FROM HOTSPOTS TO EDIT LOCATIONS: GUIDING REPOSITORY-LEVEL CODE OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Modern automatic code optimization approaches rely on an LLM to optimize code. Given the cost of invoking an LLM and its limited context length, in large repositories, we must choose where to optimize within a limited budget. Profiling identifies costly functions, called hotspots, but optimizing hotspots alone may not be sufficient: we may also need to optimize their (direct or indirect) callers to reduce the calls to hotspots. However, the callers may be too many to examine within a limited budget. We introduce CHICO, a localization approach that combines profiling information with optimization opportunity detection. Given a hotspot function and its callers, our approach uses a lightweight static analysis to detect potential opportunities to reduce the number of calls to the hotspot in the callers, and considers these callers for the LLM to optimize. We implement CHICO as a prototype that integrates this localization approach with SemOpt, a state-of-the-art optimization approach for a single function. We evaluate the system on GSO and SWE-fficiency-Lite against four baselines using four language models. Our approach successfully optimizes 10.29–123.53% more cases than the baselines on GSO and 12.50–239.13% on SWE-fficiency-Lite.

1 INTRODUCTION

Automatic code optimization improves program performance while preserving program behavior. Traditional rule-based approaches apply expert-written transformations; supporting additional optimization strategies therefore requires manual efforts for creating new rules (Bacon et al., 1994; Garg et al., 2025). Recently, large language models (LLMs) have become a major approach to automatic code optimization. They generate code changes using optimization knowledge and refine them through execution feedback (Garg et al., 2025; Shypula et al., 2024; Peng et al., 2025; Zhao et al., 2026). Recent systems extend these capabilities from individual functions to entire repositories (Bai et al., 2025; Ren et al., 2025).

In this paper, we study repository-level automatic code optimization: given a repository and a performance test specifying the operations and inputs to measure, the goal is to improve performance on that test while preserving program behavior. Repositories can contain hundreds of thousands of lines of code (Jimenez et al., 2024), making it impractical to inspect and attempt to optimize every function within a limited time and resource budget. Effective optimization therefore depends on *localization*: narrowing the search to functions where edits are likely to improve performance.

Existing approaches address localization either with additional user input or through profiling, which measures execution costs and call relationships while running a performance test. On the one hand, PEACE (Ren et al., 2025) and Optimo (Peng et al., 2026) require an initial optimization target to be supplied. This additional input may be unavailable. On the other hand, existing local optimization approaches, including SemOpt (Zhao et al., 2026) and RAPGen (Garg et al., 2025), which are designed to optimize individual functions, apply to repositories by applying to the most time-consuming hotspot functions one by one. POLO (Bai et al., 2025) further includes the callers of the hotspot functions in the optimization list and ranks these functions by propagating the execution costs of callees to their callers. These approaches are fully automated in the scenario where performance tests are available, but focus on the profiled cost of the hotspot functions.

However, profiled cost alone is insufficient to guide the optimization. Based on the well-known Amdahl's law (Amdahl, 1967), the potential speedup of a function is not only decided by its profiled

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

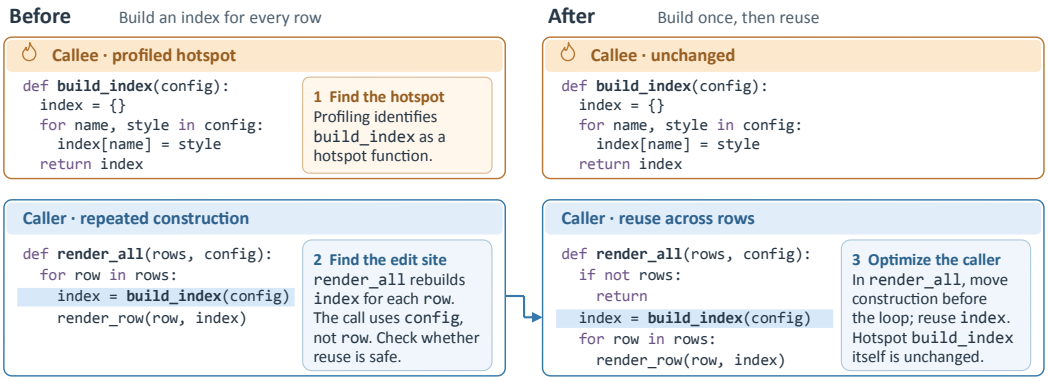

Figure 1: Motivating example. Profiling identifies `build_index` as a hotspot. Its caller, `render_all`, rebuilds an index for every row from `config`. This call suggests investigating reuse in the caller. If the inputs remain stable and the result can be safely shared, moving construction before the loop reduces repeated calls while leaving the hotspot unchanged.

cost, but also the potential room for improvement. Figure 1 illustrates such a case: the hotspot, `build_index`, repeatedly constructs an index because `render_all`, its caller, calls it once for every row. Though `build_index` is the most time-consuming function, it has little room for improvement. In contrast, `render_all`, though it is less time-consuming, has more room for improvement because it repeatedly calls `build_index`. These calls can be eliminated by moving the call outside the loop, because the inputs to `build_index` remain unchanged across iterations. Thus, optimizing the caller can reduce the number of calls to the hotspot, which can yield a greater speedup than optimizing the hotspot itself. However, simply adding every caller to the optimization list is also ineffective, because this substantially increases the number of functions to optimize. Based on our analysis on our benchmarks, the hotspot functions have on average 2-5 callers, and even if we add the callers of two levels above the hotspot, the number of functions to optimize can be as high as 133 in SWE-fficiency-Lite (Ma et al., 2026). Thus, we not only need to consider the profiling information, but also the optimization opportunities in the functions.

Our solution combines profiling information with *optimization opportunity detection*: lightweight static analysis determining whether an opportunity to significantly reduce execution cost exists based on the hotspot information. For example, a call to the hotspot function inside a loop whose inputs remain unchanged suggests an opportunity to move repeated computation outside the loop. Profiling identifies hotspots and their callers; optimization opportunity detection then filters the callers according to how they invoke expensive work. In this way, we consider the optimization opportunities in the callers with affordable cost.

We introduce CHICO (**C**aller-**H**otspot **I**nvestigation for **C**ode **O**ptimization), a prototype that integrates this localization approach with an existing local optimizer (Figure 2). First, CHICO profiles the performance test to measure execution costs and analyzes call relationships. Second, it identifies hotspot functions and follows their call relationships to identify callers. Third, it filters the callers through optimization opportunity detection. As the first study towards this direction, we consider two types of optimization opportunities: *reuse*, where calls inside a loop have inputs that remain constant across iterations and can be moved outside the loop when their results can be safely reused; and *reorganization*, where call inputs can vary across iterations but reorganizing the loop can reduce the number of calls. We use lightweight static analysis that examines loops and call inputs to identify potential opportunities. The LLM later checks the conditions needed to preserve program behavior before applying a change. Fourth, CHICO ranks candidate functions using both execution costs and the diversity of optimization opportunities. After selecting a candidate with one type of opportunity, it prioritizes candidates with other types. Within each type, the candidates are ranked by execution costs. Here we retain direct hotspot optimization as a type. Finally, it applies an LLM-based local optimizer to optimize the selected functions with prompts of the optimization opportunities. Our implementation uses SemOpt (Zhao et al., 2026), a state-of-the-art local optimization approach, for

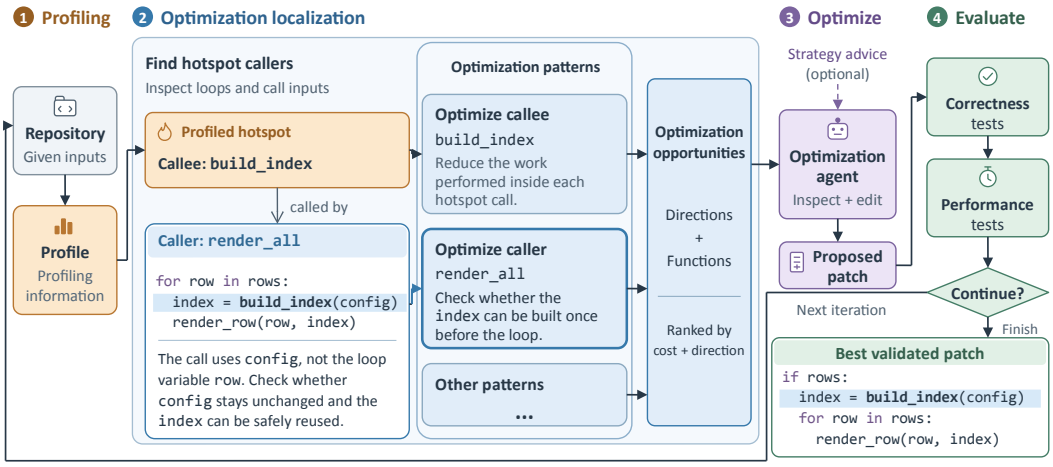

Figure 2: Pipeline of CHICO. *Profiling* records execution costs and caller–callee relationships. *Optimization localization* uses optimization patterns to identify functions and directions to investigate. The example retains direct hotspot optimization and identifies a caller-side reuse opportunity; the *Other patterns* box denotes patterns not expanded in this example. Opportunities are selected using execution cost and optimization direction. In *Optimize*, an LLM optimization agent inspects the code and proposes edits with optional strategy advice. *Evaluate* runs correctness and performance tests, provides feedback for further iterations, and retains the best validated patch.

this step. Correctness tests and performance tests evaluate the resulting changes and guide further iterations.

We evaluate CHICO on the repository-level GSO (Shetty et al., 2025) and SWE-fficiency-Lite (Ma et al., 2026) benchmarks against Codex, PEACE, Optimo, and POLO. Across comparisons using the same model, CHICO improves the rate of producing correct patches that meet each benchmark's speed criterion by 10.29–123.53% on GSO and 12.50–239.13% on SWE-fficiency-Lite. Further experiments show that our localization approach leads to substantial performance gains.

Our work makes three contributions: (1) An optimization localization approach that combines profiling information with optimization opportunity detection to select hotspot functions and callers for optimization. (2) A prototype implementation, CHICO, that integrates this localization approach with SemOpt, a state-of-the-art local optimization approach. (3) A comprehensive evaluation on GSO (Shetty et al., 2025) and SWE-fficiency-Lite (Ma et al., 2026) across four LLMs, comparing CHICO with Codex, PEACE, Optimo, and POLO and using an ablation to assess the contribution of localization.

2 Method

Given a repository, a performance test, and correctness tests, CHICO aims to reduce execution time on the performance test while preserving program behavior. Localization combines profiling information with *optimization opportunity detection*: source-code checks of loops and call inputs that identify callers where repeated calls may be reduced. This allows optimization to consider callers as well as hotspots.

Figure 2 shows four stages. *Profiling* identifies hotspots, functions accounting for substantial execution time. *Optimization localization* selects hotspots and callers and suggests changes. *Optimize* uses a local optimizer based on a large language model (LLM) to generate an *optimization patch*, code changes intended to improve performance. *Evaluate* checks correctness and measures performance to guide the next iteration.

2.1 PROFILING

Our Python implementation profiles the performance test with cProfile (Python Software Foundation, n.d.), recording each function's self time, cumulative time, and caller–callee call counts. Self time excludes time spent in callees; cumulative time includes it. A caller can have little self time but substantial cumulative time when most execution time is spent in its callees. We therefore consider both the function with the highest self time and the function with the highest cumulative time when selecting up to three distinct hotspots for localization. Appendix A gives the selection order and rules for optionally including a target function named in the public task description.

2.2 OPTIMIZATION LOCALIZATION

Localization retains direct optimization of selected hotspots and filters their callers through optimization opportunity detection. Each *optimization opportunity* pairs a candidate function with a suggested change. The three types are direct hotspot optimization; *reuse*, sharing a result across loop iterations to avoid repeated calls; and *reorganization*, reorganizing a loop to reduce calls even when inputs may vary. The following steps detect and select opportunities for Optimize.

Identify hotspot callers. We match profiled caller–callee pairs to source calls using CodeQL (Avgustinov et al., 2016), a static-analysis tool that identifies call expressions and their caller and callee functions. We retain a relationship only if both functions are identifiable in the current source and uniquely matched to a profiling record with a positive call count. Each record gives a total count per function pair, not per source location, and no runtime argument values. We inspect up to two levels: hotspot callers and their callers. We retain paths to hotspots and can search through callers without detected opportunities to reach higher callers. For example, suppose function `A` calls `B` inside a loop, while `B` calls hotspot `H` outside a loop. Searching through `B` lets us investigate `A`: reducing calls to `B` may also reduce calls to `H`. The next step checks `A`'s loop and the inputs of its call to `B`.

Detect optimization opportunities. Changing a caller may avoid calls inside its loops. *Optimization patterns* specify conditions in profiling information or source code for considering an opportunity type. Semgrep, a source-code pattern matcher (Semgrep, n.d.), finds calls in supported loop structures, including `for` and `while` bodies. We retain only matches with the caller–callee relationships established above, then check call inputs to choose between reuse and reorganization.

We check call inputs for direct references to loop variables, whose values may change across iterations. CodeQL identifies variables assigned by an ordinary `for` loop's header and checks call inputs for references to them. Each reference must resolve to the same variable in the same scope. The check covers positional arguments before any unpacked argument such as `*args` and the method-call receiver, the object before the dot. It excludes keyword argument values.

If checked inputs do not directly reference a loop variable, we suggest reuse: checking whether the loop can reuse one computation. In Figures 1 and 2, `build_index(config)` does not directly reference the loop variable `row`, so we suggest checking whether the caller can build the index once before the loop. If a direct reference is found, we suggest reorganization: checking whether the caller can combine several calls' work into fewer calls even when inputs may differ.

These changes are suggestions: direct references do not fully determine how inputs change during execution. The check does not track values through arbitrary assignments, and other loop code may modify `config` even when the call does not reference `row`. If the input check is incomplete, including for loop forms it does not cover, we still suggest reuse and mark unchecked conditions as unknown. Optimize inspects the code to decide whether the suggested change preserves behavior.

Merge and select opportunities. Several calls in one caller may suggest the same opportunity type, even for different hotspots. We merge opportunities with the same function and type, retaining call locations, paths to hotspots, and conditions still to check. Different types at one function remain separate. Table 1 summarizes the three types.

We select opportunities using both their type and execution cost. For a caller's path p to a selected hotspot, we use

$$c(p) = \max_{f \in p} \text{cumulativeTime}(f), \tag{1}$$

Table 1: Optimization opportunity types and checks required before editing.

Type	Function	Basis for investigation	What to check
Direct hotspot optimization	Hotspot	Selected from profiling information	Can work inside the hotspot be reduced?
Reuse	Caller	Call in a supported loop; no direct loop-variable reference found, or input check incomplete	Can the loop reuse one computation or result?
Reorganization	Caller	Call in a supported loop; a direct loop-variable reference found	Can several calls be combined into fewer calls while preserving behavior?

where $\mathrm{cumulativeTime}(f)$ is function f's profiled cumulative time. We take the maximum rather than add times because caller and callee times can overlap. Caller opportunities are ranked by decreasing $c(p)$, with shorter paths breaking ties; merged opportunities use their highest-ranked path. This prioritizes functions associated with more execution time. The benefit of changing them is measured after editing.

Optimize receives at most three opportunities per iteration. To include different types, we first retain direct optimization of the hotspot with the highest self time, then select ranked caller opportunities, at most one per type. Other direct hotspot opportunities fill remaining positions. For example, suppose two reuse opportunities rank above a reorganization opportunity. After retaining direct hotspot optimization, we select the higher-ranked reuse opportunity and the reorganization opportunity. This gives the optimizer both reuse and reorganization opportunities to investigate. A target function named in the public task description is also included if it has current timing information, replacing the last opportunity if necessary. Without caller analysis, we retain direct hotspot opportunities with current profiling information. Algorithm 1 in Appendix A gives the complete procedure.

2.3 Optimize

Optimize uses an adaptation of SemOpt (Zhao et al., 2026) as the local optimizer to generate an optimization patch from selected opportunities. Its *optimization agent* is an LLM with tools to inspect and edit repository code. We pass the selected hotspots and callers to the agent. For callers, we also provide the detected opportunity type (reuse or reorganization), supporting call locations and paths, and conditions still to check. It checks whether suggested changes preserve behavior and can adapt or reject them. Selected functions indicate where to start; the agent determines final edit locations and can change related functions when needed. Optional *strategy advice* describes historical performance changes and their applicability conditions (Appendix B). The agent can use this advice; localization does not use it.

2.4 Evaluate

Evaluation checks whether a patch preserves tested behavior and improves performance. CHICO runs correctness tests and checks that modified code is rebuilt and loaded where needed. It then runs the same performance test on the original and modified programs under the same settings. We time execution separately from profiling, repeating measurements to account for runtime variation (Kalibera & Jones, 2013). Speedup is the original mean time divided by the modified mean time. To avoid selection based on noisy measurements, we reduce speedup according to relative timing variation, yielding a selection score. A patch can be returned only if it passes the checks, timing variation is within the accepted range, and its score exceeds one. Unusually large gains require additional verification. We save the highest-scoring eligible patch as the *best validated patch*.

To repair failed patches without losing earlier improvements, CHICO keeps the best validated patch separately from the program being edited. If code generation completes with a nonempty patch obeying the task's editing restrictions, including unchanged tests, the next iteration continues from that modified program. The agent can then repair failing tests or revise a change that slowed the program. Otherwise, we restore the program obtained from the best validated patch, or the original if none qualifies. The agent receives this code, the latest test and timing results, and the best vali-

dated patch's summary. Localization uses only this code version's profiling information; unavailable information remains unknown.

If recent iterations fail, repeat the same patch, or do not improve the score, CHICO can stop supplying the localization ranking and strategy advice. The agent then uses code, profiling information, and evaluation results to try other changes. This *open exploration* mode persists once entered. The prototype runs up to five iterations and always uses open exploration in the final configured iteration. When optimization ends or execution cannot continue, it returns the best validated patch, or the original program if none qualifies. Appendices C and D give measurement and iteration rules.

3 EXPERIMENTS

We compare CHICO with existing baselines as complete systems for repository-level code performance optimization (Section 3.2), then measure the contribution of our localization with the same local optimizer (Section 3.3).

3.1 EXPERIMENTAL SETUP

3.1.1 TASKS AND BENCHMARKS

Each task asks the optimizer to improve a public workload's performance while preserving program behavior, using the original repository and permitted workload feedback. Edits may span functions or files. We use GSO (Shetty et al., 2025) (102 tasks, 10 repositories) and SWE-fficiency-Lite (Ma et al., 2026) (100 tasks, 9 repositories) for their repository-level performance optimization tasks, executable workloads, correctness tests, and developer-written expert optimization patches for evaluation. We exclude PEACE's PeacExec because public materials are insufficient to reconstruct the complete benchmark (Ren et al., 2025), and Optimo's COFFE and EffiBench because they cover individual functions or standalone programs rather than repository-level tasks (Peng et al., 2026).

3.1.2 BASELINES AND ADAPTATIONS

The four baselines are Codex, PEACE, Optimo, and POLO. PEACE and Optimo receive initial target functions under two settings: Profile for fair baseline comparison, and Reference using edit locations identified from expert optimization patches. Other methods lack this extra location information, so Reference results are analyzed separately and excluded from the main baseline comparison.

Codex (OpenAI, n.d.) receives access to the repository and the optimization task, then uses the selected LLM to inspect and edit code. The agent chooses edit locations and returns an optimization patch.

PEACE (Ren et al., 2025) uses dependencies and code similarity to select the target function's related callers and callees. Historical edits from the same repository and external efficient code examples guide optimization. Its optimizer weights were not publicly available, so we fine-tuned Llama-3-8B (AI@Meta, 2024) with the authors' released data and reported setup.

Optimo (Peng et al., 2026) optimizes the supplied target function at the algorithm, statement, expression, and application programming interface (API) levels using its released prompts and strategies.

POLO (Bai et al., 2025) ranks functions by their own and their callees' execution costs. It proposes edits, using correctness and runtime feedback to guide further edits or stop. With no public implementation available for our experiments, we reimplemented its workflow from the paper. The original study evaluates C/C++ projects; for our Python benchmarks, we use py-spy call-stack sampling (Frederickson, n.d.) and Python source analysis.

Profile uses py-spy on the original public workload to select the repository function with the largest share of attributed execution samples (Profile-Top1; Appendix E.1). PEACE and Optimo receive this function without reference information. Optimo also receives line weights, each line's share of the selected function's samples, to estimate where time is spent in place of profiles comparing small and large inputs.

Reference uses an LLM to analyze the expert optimization patch offline, identify changed functions, and select the one containing the main performance change. We map it to the original code independently of optimizer results. Both optimizers receive its name and original code, but not the expert patch. Appendix E.1 gives implementation and mapping details.

3.1.3 MODELS AND BUDGETS

We use GPT-5.6 Sol, Terra, and Luna to cover flagship, balanced, and cost-efficient model tiers (OpenAI, 2026). We also include the open-weight DeepSeek V4.1 Flash, whose reported DeepSWE v1.1 and Terminal-Bench 2.1 scores exceed those of V4 Pro (DeepSeek-AI, 2026, Table 3). The main comparison covers all four models; Reference comparisons use the three GPT models, and the localization comparison uses Terra.

GPT models run through Codex (OpenAI, n.d.) at high reasoning effort. The Codex baseline also runs DeepSeek through Codex; all other DeepSeek configurations use its API with default settings. Each Codex invocation, including tool execution, and each DeepSeek API request has a 30-minute timeout. Appendix E.4 presents the model cost analysis.

3.1.4 EVALUATION AND METRICS

We independently evaluate each run's single returned patch for correctness and runtime with the benchmarks' task-specific Docker environments and harnesses. An unchanged program may pass correctness but provides no acceleration. Workloads run on a server with two AMD EPYC 9655 processors (192 physical cores, 384 hardware threads in total), 512 GB DDR5 memory, and Ubuntu 22.04.5 LTS.

On GSO, **Correctness** is the % of tasks passing all required correctness tests. Here @1 means returning one final optimization result per task. We additionally report **Sp@1**, which requires correctness and at least $1.2\times$ speedup over the original program. **Opt@1** follows GSO's published definition: correctness and at least $0.95\times$ the speed achieved by the expert patch (Shetty et al., 2025). Both aggregate runtime ratios harmonically across each task's tests before applying these thresholds (Appendix E.3). All three percentages use all 102 tasks.

On SWE-fficiency-Lite, speedup is the original runtime divided by the patched runtime. **Passes Tests** is the % of tasks passing all required correctness tests; **≥Expert** also requires at least the expert patch's speedup. Both percentages use all 100 tasks. **Speedup Ratio (SR)** is the harmonic mean across all 100 tasks of the method's speedup divided by the expert patch's speedup, with each ratio floored at 0.001 (Ma et al., 2026). SR $= 1$ means matching expert performance under this aggregation.

On GSO, missing required runtimes contribute no success to the corresponding performance metric. On SWE-fficiency-Lite, empty or incorrect patches and missing performance results receive the original program's speedup of 1, without granting correctness credit. All configurations follow these rules with fixed task denominators (Appendix E.3).

3.2 OVERALL COMPARISON

CHICO outperforms Codex, PEACE-Profile, Optimo-Profile, and POLO on optimization metrics for all four evaluated models (Table 2). Across comparisons with the same model, relative gains are 10.29–123.53% in GSO Sp@1 and 32.50–237.50% in Opt@1, and 12.50–239.13% in SWE-fficiency-Lite ≥Expert and 35.71–845.24% in SR. Thus, the full system produces more patches passing correctness tests and meeting performance thresholds and improves aggregate speed relative to expert patches.

Optimization with edit locations from expert patches. Under the Reference settings in Section 3.1, PEACE and Optimo improve over Profile by 48.28–93.75% in GSO Opt@1 and 67.74–156.52% in SWE-fficiency-Lite ≥Expert across the three GPT models. These gains support the importance of accurate localization: functions identified from expert patches can provide more effective starting points than the highest-cost function alone, improving optimization results.

Test pass rates and performance gains. Higher test pass rates need not mean better optimization. With Sol, CHICO has lower GSO Correctness than Codex (96.08% versus 98.04%) but nearly dou-

Table 2: **System comparison on GSO and SWE-fficiency-Lite.** Model interfaces and metrics follow Section 3.1; Corr. denotes Correctness. Rates are in %, SR in ×. Reference (†) uses edit locations identified from expert patches. Other methods lack this information, so its results are analyzed separately and excluded from the main baseline comparison, ranking, bolding, and Δ. Within each model and benchmark, bold marks the best score per metric, including ties; Δ is CHICO minus the best baseline, using displayed scores. Green and orange indicate positive and negative Δ, respectively. Darker shades mark larger absolute differences across models for the same benchmark and metric. —: not evaluated.

System	*GPT-5.6 Sol / Codex / high*			*GPT-5.6 Terra / Codex / high*			*GPT-5.6 Luna / Codex / high*			*DeepSeek V4.1 Flash*		
GSO	Corr.	Sp@1	Opt@1	Corr.	Sp@1	Opt@1	Corr.	Sp@1	Opt@1	Corr.	Sp@1	Opt@1
Codex	**98.04**	65.69	32.35	90.20	50.00	23.53	**94.12**	58.82	21.57	76.47	63.73	28.43
PEACE-Profile	87.25	55.88	28.43	81.37	38.24	18.63	80.39	48.04	17.65	82.35	45.10	23.53
PEACE-Reference†	90.20	70.59	42.16	88.24	58.82	33.33	85.29	63.73	32.35	—	—	—
Optimo-Profile	92.16	53.92	26.47	90.20	33.33	15.69	89.22	39.22	14.71	88.24	37.25	16.67
Optimo-Reference†	95.10	66.67	43.14	94.12	55.88	30.39	92.16	59.80	28.43	—	—	—
POLO	94.12	71.57	46.08	92.16	60.78	37.25	91.18	64.71	33.33	86.27	66.67	39.22
CHICO	96.08	**81.37**	**63.73**	**95.10**	**74.51**	**52.94**	**94.12**	**71.57**	**47.06**	**92.16**	**73.53**	**51.96**
Δ	−1.96	+9.80	+17.65	+2.94	+13.73	+15.69	0.00	+6.86	+13.73	+3.92	+6.86	+12.74

SWE-fficiency-Lite	Passes Tests	≥ Expert	SR	Passes Tests	≥ Expert	SR	Passes Tests	≥ Expert	SR	Passes Tests	≥ Expert	SR
Codex	83	50	0.088	77	38	0.059	76	34	0.058	81	48	0.125
PEACE-Profile	79	35	0.053	77	31	0.024	70	19	0.029	77	34	0.059
PEACE-Reference†	84	61	0.185	84	52	0.089	75	38	0.066	—	—	—
Optimo-Profile	**94**	23	0.038	**87**	24	0.025	**84**	21	0.020	**92**	39	0.042
Optimo-Reference†	92	59	0.164	91	48	0.069	88	43	0.065	—	—	—
POLO	90	66	0.234	83	56	0.154	82	50	0.123	84	65	0.194
CHICO	90	**78**	**0.337**	86	**63**	**0.209**	**84**	**62**	**0.186**	**92**	**79**	**0.397**
Δ	−4	+12	+0.103	−1	+7	+0.055	0	+12	+0.063	0	+14	+0.203

bles Opt@1 (63.73% versus 32.35%). On SWE-fficiency-Lite with the same model, it has lower Passes Tests than Optimo-Profile (90% versus 94%) but much higher ≥Expert (78% versus 23%). Passing tests alone is therefore insufficient; patches must also improve performance.

Across models, CHICO achieves SWE-fficiency-Lite ≥Expert of 62–79%, while SR is 0.186–0.397. ≥Expert counts tasks reaching expert speed; SR aggregates all tasks' speedup ratios relative to the expert patch. A few small ratios, including those from scoring fallbacks to the original program, can lower SR even when many tasks reach expert speed (Appendix E.3).

Model cost and optimization performance. CHICO costs more per task than Codex and Optimo-Profile but less than POLO, while outperforming all four baselines on every optimization metric in each evaluated model–benchmark configuration (Appendix E.4).

3.3 Contribution of Localization

We compare CHICO with *w/o localization* to measure the contribution of our localization (Table 3). This variant replaces CHICO's candidate functions and suggested changes with Profile-Top1, the highest-ranked function from profiling the original program. It is the same input used for PEACE and Optimo in the main baseline comparison (Section 3.1). Both configurations retain the same local optimizer, strategy library, behavioral validation, and control flow.

Replacing Profile-Top1 with our localization improves all four optimization metrics. Relative gains are 26.67% in GSO Sp@1 and 54.29% in Opt@1, and 8.62% in SWE-fficiency-Lite ≥Expert and 37.50% in SR. With the local optimizer held fixed, these gains show that our localization helps it produce more effective optimization patches than selecting the highest-cost function alone.

Table 3: **Contribution of localization with the same local optimizer.** The full-method SWE results are reused from Table 2. Metrics and units follow Table 2; bold marks the best score in each column, including ties.

	GSO			SWE-fficiency-Lite		
Variant	Corr.	Sp@1	Opt@1	Passes Tests	≥Expert	SR
CHICO	**95.10**	**74.51**	**52.94**	**86**	**63**	**0.209**
w/o localization	93.14	58.82	34.31	**86**	58	0.152

4 Related Work

For years, automated code optimization has been done using rule-based, analytical, and search-based approaches such as compiler optimizations (Bacon et al., 1994), memoization (Della Toffola et al., 2015), and genetic optimization (Giavrimis et al., 2021). Rule-based approaches require manually defining optimization rules and are therefore limited to a fixed set of optimization patterns and are hard to extend across languages.

Deep learning approaches improved on this. DeepDev-PERF (Garg et al., 2022) fine-tunes a model on C# source code, while Supersonic (Chen et al., 2024) uses a seq2seq model to optimize C/C++ code. However, these approaches were quickly overshadowed by the introduction of LLMs, which have extensive knowledge of multiple programming languages and remove the need for training a separate model. RAPGen (Garg et al., 2025) achieved better results than DeepDev-PERF by zero-shot prompting. SBLLM (Gao et al., 2025) instead iteratively refines LLM results using a search-based technique. EffiLearner (Huang et al., 2024) uses light-weight profiling to identify hotspots in generated code and feeds the results back to the LLM for refinement. These prompting, search, and feedback techniques are orthogonal to our edit-location selection and can be combined with it to optimize the selected code. SemOpt (Zhao et al., 2026) uses profiling to identify hotspots in repositories and a strategy library to optimize them one function at a time, whereas our method also investigates their callers as alternative edit locations. Optimo (Peng et al., 2026) uses differential profiling to optimize supplied functions or standalone programs at multiple levels, but does not autonomously select the initial target across a repository.

Recent work has also explored project-level optimization. PEACE (Ren et al., 2025) requires a prespecified target function to start optimization and therefore lacks autonomous localization of the initial target within a repository. Profiling-based methods instead select targets from execution costs: POLO (Bai et al., 2025) propagates cost through call relationships to rank hotspot functions. However, cost alone cannot determine whether reorganizing the code at a given location would actually improve performance without changing its behavior. CHICO addresses the edit-location decision without requiring a given target, by combining dynamic and static analysis to examine whether to optimize a hotspot itself or reduce repeated work at its caller, considering callers with little execution cost of their own.

Finally, EffiHolmes (Yang et al., 2026) addresses performance issue localization. This approach takes a performance issue reported by the user, and localizes the code location related to this issue. On the contrary, our approach finds the location to optimize based on a performance test.

5 Conclusion

Hotspots show where execution time accumulates, but callers can offer more useful edit locations by controlling how often that work is requested. CHICO combines execution costs with source analysis of calls, loops, and call inputs to guide repository-level optimization toward hotspot and caller candidates. Across GSO and SWE-fficiency-Lite, the complete system consistently outperforms four baselines on optimization performance. Further experiments show that our localization approach leads to substantial performance gains. These results support profiling as the starting point for selecting edit locations rather than the final answer.

AI USE STATEMENT

We used ChatGPT and Codex to assist with literature discovery, discussions of the research problem and experimental design, and manuscript revision. Our experiments also use GPT-5.6 and DeepSeek to extract historical optimization strategies, generate associated detection rules, and generate candidate code edits. These methodological uses are described in the Method, Experiments, and Appendix sections. The authors take responsibility for the final content of this work, including its claims, references, code, and figures.

REPRODUCIBILITY STATEMENT

The artifact page is available at https://figshare.com/s/1d0fefd1541747db6fc7. Sections 2–3 and Appendices A–E describe our method, implementation details, and evaluation protocol.

REFERENCES

AI@Meta. Llama 3 model card. Official model card, 2024. URL https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.

Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pp. 483–485. Association for Computing Machinery, 1967. doi: 10.1145/1465482.1465560. URL https://doi.org/10.1145/1465482.1465560.

Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: Object-oriented Queries on Relational Data. In Shriram Krishnamurthi and Benjamin S. Lerner (eds.), *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 2:1–2:25, Dagstuhl, Germany, 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-014-9. doi: 10.4230/LIPIcs.ECOOP.2016.2. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2016.2.

David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994. ISSN 1557-7341. doi: 10.1145/197405.197406. URL https://doi.org/10.1145/197405.197406.

Jiameng Bai, Ruoyi Xu, Sai Wu, Dingyu Yang, Junbo Zhao, and Gang Chen. POLO: An LLM-powered project-level code performance optimization framework. In *International Joint Conference on Artificial Intelligence*, pp. 7319–7328, 2025. doi: 10.24963/ijcai.2025/814. URL https://www.ijcai.org/proceedings/2025/814.

Zimin Chen, Sen Fang, and Martin Monperrus. Supersonic: Learning to generate source code optimizations in C/C++. *IEEE Transactions on Software Engineering*, 50(11):2849–2864, 2024. doi: 10.1109/TSE.2024.3423769. URL https://doi.org/10.1109/TSE.2024.3423769.

DeepSeek-AI. DeepSeek-V4.1-Flash: Pushing the limits of KV cache compression. Technical report, 2026. URL https://huggingface.co/deepseek-ai/DeepSeek-V4.1-Flash/blob/main/DeepSeek_V41_Tech_Report.pdf.

Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 607–622, 2015. doi: 10.1145/2814270.2814290. URL https://www.research-collection.ethz.ch/handle/20.500.11850/111423.

Ben Frederickson. py-spy: Sampling profiler for Python programs. Software and official repository documentation, n.d. URL https://github.com/benfred/py-spy. Undated repository documentation; accessed September 22, 2026.

Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael R. Lyu. Search-based LLMs for code optimization. In *IEEE/ACM International Conference on Software Engineering*, 2025. doi: 10.1109/ICSE55347.2025.00021. URL https://conf.researchr.org/details/icse-2025/icse-2025-research-track/27/Search-Based-LLMs-for-Code-Optimization.

Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B. Clement, Neel Sundaresan, and Chen Wu. DeepDev-PERF: A deep learning-based approach for improving software performance. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 948–958, 2022. doi: 10.1145/3540250.3549096. URL https://doi.org/10.1145/3540250.3549096.

Spandan Garg, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. RAPGen: An approach for fixing code inefficiencies in zero-shot. In *IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 124–135, 2025. doi: 10.1109/ICSE-SEIP66354.2025.00017. URL https://doi.org/10.1109/ICSE-SEIP66354.2025.00017.

Rafail Giavrimis, Alexis Butler, Constantin Cezar Petrescu, Michail Basios, and Santanu Kumar Dash. Genetic optimisation of C++ applications. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1180–1182. IEEE, 2021. doi: 10.1109/ASE51524.2021.9678650. URL https://doi.org/10.1109/ASE51524.2021.9678650.

GitHub Software UK Ltd. Call. CodeQL Python standard library reference, n.d. URL https://codeql.github.com/codeql-standard-libraries/python/semmle/python/Exprs.qll/type.Exprs$Call.html. Accessed September 22, 2026.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

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. In *Advances in Neural Information Processing Systems*, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/99c66755871ae101a4cef87c67fb29e8-Abstract-Conference.html.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.06770v3.

Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In *ACM International Symposium on Memory Management*, pp. 63–74, 2013. doi: 10.1145/2464157.2464160. URL https://kar.kent.ac.uk/33611/.

Jeffrey Jian Ma, Milad Hashemi, Amir Yazdanbakhsh, Kevin Swersky, Ofir Press, Enhui Li, Vijay Janapa Reddi, and Parthasarathy Ranganathan. SWE-fficiency: Can language models optimize real-world repositories on real workloads? In *International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2511.06090v3.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition. Official model release, July 9, 2026. URL https://openai.com/index/gpt-5-6/. Accessed September 22, 2026.

OpenAI. Codex CLI. Software and official repository documentation, n.d. URL https://github.com/openai/codex. Undated repository documentation; accessed September 22, 2026.

Yun Peng, Akhilesh Deepak Gotmare, Michael R. Lyu, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. PerfCodeGen: Improving performance of LLM generated code with execution feedback. In *IEEE/ACM 2nd International Conference on AI Foundation Models and Software Engineering (FORGE)*, pp. 1–13, 2025. doi: 10.1109/Forge66646.2025.00008. URL https://doi.org/10.1109/Forge66646.2025.00008.

Yun Peng, Jun Wan, Jiakun Liu, Shuzheng Gao, David Lo, and Xiaoxue Ren. Multi-level code optimization via mixture of prompts. arXiv:2607.23665v1, 2026. URL https://arxiv.org/abs/2607.23665v1. Accepted to ASE 2026.

Python Software Foundation. The Python profilers. Python documentation, n.d. URL https://docs.python.org/3/library/profile.html. Accessed September 22, 2026.

Xiaoxue Ren, Jun Wan, Yun Peng, Zhongxin Liu, Ming Liang, Dajun Chen, Wei Jiang, and Yong Li. PEACE: Towards efficient project-level efficiency optimization via hybrid code editing. In *IEEE/ACM International Conference on Automated Software Engineering*, pp. 1831–1843, 2025. doi: 10.1109/ASE63991.2025.00153. URL https://doi.org/10.1109/ASE63991.2025.00153.

Semgrep. Rule pattern syntax. Semgrep documentation, n.d. URL https://docs.semgrep.dev/writing-rules/pattern-syntax. Accessed September 22, 2026.

Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. GSO: Challenging software optimization tasks for evaluating SWE-agents. In *Advances in Neural Information Processing Systems*, 2025. URL https://papers.nips.cc/paper_files/paper/2025/hash/ddd98389004161943e646322a16348c3-Abstract-Datasets_and_Benchmarks_Track.html. Datasets and Benchmarks Track.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/ce65173b994cf7c925c71b482ee14a8d-Abstract-Conference.html.

Haowen Yang, Yun Peng, and Zishuo Ding. EffiHolmes: Differential profiling-guided repository level time inefficiency fix localization. arXiv:2608.03558v1, 2026. URL https://arxiv.org/abs/2608.03558v1. Accepted to ASE 2026.

Yuwei Zhao, Yuan-An Xiao, Qianyu Xiao, Zhao Zhang, and Yingfei Xiong. SemOpt: LLM-driven code optimization via rule-based analysis. *ACM Transactions on Software Engineering and Methodology*, 2026. doi: 10.1145/3820167. URL https://doi.org/10.1145/3820167.

A LOCALIZATION IMPLEMENTATION DETAILS

This appendix specifies the Python localization path used by CHICO. The hotspots selected in Section 2.1 serve as the *hotspot seeds* from which caller search begins. Table 4 lists the implementation limits for collecting source evidence, searching caller relationships, and presenting optimization opportunities. A function can participate in more than one function–hotspot search state, and several opportunities can refer to the same function with different opportunity types.

Table 4: Default localization limits and their units.

Stage	Limit	What is bounded
Hotspot initialization	3 seeds	Distinct functions selected from current timing evidence.
Source investigation	48 functions; 100 files	Scheduled investigation, including relevant files and resolvable one-level imports.
Upstream search	2 edges; 48 states	Caller distance and shared function–hotspot states across all seeds.
Opportunity selection	3 entries	Hotspot and caller opportunities in one optimization session.

A.1 SOURCE SCOPE AND EVIDENCE MATCHING

Source investigation prioritizes hotspot functions and observed callers using a deterministic order. Its function limit controls which functions are scheduled for investigation; parsing a selected file may expose other declarations. The subsequent caller search uses only relationships supported by both the static analysis and the current profile. Resolving an additional declaration does not, by itself, establish that it was executed by the workload.

Hotspot initialization first considers the highest-self-time function, the highest-cumulative-time function, and the first eligible public task target with current timing evidence, when supplied. The ordered list is deduplicated, then filled from the self-time ranking up to three distinct seeds. The target is supplied by the public task description, never derived from a reference patch. The supplied workload is sufficient; initialization does not require a small-input/large-input pair.

Source matches are assigned to the innermost enclosing function. A syntax match can be joined to a call using its file and starting position only when that pair uniquely identifies a complete call expression. Input-reference facts must agree with the complete source span. Static resolution uses CodeQL function objects, supplemented by lexical resolution of `self` and `cls` calls within their class. Dynamic endpoints must resolve uniquely to the same current functions, with a unique function-pair record and a positive call count. There is no additional threshold such as requiring 100 calls.

Dynamic counts remain attached to function pairs. For example, if `A` calls `B` at two source locations and the profile records 100 calls from `A` to `B`, the analysis retains both locations and the aggregate count. It does not assign 100 calls to each location or infer the argument values at either one. Similarly, a retained path consists of supported function-pair edges; the profile need not contain an observation of that entire path as one call stack.

A.2 LOOP PATTERNS AND INPUT REFERENCES

The fixed structural inventory contains seven patterns: calls in `for` bodies, calls in `while` bodies, result calls in list comprehensions, generator expressions, and set comprehensions, and result calls in filtered list comprehensions and filtered generator expressions. These patterns inspect specific structures. Dictionary comprehensions, filtered set comprehensions, and multiple-generator comprehensions are outside this inventory.

The input query handles ordinary `for` bindings. It resolves the variables defined by the iteration target, including unpacked targets, and examines variable uses in ordinary positional arguments before a starred argument and in the receiver of an attribute call. Positional arguments are enumerated through CodeQL's `Call.getArg` interface, which excludes arguments preceded by a starred argument (GitHub Software UK Ltd., n.d.). A positive result requires a use and an iteration-target

definition to resolve to the same variable in the same scope. Thus, both `process(row.key)` and `row.process(config)` reference the binding in `for row in rows`. Keyword argument values are not enumerated by this query.

This is a direct-reference check. In `alias = row; process(alias)`, the call has no direct reference to `row`; the query does not trace the intervening assignment. Conversely, `row = fixed; process(row)` still references that binding, even though the assignment may change its value. These cases explain why a detected opportunity is an investigation suggestion requiring further code inspection before editing.

For reuse, the optimization agent inspects assignments, input and state changes between calls, side effects, and whether iterations can share a result. Moving calls outside loops must also preserve exception order and behavior when a loop executes zero times. Reorganization must preserve each item's required work and ordering. The agent can adapt or reject suggestions after these checks.

Nested loops are checked separately, and a positive binding reference associated with any enclosing checked loop supplies positive evidence. A call in a separately defined nested function or lambda does not inherit the outer function's binding evidence. There is no corresponding binding query for `while`, nor a dedicated enumeration of comprehension generator bindings. A comprehension may also execute in a separate runtime frame; its observation is not automatically converted into an edge from the lexical owner to the callee. A syntax match therefore produces a caller opportunity only if the required call relationship is established. If that relationship is established but input evidence is incomplete, the caller remains eligible for reuse investigation, with unresolved input conditions.

A.3 Search, merging, and final selection

Each search state pairs a function with a target hotspot and retains one path discovered by deterministic bounded traversal. Cycles are avoided. The search can pass through a function whose outgoing call does not match a loop pattern. For example, along $A \to B \to H$, a loop-related call from A to B can motivate investigating A even when B's call to hotspot H is outside a loop. The input check concerns the call from A to B, without propagating arbitrary values through B.

Caller opportunities are grouped by caller function and opportunity type. The group preserves the individual source locations, paths to hotspots, function-pair counts, and required checks. Its priority comes from its highest-ranked retained path: decreasing maximum cumulative time along the path, then increasing path length, then deterministic tie breaking. Missing cumulative time contributes zero. Cumulative caller and callee times are not added, and counts from different function pairs are not collapsed into a per-location count.

The final allocation preserves the direct hotspot opportunity for the highest-self-time hotspot, then selects at most one caller opportunity per opportunity type in ranked order, and fills remaining slots with other direct hotspot opportunities. The three-entry limit applies across all seeds. If the public task target included during initialization is absent from the selected functions, its direct hotspot opportunity is appended when space remains or replaces the last entry otherwise. This rule preserves the first direct hotspot opportunity but can replace the last caller opportunity.

Algorithm 1 collects the localization steps described in Section 2.

Algorithm 1: Construct and select optimization opportunities.

```
Input: current source, matching profile, static-analysis results
Output: at most three optimization opportunities

1. Select up to three distinct hotspot seeds from the profile.
2. Create one direct hotspot opportunity for each seed.
3. Join source-resolved calls with unique positive dynamic function
    pairs.
4. Search at most two caller edges from the seeds, retaining at most
   48 (function, target hotspot) states and avoiding cycles.
5. For each supported loop-related call that starts a retained path:
     inspect available direct loop-variable references;
     choose reorganization if a reference is found,
       otherwise choose reuse;
     attach the source evidence, hotspot paths, and required checks.
6. Merge caller opportunities with the same function and opportunity
    type.
7. Rank caller opportunities by the maximum function cumulative time on
   their highest-ranked retained path, then by that path's edge count.
8. Retain the first direct hotspot opportunity.
9. Add ranked caller opportunities, at most one per opportunity type,
   while space remains.
10. Fill remaining positions with other direct hotspot opportunities.
11. Include the observed public target, keeping the three-entry limit.
```

B Optional Strategy Library

The strategy library provides optional transformation advice to the downstream optimization agent. Historical strategy extraction and rule-based matching follow SemOpt's strategy-guided design (Zhao et al., 2026), but the source collection is broader. SemOpt represents historical strategies with single-function performance changes, whereas our collection also retains commits whose supported optimization requires main and companion edits across related functions. The following details describe the strategy library used by our SemOpt-based optimization agent; the localization algorithm does not depend on the library.

B.1 Extracting and grouping historical changes

An LLM reads a performance commit's before/after code and relevant functions, then separates independently supported transformations into records. The extraction permits at most eight records per commit and asks for one or two when sufficient. Each record describes the operation before the change, the work removed or reorganized, the main and companion changes, and the conditions to check. Source excerpts must correspond to the actual before/after functions. For example, introducing a reusable index and updating its callers constitute one transformation when the caller changes are needed to use that index; an unrelated algorithm replacement in the same commit can form a separate record. The current cross-function collection requires at least two production functions in either the before or after version of a retained record. Here, production functions are parsed Python or C/C++ functions outside directories named test, tests, doc, docs, benchmark, benchmarks, examples, or generated.

The LLM selects a transformation type from a predefined set: batching independent calls, reusing a precomputed result, hoisting invariant work, building a lookup index, specializing a known case, stopping at a requested limit, prefiltering candidates, vectorizing a scalar loop, fusing repeated passes, streaming materialization (`stream_materialization` in the extraction schema), removing a redundant reset, changing an algorithm, inlining an existing kernel, dispatching a specialized kernel, or other. These types organize historical records; they are distinct from the opportunity types produced by localization.

Records are compared only within the same language pool and transformation type. The comparison uses four feature sets: words in the before-operation description, a relation marker, words in the

conditions, and words describing removed work. Word features are lowercase English alphabetic tokens of at least three characters. The relation marker currently states that the record is declared to involve multiple functions; it does not encode a verified historical call or data-flow graph. Missing before-operation or relation features prevent merging.

For nonempty feature sets X and Y, the Jaccard distance is $1 - |X \cap Y|/|X \cup Y|$. As a separate missing-feature rule, either feature set being empty gives distance one for that component, including when both are empty. The four component weights are respectively $0.4, 0.3, 0.2, 0.1$. Complete-linkage clustering merges groups at a distance threshold of 0.35, using the maximum cross-group pair distance. This prevents two dissimilar records from joining solely through a shared intermediate record. Singletons are retained. Within each group, the shortest available complete before-function is selected as the representative. A representative before-function must fit within 24,000 bytes, and its before/after pair within 48,000 bytes.

B.2 Generating and checking strategy-matching rules

The generation input contains the representative before-function, its available after counterpart, the transformation description, related-function identities, and a bounded set of other member summaries. Each of three generation rounds asks for one primary strategy-matching rule (a Semgrep rule matching a relevant source operation), strategy advice, conditions requiring inspection, and a relation plan listing additional relationships to check. The plan can request calls, argument-to-parameter correspondences, local value flow, short value flow, or shared-state checks. In this adapter, the local-flow and short-value-flow requests use the same check for flow from a call result to a positional argument of another call within the same function. These requests are later handled by shared analyses; the model does not generate a new CodeQL query for each strategy.

A generated strategy-matching rule must parse, use the appropriate language, and identify a concrete operation. Its quoted source anchor must occur in the representative before-function. The rule is then executed on that complete function, and a match must correspond to the quoted operation rather than an unrelated location in the same file. Each round allows an initial generation and up to seven repairs using validation feedback. Distinct valid matching structures are retained, allowing one strategy to have several rules. If generation identifies incompatible transformations in a multi-record group, the group can be split into singleton children for generation. Validation establishes that the rule finds its representative operation; it does not require zero matches after the historical change or certify safe transfer to a new program.

The frozen library contains 433 Python strategies with 798 strategy-matching rules and 252 C/C++ strategies with 453 rules. Respectively, 164/173/96 Python strategies and 102/99/51 C/C++ strategies have one/two/three rules. Each released strategy has one source commit. Two Python strategies and one C/C++ strategy contain two records from their respective commit; all remaining strategies contain one record. These are library contents, not measurements of optimization effectiveness. Availability of C/C++ advice does not establish build and execution support for every native repository.

B.3 Matching advice to current code

The adapter starts from the selected functions and additional observed hotspot seeds. It scans a bidirectional neighborhood of at most two call edges and 48 functions, using available static or dynamic call relationships. This broader scan scope differs from the upstream localization search, which requires both forms of evidence. Historical sources excluded by the task configuration are filtered before eligible strategy-matching rules are run for the applicable language.

Matches are bound to concrete source operations. For each strategy–operation pair, the adapter counts distinct supporting rules and retains at most 20 pairs for relationship checks. It first considers one location per strategy and then fills the remaining positions with additional locations. Shared CodeQL and syntax analyses check the requested calls, argument-to-parameter correspondences, and available local value flow. A general shared-state check is unavailable, so such a requirement remains unresolved. Strategy-matching rules and their relation plans support inspection; they do not establish all semantic conditions of the proposed transformation.

Pairs are ranked lexicographically by the number of supporting rules, whether all requested relations are supported, and observed cost; later criteria break ties in earlier criteria. At most four strategies are selected, with at most two locations each. A selected strategy is then retained only if a displayed location belongs to the initial investigation set or its primary location has the requested relation support. Removing a strategy at this stage does not trigger backfilling. The optimization agent receives the historical transformation, concrete current operations, relation evidence, and unresolved conditions in the same optimization session. It may adapt or reject the advice. The library remains frozen throughout that session.

C Behavioral Validation and Performance Measurement

C.1 Checks and paired timing

The controller checks each complete patch using the task's available correctness tests. Changes to compiled native code, such as C/C++ extensions, additionally require a supported rebuild and confirmation that the rebuilt code is loaded. A failed check or an inconclusive execution environment cannot establish a validated improvement. These checks assess behavior exercised by the supplied tests and workload; they do not prove equivalence for arbitrary inputs.

Performance measurement is separate from profiling (Python Software Foundation, n.d.). Execution times can vary across runs, motivating repeated measurements and reporting of variability (Kalibera & Jones, 2013). In the Python paired collectors, each timing observation runs in a fresh process after one untimed workload warmup. Groups alternate the order of the original and changed-program executions. The original and changed programs use matching workload and execution settings, and setup, warmup, and profiling are excluded from the timed interval. The Python SWE path uses five paired groups; the collectors accept a configured count from five through twenty. Reported means and sample standard deviations are computed separately for the two versions. Native measurements must also satisfy the applicable build, loading, and measurement protocol. Measurements must correspond to the evaluated source and patch before they can be used to select the best validated patch.

C.2 Local selection score

Let μ_0, μ_1 be the original and changed-program mean times and s_0, s_1 their sample standard deviations. The controller computes

$$r = \frac{\mu_0}{\mu_1}, \qquad u = \sqrt{\left(\frac{s_0}{\mu_0}\right)^2 + \left(\frac{s_1}{\mu_1}\right)^2}, \qquad q = r \max(0, 1 - 2u). \tag{2}$$

The means must be finite and positive, and the deviations finite and nonnegative. The quantity r is the measured speedup; u combines relative variability; q penalizes a speedup accompanied by large variability. This is a deterministic selection heuristic, not a confidence interval or a significance test.

Ordinarily, a measurement is stable when $u \leq 0.2$ and $r < 50$. A noisier result is ineligible to update the best validated patch. A result with $r \geq 50$ is marked for additional review rather than automatically accepted. Under the paired measurement protocol, its stable status can be restored only after an explicitly recorded audit of the same patch using permitted task feedback and a separate repeated validation whose measurement records are matched to that patch and protocol, with successful behavioral checks and a stable repeated score greater than one. Both measurements must pass the variability check when the large-gain flag is removed. The resulting score is the minimum of the two scores; the recorded speedup is their minimum speedup and the recorded variability their maximum variability. Without that review, the flagged result remains ineligible.

A patch can update the best validated patch only if it contains an actual permitted change, passes the applicable behavioral checks, has comparable stable measurements, and has $q > 1$. Eligible patches are ranked by q, retaining the earlier patch in an exact tie. This local decision governs search and the returned patch. The benchmark evaluation and comparisons in Section 3 are separate; their results are not inferred from this score.

D Optimization Controller

The controller gives the optimization agent at most five optimization sessions, stopping earlier if execution cannot continue. Each iteration in Section 2 consists of an optimization session followed by evaluation. A session lets the agent inspect the current program and return a patch, potentially through multiple model requests and related edits. The selected optimization opportunities are alternatives within that session, rather than separate model calls. Optional strategy advice enters the same session.

D.1 Current program and best validated patch

The *current program* is the source version from which the next attempt starts, initially the original program. Each session edits this version and may retain, revise, or remove earlier changes. The controller saves the resulting program as a complete patch relative to the original snapshot, together with its validation results. Applying that saved patch to the original reconstructs the edited program, including any retained changes from earlier sessions.

After each evaluation, the controller selects the *best validated patch* from all saved attempts using Appendix C. Eligibility requires permitted changes, successful behavioral checks, comparable stable measurements, and a score greater than one. The highest-scoring eligible patch is retained, with the earlier patch winning an exact tie. A later failed, inconclusive, or lower-scoring attempt therefore leaves the best unchanged. This saved best is the final output when search ends; if no patch qualifies, the original program is returned.

By default, the latest normally returned, permitted nonempty patch becomes the next current program, even if its tests fail, its timing is inconclusive, or its performance regresses. Normal return requires a completed agent invocation without a timeout or generation-service failure. Permitted changes exclude protected tests, evidence files supplied by the execution harness, and other forbidden paths. If these continuation conditions are not met, the controller restores the program obtained from the best validated patch, or the original if none exists. This lets the agent repair or revise an unfinished attempt while the best validated patch remains available for final delivery.

D.2 Feedback and source evidence

Each fresh session receives the current source and explicit, bounded feedback. This feedback identifies the source round and whether it is the best validated version, summarizes the latest attempt's checks and timing, and identifies the best validated patch by its originating round and patch identifier, with its recorded optimization decision when available. The agent thus receives information about the best attempt during search. This summary does not automatically load the best patch's source: the continuation policy determines which saved program the agent edits. State is carried by the selected source and feedback rather than implicit conversation history.

The current program carries only the profile obtained for that exact program version. If no profile is available for the program obtained by applying the selected patch (the residual profile), its current dynamic evidence is unknown; an older profile is not substituted. Returning to the original program permits reuse of its original profile. The localization method and strategy-library adapter operate on evidence for the selected source version.

D.3 Open exploration

The controller can withdraw its localization ranking and optional strategy advice and allow open exploration using the public profiling information and source. This uses the remaining sessions. For these triggers, a usable record contains a nonempty patch admitted for evaluation and has no recorded validation failure; it may still have an inconclusive validation outcome. A successful result has passed the required validation checks. These conditions do not require the stable score greater than one needed for final selection. Open exploration is triggered when the last two attempts both lack a usable successful result, when two successful attempts produce exactly the same nonempty patch, or when neither of the last two attempts has a usable score exceeding the earlier score reference. That reference is the maximum of one and the scores of usable records preceding those two attempts. It is a search trigger, distinct from eligibility for final delivery.

Once open exploration starts, subsequent sessions remain in open exploration. The final configured session also uses open exploration if that mode has not already been entered. No additional session is granted by this change. Behavioral validation, timing, and selection of the best validated patch remain the same, and the agent may still use earlier feedback to repair its current program.

Algorithm 2 summarizes these decisions and the final return rule.

Algorithm 2: Optimization with a separate current program and best validated patch.

```
Input: original program P0, workload, tests
State: current = P0; best_patch = none; attempt_records = empty
for each of at most five sessions:
    obtain available evidence corresponding to current
    choose guided or open exploration from attempt_records and session
    count
    if guided:
        select at most three optimization opportunities for current
        attach optional matching strategy advice
    else:
        use public profiling information and source without ranking or
    advice
    supply source identity, latest feedback, and best summary
    ask the optimization agent to edit current
    save the resulting program as latest_patch relative to P0
    run applicable behavioral, build, and loading checks
    collect original/changed-program timing separately from profiling
    append latest_patch, checks, timing, and available profile of the
        patched program to attempt_records
    if latest_patch is eligible and
        (best_patch is none or score(latest_patch) > score(best_patch)):
        best_patch = latest_patch
    if latest_patch is nonempty, permitted, and normally returned:
        # Test or timing outcomes do not block continuation.
        current = apply(P0, latest_patch)
    else:
        current = apply(P0, best_patch) if available, otherwise P0
    keep only the profile matching the selected current program
return best_patch if available, otherwise P0
```

E Experimental Details

This appendix supplements the baseline adaptations and evaluation protocol in Section 3.1 with source-mapping details, implementation settings, full scoring definitions, and cost accounting. The controller's local validation and selection protocol is specified separately in Appendices C and D.

E.1 Baseline implementation and initialization

PEACE. We fine-tune PEACE's auxiliary optimizer using low-rank adaptation (LoRA), which freezes the base model weights and trains low-rank updates (Hu et al., 2022).

Optimo. Function replacement preserves the original signature and decorators (callables applied with Python's `@` syntax). The syntax and interface checks precede insertion into the repository; behavioral and performance evaluation follows the common protocol in Section 3.1.

Profile initialization. Frame mapping uses source paths, names, and abstract syntax tree (AST) source ranges to identify production functions. Each sampled stack is attributed to its leaf function; an unmappable native or library leaf is instead attributed to the deepest mappable repository Python caller. We call the sampled execution attributed to a function its leaf occupancy. Profile-Top1 selects the function with the largest leaf occupancy; ties are resolved by cumulative occupancy and deterministic source order. For cumulative occupancy, each distinct repository function in a sample

receives one count. Optimo's line weights aggregate samples by source line and normalize within the selected function; they estimate neither execution counts nor per-call times. CHICO uses its own profiling and static analysis from Section 2.

Reference initialization. Source verification checks the file, lexical scope, and function range to resolve the selected entry uniquely in the original snapshot. A function introduced only by the reference patch cannot itself serve as an original-code entry. The optimizer input follows Section 3.1; reference-modified lines are not converted into profiling weights.

E.2 Implementation settings

The task-input boundary is defined in Section 3.1. All GPT runs and the DeepSeek Codex baseline use Codex's code-reading, editing, and command interface. Other DeepSeek runs call its API, with each optimizer organizing its own processing steps.

Hardware. The evaluation server has two 96-core AMD EPYC 9655 processors with simultaneous multithreading enabled and two NUMA nodes. Its eight 64 GB DDR5 memory modules operate at 4800 MT/s. The host runs Ubuntu 22.04.5 LTS with Linux kernel 6.8.0-136-generic. It also has eight NVIDIA GeForce RTX 4090 GPUs; the installed driver (580.173.02) reports approximately 48 GiB of device memory per GPU. The main LLM interfaces are described above.

Strategy library. The frozen library contains 433 Python and 252 C/C++ strategies extracted from GitHub performance commits. Each entry combines optimization advice, before/after code, and applicability checks; matched entries guide edits at the selected locations. Construction and matching details are given in Appendix B.

E.3 Benchmark scoring

For GSO, each task returns one final optimization result as described in Section 3.1. Let N be the fixed task count, C_i indicate passed correctness tests, and s_i denote the speedup over the original program, aggregated across tests as defined below. Correctness and Sp@1 are

$$\text{Correctness} = \frac{100}{N}\sum_{i=1}^{N} C_i, \qquad \text{Sp@1} = \frac{100}{N}\sum_{i=1}^{N} \mathbf{1}[C_i = 1,\ s_i \geq 1.2]. \tag{3}$$

Failed optimization and unavailable initialization remain in the fixed task count N.

For GSO, define the harmonic mean $H(x_1, \ldots, x_m) = m/\sum_j x_j^{-1}$. For test j of task i, let $T_{0,ij}$ be the original program's runtime, $T_{a,ij}$ the runtime after applying the method's patch, and $T_{h,ij}$ the runtime after applying the expert patch. Then

$$\begin{aligned} s_i = H_j(T_{0,ij}/T_{a,ij}), \qquad r_i = H_j(T_{h,ij}/T_{a,ij}), \\ \text{Opt@1} = \frac{100}{N}\sum_i \mathbf{1}[C_i = 1,\ r_i \geq 0.95]. \end{aligned} \tag{4}$$

This follows GSO's published definition relative to the expert patch (Shetty et al., 2025). A performance-success indicator is zero if any runtime required for its test comparisons is missing, nonpositive, or nonfinite; the task remains in N. Correctness depends on recorded test outcomes. Each runtime ratio is computed within a test before aggregation across that task's tests; the final percentages count qualifying tasks rather than averaging their speedups. The archived evaluator's `opt_commit` flag additionally requires an improvement over the original with a rounded geometric-mean speedup of at least 1.2, and uses $r_i > 0.95$. We distinguish that flag from Equation 4; its `opt_base` flag likewise does not implement our harmonic-mean Sp@1.

SWE-fficiency-Lite. Let $N = 100$ and $C_i = 1$ only when the final optimization result passes all correctness tests required for task i; missing correctness output gives $C_i = 0$. An empty patch receives passing credit only if the evaluation records all required tests as passed. Let s_i and g_i be the

speedups achieved by the method's patch and the expert patch, respectively, each computed as its recorded mean original runtime divided by mean patched runtime. For performance scoring, define

$$\widehat{s}_i = \begin{cases} s_i, & \text{nonempty, correct patch with recorded workload runtimes,} \\ 1, & \text{otherwise,} \end{cases} \qquad r_i = \widehat{s}_i / g_i. \tag{5}$$

Here s_i requires finite, positive mean runtimes before and after the patch. The fallback assigns original-program speed, so its expert-relative ratio is $1/g_i$, not 1, and it does not change C_i. The two task rates are

$$\text{PassesTests} = \frac{100}{N} \sum_i C_i, \qquad \geq \text{Expert} = \frac{100}{N} \sum_i C_i \, \mathbf{1}[r_i \geq 1]. \tag{6}$$

Thus $\geq$Expert cannot exceed Passes Tests. Following SWE-fficiency (Ma et al., 2026), the aggregate Speedup Ratio is

$$\text{SR} = \frac{N}{\sum_i 1/\max(r_i, 0.001)}. \tag{7}$$

The floor limits small-ratio contributions; ratios above 1 remain uncapped. A few small ratios can lower SR even when many tasks reach expert speed. Thresholding and aggregation use unrounded task values; rounding is applied only for display.

All tasks remain in the denominator. Infrastructure failures and missing logs are recorded separately from test failures.

Relative score improvements. For a metric score a and comparison score b, the relative improvement is $100(a-b)/b\%$. For task rates, relative improvements are computed from task counts before rounding. Main-comparison ranges span the four models and four baselines, using Profile for PEACE and Optimo; Reference–Profile comparisons use their corresponding paired configurations, and the localization comparison pairs CHICO with *w/o localization*. These relative improvements differ from the absolute score differences in Table 2.

E.4 Cost accounting

Table 5 reports model cost per task for the main comparison in Table 2. Model cost is computed from token usage at the published API prices for GPT-5.6 Sol, Terra, Luna, and DeepSeek V4.1 Flash (accessed September 25, 2026). It includes all model calls in each run, including failed attempts; shared generation costs are counted once. Comparisons use the same model and benchmark.

Table 5: **Model cost per task ($) on GSO and SWE-fficiency-Lite.** Model configurations and the exclusion of Reference (†) from the main baseline comparison follow Table 2; —: not evaluated.

	GSO				SWE-fficiency-Lite			
System	Sol	Terra	Luna	DeepSeek	Sol	Terra	Luna	DeepSeek
Codex	4.306	2.223	0.363	0.215	6.563	4.805	0.565	0.412
PEACE-Profile	5.011	2.712	0.418	0.260	9.321	6.068	0.903	0.527
PEACE-Reference†	4.754	2.643	0.343	—	6.522	5.616	0.631	—
Optimo-Profile	1.950	1.186	0.151	0.136	3.076	2.524	0.335	0.224
Optimo-Reference†	1.646	1.076	0.144	—	2.912	2.407	0.271	—
POLO	6.705	4.061	0.569	0.332	11.314	8.221	1.084	0.569
CHICO	4.695	2.859	0.410	0.241	10.361	5.395	0.952	0.473

Model cost and optimization performance. Across the eight model–benchmark comparisons, CHICO costs more per task than Codex and Optimo-Profile, less than PEACE-Profile in five configurations, and 8.42–34.38% less than POLO in all eight (Table 5). CHICO also outperforms all four baselines on every optimization metric (Table 2). For example, with GPT-5.6 Sol on GSO, CHICO costs $4.695 per task versus $4.306 for Codex, a 9.03% increase, while Opt@1 rises from 32.35% to 63.73%, a 96.97% relative improvement. These comparisons show that CHICO achieves stronger optimization at additional model cost relative to Codex and Optimo-Profile, while improving both model cost and optimization performance over POLO.