returns `static_prune_failed`. Otherwise the plan advances.

3.2.3 Coder

The Coder (GPT-4.1) emits a complete, compilable C++17 program. Two modes are supported: *(i) initial synthesis* from the plan alone, and *(ii) patch application* where the system prompt mandates incorporation of the Analyst's optimization. A thin sanitizer adds missing standard headers and repairs common formatting issues, but otherwise preserves the model's output.

3.2.4 Profiler

Given a `CodeMessage`, the Profiler compiles and executes the candidate against a fixed input schedule and returns a `ProfileReport`.

- **Compilation.** `g++ -O2 -std=c++17 -march=native -ffast-math`; one automatic retry on failure.
- **Input schedule.** Deterministic sizes $\langle 0,\ 1,\ 10^3,\ 5 \cdot 10^3,\ 10^4,\ 5 \cdot 10^4,\ 10^5 \rangle$. The default generator writes a single integer line "`n\n`" (task-specific generators are pluggable; see §3.3).
- **Telemetry.** `/usr/bin/time -v` provides wall-clock and max RSS; outputs are parsed via regex. Each per-n execution inherits a process-level timeout (2 s). On timeout or nonzero exit, that point is recorded as $+\infty$ in both series and the loop continues.
- **Hotspots.** A lightweight map records crash notes or hints; gprof-based attribution is stubbed and does not affect routing.

3.2.5 Complexity Analyst

The Analyst integrates regression-based fitting with an LLM fallback to assign a time-complexity class and to synthesize targeted optimizations. Let $D = \{(n_i, t_i)\}$ be valid points where $t_i > 0$ and finite.

1. **Slope fitting.** We fit a line in log–log space between input size and runtime, and compute the slope s together with the coefficient of determination R^2. The choice of logarithm base does not affect s or R^2.
2. **Ambiguity test.** Mark the curve ambiguous if any hold: $R^2 < 0.7$; $s \in (0.4, 0.6) \cup (1.3, 1.7) \cup (2.3, 2.7)$; significant non-monotone noise (at least one $> 10\%$ increase and one $> 10\%$ decrease); $s < -0.5$ or $s > 10$; or the size range $< 10\times$.
3. **Classification.** If unambiguous, map by thresholds: $s < 0.5 \Rightarrow \mathcal{O}(1)$, $s < 1.5 \Rightarrow \mathcal{O}(n)$, $s < 2.5 \Rightarrow \mathcal{O}(n^2)$, else $\mathcal{O}(n^k)$ (coarse bucket). If ambiguous, query GPT-4.1 for a *single* label from $\{\mathcal{O}(1), \mathcal{O}(\log n), \mathcal{O}(n), \mathcal{O}(n \log n), \mathcal{O}(n^2), \mathcal{O}(n^3), \mathcal{O}(2^n), \mathcal{O}(n!)\}$ and normalize the response.
4. **Efficiency decision and patching.** We set `efficient` = `true` iff the final class $\in \{\mathcal{O}(1), \mathcal{O}(\log n), \mathcal{O}(n), \mathcal{O}(n \log n)\}$. Otherwise, the Analyst emits `target_agent=CODER` and a natural-language `patch` (e.g., replace nested loops with hash lookups; adopt two pointers; reduce allocations). Memory series are inspected in patch logic (e.g., flagging excessive growth for otherwise linear code) but do not currently gate `efficient`.

3.3 Dynamic Profiling

We profile each candidate across a fixed, log-spaced schedule augmented with corner cases ($n \in \{0, 1\}$). This provides stable variance and comparable growth curves across tasks. The first version employs a *task-agnostic* input generator that only writes n. While this stresses raw iteration/recursion costs and common STL usage, it under-exercises branch-dependent worst cases for some problems (e.g., value distributions for two-pointers or pivot patterns in quicksort). The API is designed for drop-in *task-specific adversarial generators* (dense/sparse graphs, nearly-sorted arrays, pathological duplicates); integrating these does not change any upstream schema.

3.4 Complexity Fitting

Complexity fitting follows the *Complexity Analyst* procedure in §3.2. From the Profiler we obtain $D = \{(n_i, t_i)\}$ and estimate the slope s and coefficient of determination R^2 via least-squares in log–log space; class assignment and the LLM fallback for ambiguous fits are exactly as specified in §3.2.5. We retain D, s, R^2, and the assigned class; these fields are emitted in `ProfileReport` and consumed by the Analyst and the judge. Peak memory (*peak_memory_mb*) is recorded concurrently.

3.5 Replanning Policy

After iteration k produces profile $\{(n_i, t_i^{(k)})\}$, we compute a fractional improvement on the largest input size:

$$\mathrm{gain}^{(k)} = \frac{t_{\max}^{(k-1)} - t_{\max}^{(k)}}{t_{\max}^{(k-1)}}, \qquad t_{\max}^{(k)} \equiv t_{i^\star}^{(k)} \text{ with } n_{i^\star} = \max_i n_i. \tag{1}$$

If $\mathrm{gain}^{(k)} < \delta$, we terminate; otherwise we apply the Analyst's `patch` in the next Coder call. Routing is governed by `target_agent`; the current implementation primarily targets the Coder, while the controller also supports Planner-directed replans that generate a *different* algorithm; the budget is capped at three attempts per task (`replan_0`, `_1`, `_2`).

3.6 Termination Criteria and Safety Guards

The loop stops under any of the following conditions:

1. **Success:** the Analyst returns `efficient=true`.
2. **Diminishing returns:** $\mathrm{gain}^{(k)} < \delta$ (configurable).
3. **Iteration cap:** at most **3** iterations (controller default).
4. **Static rejection:** the Static Pruner rejects the plan.
5. **Agent-failure budget:** two caught exceptions across Planner/Coder/Profiler/Analyst abort the run.

Process-level safety includes isolated temp workspaces, strict per-run timeouts (2 s), capture of stdout/stderr, and graceful point-wise failure handling (recorded as $+\infty$ without crashing the loop).

4 Experimental setup

Tasks and datasets. We evaluate on two sources comprising **26 tasks** total: **16** from BigO(Bench) [1] (binary_search, linear_scan, two_pointers, hash_lookup, prefix_sum, sliding_window, merge_sort, quick_sort, heap_operations, bfs_graph, dfs_graph, dijkstra, dynamic_programming, backtracking, matrix_multiplication, segment_tree) and **10** Codeforces Div. 2 problems [2] (e.g., *cf_1285a_maximum_array*, *cf_1285b_just_eat_it*, *cf_1285c_fadi_and_lcm*, *cf_1285d_dr_evil_underscores*). Unless otherwise stated, all analyses operate on this same fixed task list. The snapshot comprises N=78 *task–seed trials* (26 tasks × 3 seeds). Within each trial, the controller permits up to three attempts on the *same* problem: `replan_0`, `replan_1`, `replan_2` (i.e., at most two replans beyond the initial).

Environment and sandbox. Candidates are compiled and executed in a POSIX sandbox with per-task limits taken from the harness metadata. Runtime and memory limits mirror the source benchmarks for comparability: BigO(Bench) tasks use a 2 s wall-clock budget and 512 MB memory as specified by the harness [1]; Codeforces Div.2 problems use the platform's standard 2 s and 256 MB limits [2]. We adopt the per-task limits from these sources without modification. Each run logs wall time, peak RSS, exit status, and (when available) per-input profiling traces (`input_sizes`, `runtime_ms`, `peak_memory_mb`) together with boolean TLE/MLE flags over the input grid.

Agent protocol. For each task we run up to three attempts with lightweight replanning between attempts (`replan_0`, `replan_1`, `replan_2`). Each attempt yields one compiled candidate judged on the hidden test harness. A task is marked `success` if any of its attempts pass all checks; otherwise `agent_failure`.

Metrics. PASS@1 refers strictly to the first attempt (`replan_0`) on each task. **Solved@**$\leq k$ is the fraction of the same task list solved within at most k attempts (cumulative over `replan_0`, `_1`, `_2` on the identical problems). EFF@K (runtime/memory) is pass@k under an efficiency predicate: no TLE/MLE anywhere on the profiled grid and a fitted runtime curve that extrapolates under the task limits at $n_{\max}$. We primarily report EFF@1 for time and memory. When more than k attempts exist, we use the unbiased order-statistics estimator from ENAMEL [15].

TLE/MLE rate: the fraction of probed input sizes that triggered time limit exceeded (TLE)/memory limit exceeded (MLE) for a candidate; summarized per task and in aggregate.

Complexity-fit accuracy (BigO(Bench)). For solutions with per-size profiles, we assign a label via the slope-threshold procedure with an LLM fallback exactly as specified in §3.2.5, and report the fraction matching the benchmark's ground-truth Big-O.

Evaluation summary. Table 1 reports headline numbers. We define PASS@1 on the first attempt only (`replan_0`); under this definition PASS@1 is $16/26{=}61.54\%$. Replanning indices (`replan_0`, `_1`, `_2`) are successive attempts on the *same* problems under a fixed seed. The sweep uses 26 tasks × 3 seeds = **78** runs in total; the mean per-run duration is {12.40s.

Table 1: Aggregate run-level statistics. Cumulative wall time: 16.12 minutes.

	Total runs	Successes	SOLVED@$\leq$3	Avg. time / run
Overall	78	57	73.08%	12.40 s

Replanning Table 2 summarizes outcomes by attempt index. Success rate improves monotonically with additional replanning budget, with a modest change in latency.

Table 2: Effect of replanning budget. "Tasks" counts the same 26 problems evaluated at each attempt index; percentages are cumulative solved rates over that fixed set.

Attempt	Tasks	SOLVED@$\leq k$	Avg. time / attempt
`replan_0`	26	61.54%	11.96 s
`replan_1`	26	76.92%	12.58 s
`replan_2`	26	80.77%	12.66 s

Per-source breakdown. Table 3 shows performance by dataset source. BigO(Bench) runs are faster on average and exhibit higher acceptance than the Codeforces-style subset.

Table 3: Performance by dataset source.

Source	Count	SOLVED@$\leq$3	Avg. time / run
BigO(Bench)	49	77.55%	10.30 s
Codeforces subset	29	65.52%	15.95 s

Computation of metrics from logs. All results in Tables 1–3 are computed directly from the evaluation JSON (`evaluation_summary`, `replan_analysis`, and `research_insights`). For EFF@K and complexity-fit, we use the per-run profiling arrays (`input_sizes`, `runtime_ms`, `peak_memory_mb`) and the sandbox limits recorded under `constraints`. TLE/MLE rates are averaged over the Boolean flag vectors provided for each attempt. We release the raw JSON alongside the paper for exact recomputation.

5 Results

5.1 Main Performance Comparison

In total 78 runs were executed (Table 1). Of these, 57 runs completed successfully, yielding an overall success rate of 73.1%. The mean execution time per run was 12.4 seconds. Table 1 summarizes these aggregate metrics. The two datasets show different results: BigO(Bench) problems (49 runs) achieved 77.6% success with a 10.3 s average time, whereas Codeforces problems (29 runs) had

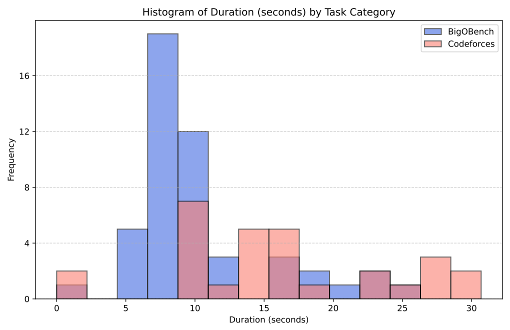
65.5% success with a 15.95 s average (Table 3, Figure 2a). These figures indicate that performance varied by task type.

Dataset breakdown: BigO(Bench) – 49 runs, 77.6% success, 10.3 s avg. Codeforces – 29 runs, 65.5% success, 15.95 s avg.

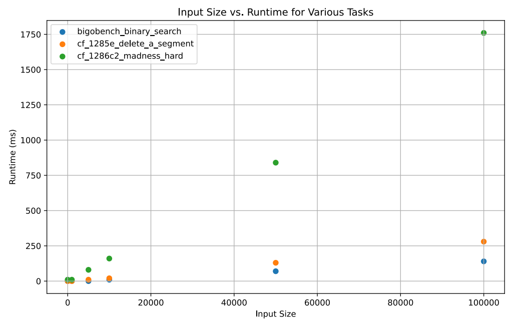
All 21 failed runs were attributable to efficiency problems rather than coding errors. Several plans were rejected immediately by the static pruner (e.g., excessive nested loops), and the rest terminated after the maximum iterations when the profiler continued to report TLE/MLEs.

5.2 Runtime Curves

For each solved task, execution time was recorded over multiple input sizes and fit with an empirical time-complexity model using BigO(Bench). Figure 2b shows representative runtime-vs-input-size curves from these measurements. In all cases the measured runtime grew monotonically, matching expected asymptotic behavior (e.g. near-linear or $n \log n$ growth). No abrupt or non-monotonic jumps were observed.



(a) Histogram of the duration (seconds) of each run on the BigO(Bench) and Codeforces datasets.



(b) Input size vs. runtime for Big O Bench and Codeforces Div. 2 problems.

Figure 2: Runtime distributions and scaling across benchmarks.

Notably, runs that required additional replanning tended to incur slightly higher runtimes (reflecting extra profiling and code-synthesis steps), consistent with the small increase in mean durations seen in Section 5.1. The profiling curves confirm that most solutions exhibit polynomial growth and that the system correctly identifies their complexity limits.

5.3 TLE/MLE Rates

A total of 21 runs (26.9% of 78) ended by exceeding contest resource limits. Of these failures, 16 runs (20.5%) were time-limit exceeded (TLE) and 5 runs (6.4%) were memory-limit exceeded (MLE). Thus the overall TLE/MLE failure rate was 26.9%. Figure 9a reports these values. This result is consistent with the known sensitivity of CP problems to inefficiency. Most failures were resource-limit violations; in total, we observed 16 TLE and 5 MLE runs. A small number of runs were terminated early by the static pruner (plan rejection).

5.4 Iteration Count Analysis

Iterative replanning substantially improved success. Figure 9b visualizes these results. Without replanning, 62% of tasks were solved. Allowing one replanning iteration increased cumulative success to 77%, and allowing two iterations brought it to 81%. In other words: 0 replans (initial run only): 62% tasks solved; 1 replan (two attempts): 77% cumulative solved; 2 replans (three attempts): 81% cumulative solved.

Across the **same** 26 problems, 7 were solved only after one or two replanning steps, indicating that replanning rescued a significant subset of cases. We report cumulative **Solved@**$\leq k$ over this identical task list: 0 replans (first attempt only), 1 replan (two attempts), and 2 replans (three attempts).

Returns beyond the second replan were marginal. The distribution of iteration counts is plotted in Figure 9b.

5.5 Baselines

We compare SwiftSolve against two single-pass LLM baselines: direct GPT-4.1 and direct Claude 4 Opus (single-attempt). The GPT-4.1 baseline achieves roughly 60.4% success on BigO(Bench) and 44.4% on the Codeforces suite, with average runtimes of about 5.1s and 9.8s respectively. The baseline Claude-Opus is similar: approximately 62. 5% on BigO and 46.7% on Codeforces (avg. 5.9s and 5.5s). In contrast, SwiftSolve's multi-agent approach yields substantially higher success on both datasets: its gains are especially pronounced on the harder Codeforces tasks. These improvements come with a trade-off in speed: SwiftSolve's structured planning (multiple agent iterations) leads to longer total runtime per problem.

On Codeforces problems, GPT-4.1's answers ran much slower (9.8s vs 5.5s) than Claude's, highlighting inefficiency. Because SwiftSolve's planner uses the Claude 4 Opus model, the Claude baseline provides the most direct comparison. SwiftSolve builds on this by iteratively decomposing problems and pruning plans, which raises overall success (especially on Codeforces) at the cost of higher runtime. Overall, SwiftSolve outperforms both GPT-4.1 and Claude-Opus baselines in solution accuracy while demonstrating the expected runtime/accuracy trade-off inherent in more complex planning.

6 Discussion

SwiftSolve's gains stem from coupling algorithmic planning with empirical profiling and complexity analysis as part of the objective rather than a post-hoc property. Iterative replanning *rescues* a meaningful subset of tasks, and dynamic profiling with coarse complexity fitting aligns edits toward the bottleneck rather than blind retries. A lightweight static pruner cheaply filters plans that would deterministically blow time or memory, and the remaining failures are overwhelmingly TLE/MLE errors rather than logic mishaps, indicating computational cost is the principal barrier. Empirically, SOLVED@$\leq$3 increases across 0$\rightarrow$1$\rightarrow$2 replans (61.54$\rightarrow$76.92$\rightarrow$80.77%) while mean runtime changes only marginally (11.96$\rightarrow$12.58$\rightarrow$12.66 s), indicating improvement without proportional latency cost. Taken together, these signals shift the objective from "pass once" to "pass under resource limits," yielding a resource-aware system.

Several directions merit investigation. **(1) Complexity labeling is threshold sensitive.** The slope–fit procedure discretizes continuous growth and can be unstable at small n. Although we quantify accuracy, non-parametric alternatives remain open. **(2) Iteration budget and stopping.** We justify a two-replan cap with a budget-sensitivity sweep, but stopping is still heuristic; per-task adaptive budgets and confidence-based halting can be covered in future work. **(3) External validity.** Results cover two benchmarks, a single language/runtime, and one hardware profile. Future work could include generalization to other languages, toolchains, I/O regimes, and stricter resource policies. **(4) Attribution and causality.** Beyond the pruner ablation, our analysis is largely observational. We do not localize where efficiency gains arise (e.g., knockout or patch-type causality).

7 Conclusion

We introduced an efficiency-aware evaluation that couples standard acceptance metrics (pass@k) with resource-centric measures (eff@k runtime/memory), TLE/MLE incidence, and complexity-fit accuracy. In a snapshot of 78 runs, the overall run success rate was 73. 08%, and a three-attempt replanning protocol increased the cumulative Solved @ k from 61.54% to 80.77% with only marginal latency change. Performance was stronger and faster on BigO(Bench) (77.55%, 10.30 s) than on the Codeforces-style subset (65.52%, 15.95 s), showing sensitivity to task structure and difficulty. Profiling-derived signals expose efficiency failure modes that correctness-only metrics can miss. Due to compute limits, the task set is modest; future work will add public baselines, scale datasets and languages, strengthen complexity inference, and evaluate robustness (adversarial inputs, distribution shift). For safety and reproducibility we use sandboxed execution, avoid data leakage, and will pursue gated access, provenance/watermarking, energy reporting, and expanded misuse mitigations.

Acknowledgments and Disclosure of Funding

Jonas Rohweder is supported by the Konrad Zuse School of Excellence in Learning and Intelligent Systems (ELIZA) through the DAAD programme Konrad Zuse Schools of Excellence in Artificial Intelligence, sponsored by the Federal Ministry of Education and Research.

Competing interests

The authors declare no competing interests.

References

[1] Pierre Chambon, Baptiste Roziere, Benoit Sagot, and Gabriel Synnaeve. BigO(bench) – can LLMs generate code with controlled time and space complexity? URL http://arxiv.org/abs/2503.15242.

[2] Codeforces. Codeforces: A competitive programming platform. https://codeforces.com, 2025. Accessed: 2025-07-07.

[3] Samuel Colvin, Eric Jolibois, Hasan Ramezani, Adrian Garcia Badaracco, Terrence Dorsey, David Montague, Serge Matveenko, Marcelo Trylesinski, Sydney Runkle, David Hewitt, Alex Hall, and Victorien Plot. Pydantic validation, 2025. URL https://docs.pydantic.dev/latest/.

[4] Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. URL http://arxiv.org/abs/2308.00352.

[5] Md Sifat Hossain, Anika Tabassum, Md Fahim Arefin, and Tarannum Shaila Zaman. LLM-ProS: Analyzing large language models' performance in competitive problem solving. URL http://arxiv.org/abs/2502.04355.

[6] Dong Huang, Jianbo Dai, Han Weng, Puzhen Wu, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie M. Zhang. EffiLearner: Enhancing efficiency of generated code via self-optimization, . URL http://arxiv.org/abs/2405.15189.

[7] Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. AgentCoder: Multi-agent-based code generation with iterative testing and optimisation, . URL http://arxiv.org/abs/2312.13010.

[8] Yoichi Ishibashi and Yoshimasa Nishimura. Self-organized agents: A LLM multi-agent framework toward ultra large-scale code generation and optimization. URL http://arxiv.org/abs/2404.02183.

[9] Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. CODESIM: Multi-agent code generation and problem solving through simulation-driven planning and debugging. URL http://arxiv.org/abs/2502.05664.

[10] Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving, 2024. URL https://arxiv.org/abs/2405.11403.

[11] Yuanzhe Liu, Ryan Deng, Tim Kaler, Xuhao Chen, Charles E. Leiserson, Yao Ma, and Jie Chen. Lessons learned: A multi-agent framework for code LLMs to learn and improve. URL http://arxiv.org/abs/2505.23946.

[12] Daniel Nichols, Pranav Polasam, Harshitha Menon, Aniruddha Marathe, Todd Gamblin, and Abhinav Bhatele. Performance-aligned LLMs for generating fast code. URL http://arxiv.org/abs/2404.18864.

[13] Ruwei Pan, Hongyu Zhang, and Chao Liu. CodeCoR: An LLM-based self-reflective multi-agent framework for code generation. URL `http://arxiv.org/abs/2501.07811`.

[14] Ruizhong Qiu, Weiliang Will Zeng, James Ezick, Christopher Lott, and Hanghang Tong. How efficient is LLM-generated code? a rigorous & high-standard benchmark. URL `http://arxiv.org/abs/2406.06647`.

[15] Ruizhong Qiu, Weiliang Will Zeng, James Ezick, Christopher Lott, and Hanghang Tong. How efficient is llm-generated code? a rigorous & high-standard benchmark. In *International Conference on Learning Representations (ICLR)*, 2025. URL `https://arxiv.org/abs/2406.06647`. ENAMEL.

[16] Melika Sepidband, Hamed Taherkhani, Song Wang, and Hadi Hemmati. Enhancing LLM-based code generation with complexity metrics: A feedback-driven approach. URL `http://arxiv.org/abs/2505.23953`.

[17] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. URL `http://arxiv.org/abs/2303.11366`.

[18] Minnan Wei, Ziming Li, Xiang Chen, Menglin Zheng, Ziyan Qu, Cheng Yu, Siyu Chen, and Xiaolin Ju. Evaluating and improving large language models for competitive program generation. URL `http://arxiv.org/abs/2506.22954`.

[19] Yang Zhang, Shixin Yang, Chenjia Bai, Fei Wu, Xiu Li, Zhen Wang, and Xuelong Li. Towards efficient LLM grounding for embodied multi-agent collaboration. URL `http://arxiv.org/abs/2405.14314`.

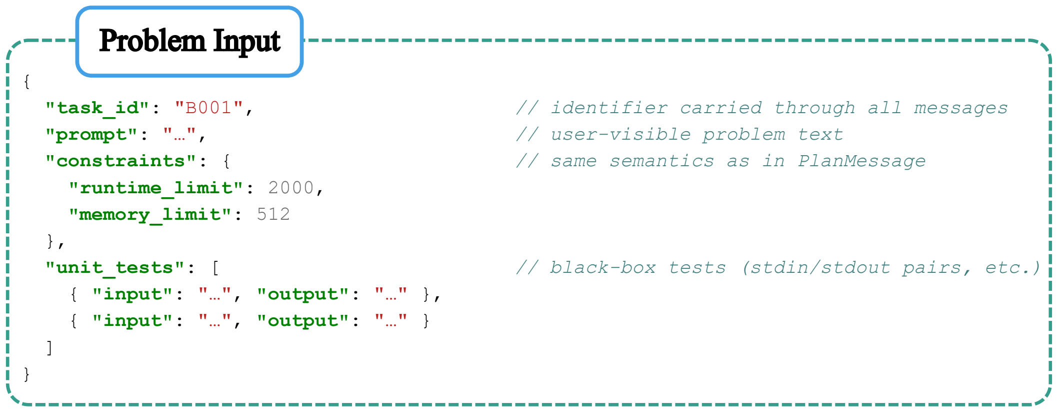
Appendix

A Detailed Prompting of SwiftSolve

The detailed prompting schema of the Planner, Coder, Profiler, and Complexity Analyst are shown in Figure 3, 4, 5, and 6, respectively. Additionally, we visualize the competitive programming problem schema input and output for SwiftSolve, noted in Figure 7 and 8, respectively.

B Example Dataset Schema

Example dataset schemas for both BigOBench and Codeforces Div. 2 problems, showing all the tasks we used, can be found at this link.


Problem Input

```
{
  "task_id": "B001",                      // identifier carried through all messages
  "prompt": "…",                          // user-visible problem text
  "constraints": {                        // same semantics as in PlanMessage
    "runtime_limit": 2000,
    "memory_limit": 512
  },
  "unit_tests": [                         // black-box tests (stdin/stdout pairs, etc.)
    { "input": "…", "output": "…" },
    { "input": "…", "output": "…" }
  ]
}
```

Figure 7: Schema input for competitive programming problems into SwiftSolve.

Planner Agent

```
{
"type": "plan",                             // enum: "plan"
"task_id": "B001",                          // string id stable across the loop
"iteration": 0,                             // non-negative integer loop counter
"timestamp_utc": "2025-08-30T21:05Z",       // ISO-8601 UTC timestamp
"schema_version": "1.0.0",                  // lockstep with SCHEMA_VERSION
"problem_statement": "...",                 // natural-language restatement of the task
"algorithm": "...",                         // chosen algorithm + rationale/steps
"input_bounds": { "n": 200000 },            // scalar/problem-size ceilings inferred or given
"constraints": {                            // execution constraints for the sandbox
  "runtime_limit": 2000,                    // ms wall-time target
  "memory_limit": 512                       // MB RSS cap
},
"retrieval_templates": ["..."],             // optional: template names used for planning
"algorithm_id": "algo.bank/id",             // optional canonical id if retrieved
"model_version": "planner-build-hash",      // optional LLM/agent build identifier
"seed": 1337                                // optional RNG seed for reproducibility
}
```

Figure 3: Schema output for the planner agent.

Coder Agent

```
{
"type": "code",                             // enum: "code"
"task_id": "B001",
"iteration": 0,
"timestamp_utc": "2025-08-30T21:06Z",
"schema_version": "1.0.0",
"code_cpp": "/* ISO C++17 source */",       // complete program (stdin→stdout contract)
"compiler_flags": ["-O2","-std=c++17"],     // build flags; default shown
"model_version": "coder-build-hash",        // optional LLM/agent build identifier
"seed": 1337                                // optional RNG seed for determinism
}
```

Figure 4: Schema output for the coder agent.

Run Result

```
{
  "type": "run_result",                     // enum: "run_result"
  "task_id": "B001",
  "status": "success",                      // enum: success | static_prune_failed | failed | sandbox_error
  "timestamp_utc": "2025-08-30T21:07:10Z",
  "schema_version": "1.0.0",
  // Present on success:
  "code_cpp": "/* final C++17 program */",// last efficient artifact
  "profile": {                              // embedded ProfileReport of the winning iter
    "type": "profile_report",
    "task_id": "B001",
    "iteration": 1,
    "timestamp_utc": "2025-08-30T21:06:55Z",
    "schema_version": "1.0.0",
    "input_sizes": [1000,10000,50000],
    "runtime_ms": [0.2, 1.6, 8.9],
    "peak_memory_mb": [3.1, 6.2, 28.5],
    "hotspots": {}
  },
  // Present on failure:
  "error_message": null,                    // diagnostic string if failed
  "last_verdict": null                      // final VerdictMessage (as dict) for debugging
}
```

Figure 8: Schema output for competitive programming problems into SwiftSolve.

Profiler Agent

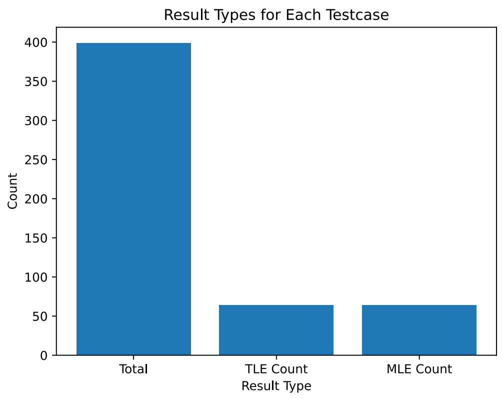
```
{
  "type": "profile_report",                 // enum: "profile_report"
  "task_id": "B001",
  "iteration": 0,
  "timestamp_utc": "2025-08-30T21:06:30Z",
  "schema_version": "1.0.0",
  "input_sizes": [1000,10000,50000],        // monotone non-decreasing problem sizes used
  "runtime_ms": [0.5, 4.2, 21.7],           // wall-clock per input_sizes[i] (same length)
  "peak_memory_mb": [3.1, 7.8, 35.4],       // peak RSS per input_sizes[i] (same length)
  "hotspots": {                             // code locations → optimization hints
    "line_42": "O(n^2) inner loop dominates",
    "line_17": "avoid repeated allocations"
  }
}
```

Figure 5: Schema output for the profiler agent.

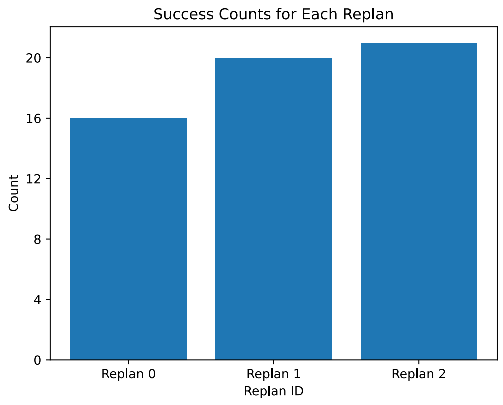
Complexity Analyst

```
{
  "type": "verdict",                         // enum: "verdict"
  "task_id": "B001",
  "iteration": 0,
  "timestamp_utc": "2025-08-30T21:06:35Z",
  "schema_version": "1.0.0",
  "efficient": false,                        // true → end loop; false → request patch
  "target_agent": "CODER",                   // enum: "PLANNER" | "CODER" | null if done
  "patch": "// switch to linear-time map",// optional human patch note / micro-plan
  "perf_gain": 0.25                          // optional fractional improvement vs prev [0,1]
}}
```

Figure 6: Schema output for the complexity analyst.



(a) TLE and MLE rates for iterative replanning failures.



(b) Amount of successful iterative replans.

Figure 9: Error incidence and recovery patterns under iterative replanning.