EFFIPAIR: IMPROVING THE EFFICIENCY OF LLM-GENERATED CODE WITH DIFFERENTIAL EXECUTION FEEDBACK

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) can generate functionally correct programs that differ substantially in execution efficiency. Existing inference-time optimization methods typically refine each candidate using pointwise runtime or profiling feedback, which identifies how costly an implementation is or where the cost arises, but offers limited guidance on how the computation should change. We introduce DIFFERENTIAL EXECUTION FEEDBACK (DEF), which compares implementations that are nearby in structural program space but separated in performance, turning their execution and implementation differences into directional optimization evidence. We demonstrate DEF in EFFIPAIR, a training-free, test-time framework that pairs structurally similar programs with different efficiencies, distills their relative execution behavior into compact feedback, and iteratively refines a candidate pool. Across EvalPerf, Mercury, and ENAMEL, using GPT-4o mini, DeepSeek-V4.1 Flash, and GPT-5 mini, EFFIPAIR achieves the highest value on each benchmark's official efficiency metric in all nine model–benchmark settings under matched evaluation conditions. Moreover, two contrastive refinement rounds improve efficiency over the selected initial draft in every setting while preserving or improving Pass@1. These results demonstrate the effectiveness of relational execution feedback as a lightweight signal for test-time code optimization.

1 INTRODUCTION

Large language models (LLMs) can generate correct programs across a wide range of programming tasks, yet these programs can differ substantially in runtime, memory consumption, and computational cost. Such differences matter when generated code is repeatedly executed, incorporated into larger software systems, or deployed at scale, where seemingly small inefficiencies can accumulate into significant overhead. Improving the efficiency of LLM-generated code is therefore an increasingly important dimension of code generation alongside functional correctness. Recent work treats program efficiency as an explicit optimization objective. Benchmarks such as EffiBench (Huang et al., 2025b) reveal substantial efficiency gaps between model-generated and expert implementations, motivating a growing class of inference-time methods that execute candidate programs and return runtime, memory, or profiling feedback to guide refinement (Huang et al., 2025a; Peng et al., 2024; Ye et al., 2025). The central challenge, however, is converting this evidence into a useful implementation change. Existing inference-time refinement approaches largely provide feedback pointwise: runtime measurements reveal how costly an implementation is, and profiles identify where that cost occurs, but neither directly indicates how the computation should change. The model must therefore infer the performance-improving transformation from an isolated diagnosis, even though alternative implementations of the same computation may already provide useful comparative evidence.

Our key observation is that alternative implementations can turn execution feedback from a diagnosis into a direction for refinement. Consider two functionally equivalent programs that are structurally similar but substantially separated in execution cost. Because much of their implementation is shared, the choices on which they differ become natural candidates for explaining how the faster program avoids costs incurred by the slower one. In contrast, arbitrarily different programs may introduce too many confounding changes to yield a clear refinement signal. We therefore seek contrasts that

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.

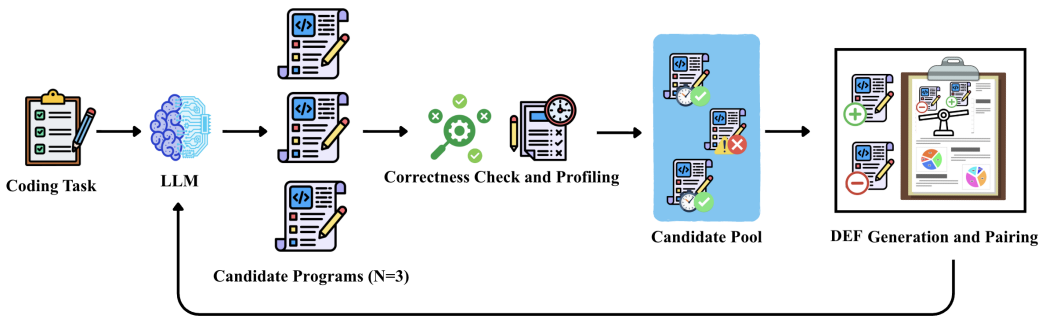

Figure 1: **Overview of EFFIPAIR.** Given a coding task, the LLM samples N candidate programs. Candidates are checked for correctness and profiled for efficiency, then stored in a candidate pool. EFFIPAIR selects a pair consisting of an efficient reference program p^+ and a similar but less efficient candidate p^-, summarizes their relative execution differences into compact DIFFERENTIAL EXECUTION FEEDBACK (DEF), and uses this signal to refine the program. The refined program is added back to the pool for subsequent rounds.

are nearby in structural program space but distant in performance space: similar enough for their differences to be interpretable, yet separated enough in cost to provide useful optimization evidence.

Building on this observation, we introduce DIFFERENTIAL EXECUTION FEEDBACK (DEF), an execution-grounded refinement signal that converts differences between an efficient program p^+ and a structurally related but less efficient program p^- into directional optimization guidance. We operationalize this idea in **EFFIPAIR**, a training-free test-time framework that uses a pool of candidate programs not only to search for strong solutions, but also to construct informative contrasts for refinement. As illustrated in Figure 1, EFFIPAIR first generates and evaluates a small candidate pool, then selects an efficient reference p^+ together with a structurally similar but slower candidate p^-, and constructs DEF from the execution and implementation differences between them. The resulting signal guides the model toward performance-improving changes, after which the refined candidate is evaluated and returned to the pool for subsequent rounds. This perspective also gives candidate diversity a different role at inference time. Whereas in conventional sampling or best-of-N generation, diversity is primarily useful for search, EFFIPAIR can also use an unselected candidate as optimization evidence when contrasted with a nearby, better-performing implementation. Candidate diversity thus becomes both a search resource and a source of cross-candidate refinement signals.

We further ask whether informative contrasts must be generated for the task or can instead be recovered from existing software context. At repository scale, we study contrasts from prior versions and structurally related code, testing whether DEF extends beyond self-contained generation to optimization under real software constraints.

We evaluate EFFIPAIR across multiple code-efficiency benchmarks and strong proprietary and open-source models. Across models, EFFIPAIR improves efficiency while preserving correctness and reducing feedback overhead, and we additionally evaluate the contrastive formulation at repository scale. Across EvalPerf, Mercury, and ENAMEL, using GPT-4o mini, DeepSeek-V4.1 Flash, and GPT-5 mini, EFFIPAIR achieves the highest value on the official efficiency metric in all 9 model–benchmark settings while preserving or improving Pass@1.

In summary, our contributions are: (1) We identify a limitation of existing inference-time execution-guided code optimization: pointwise measurements reveal *how costly* an implementation is and *where* that cost occurs, but provide limited guidance on *how it should change*. Performance differences between functionally equivalent implementations instead provide directional optimization evidence. (2) We introduce **DIFFERENTIAL EXECUTION FEEDBACK (DEF)**, an execution-grounded refinement signal that compares implementations nearby in structural program space but separated in performance space, turning isolated performance observations into actionable guidance. (3) We present **EFFIPAIR**, a training-free test-time framework that operationalizes DEF through informative efficient–inefficient pairs and compact relative execution feedback. Across multiple code-efficiency benchmarks and strong language models, EFFIPAIR improves efficiency while preserving correctness

and reducing feedback context. We further study repository-level optimization using generated, historical, and structurally related implementations. (4) We show that test-time candidate diversity can provide an optimization signal beyond search. Through comparisons with best-of-N sampling, single-candidate profiling, paired prompting without profiling, and alternative pairing strategies, we isolate the benefit of relational execution feedback.

2 Related Work

Large language models (LLMs) are increasingly used for code generation, ranging from interactive prototyping to production pipelines (Jiang et al., 2026; Fakhoury et al., 2024; Jimenez et al., 2024). However, generated programs are often inefficient in runtime and memory (Huang et al., 2025b), leading to nontrivial cost and sustainability concerns at scale. Recent work therefore treats efficiency as an explicit optimization objective, considered alongside (Peng et al., 2024), or even prior to, correctness (Ye et al., 2025). A growing set of benchmarks has been proposed to evaluate this problem (Waghjale et al., 2024; Huang et al., 2025b; Du et al., 2024; Liu et al., 2024; Qiu et al., 2025), consistently showing that LLM-generated code lags behind expert implementations in efficiency. To address this gap, prior work explores both training-time and inference-time strategies. Training-time approaches curate performance-improving edits or incorporate efficiency-aware fine-tuning (Shypula et al., 2024; Huang et al., 2025c). Code-Optimise (Gee et al., 2025) additionally constructs correctness- and efficiency-based preference pairs for preference-based training. Inference-time methods instead rely on iterative refinement using execution feedback. Waghjale et al. (2024) study execution-guided refinement in ECCO, which feeds correctness, runtime, and memory measurements from test executions back to the model. EffiLearner (Huang et al., 2025a) feeds runtime and memory profiles back to the model, while PerfCodeGen (Peng et al., 2024) uses execution feedback from test runs to guide refinement. LLM4EFFI (Ye et al., 2025) further structures this process by separating algorithm selection, implementation optimization, and correctness refinement. More broadly, compiler feedback has also been incorporated into neural code generation to improve compilability (Wang et al., 2022). At repository scale, recent work has explored LLM-based optimization of real-world codebases. ECO (Lin et al., 2025) mines historical performance-improving commits and retrieves similar code locations to transfer optimization patterns, while PerfAgent (Deng et al., 2026) iteratively guides a coding agent using summarized profiler and correctness feedback. Benchmarks such as SWE-Perf (He et al., 2026) and GSO (Shetty et al., 2025) have been introduced to evaluate code efficiency at repository scale.

Across program- and repository-level execution-guided refinement, feedback is typically pointwise, using runtime or profiling information from the current candidate in isolation. These signals reveal cost or bottlenecks but offer limited guidance on how the implementation should change. We instead compare structurally similar candidates with different efficiency profiles to derive directional evidence for refinement. Similarity-based pairing turns candidate diversity into a cross-candidate optimization signal, while EffiPair jointly preserves correctness and improves efficiency (Ye et al., 2025; Peng et al., 2024; Huang et al., 2025a).

3 Methodology

EffiPair improves the efficiency of LLM-generated code through Differential Execution Feedback (DEF). Rather than refining a program from its execution behavior in isolation, DEF compares implementations that are structurally similar but differ substantially in execution efficiency. The central design principle is therefore to select programs that are *nearby in structural program space but distant in performance space.* Structural proximity makes the contrast interpretable: when two programs share most of their implementation, the relatively small set of choices on which they differ provides localized evidence about which design decisions may account for the performance gap. At the same time, a sufficiently large difference in execution cost makes the comparison informative. In contrast, comparing arbitrarily different programs introduces many simultaneous implementation changes, making it difficult to determine which differences are relevant to efficiency. Appendix Figure 4 provides an example of this process, showing a highly similar program pair with a substantial runtime gap and the resulting refinement. Using this contrast, DEF reduces the faster program from 899.6 to 100.3 *ms* in two rounds, illustrating how nearby implementations can provide actionable optimization guidance.

3.1 PROBLEM SETUP

Consider a programming task $\mathcal{T}(d, \mathcal{S})$, where d is a natural-language task description and $\mathcal{S} = \{s_i\}_{i=1}^{|\mathcal{S}|}$ is a set of correctness tests. For a candidate program p, let $s_i(p) \in \{0, 1\}$ indicate whether p passes the i-th test, and let $e(p)$ denote its efficiency cost, such as execution time. Our objective is to find the most efficient functionally correct program:

$$p^* = \arg\min_p e(p) \quad \text{s.t.} \quad \forall i,\ s_i(p) = 1.$$

EFFIPAIR approaches this objective through repeated propose–evaluate–refine rounds. It maintains a pool of candidate programs, evaluates them for correctness and efficiency, constructs informative efficient–inefficient pairs, and derives DEF from their relative execution behavior to guide subsequent refinement.

3.2 CANDIDATE GENERATION AND PAIR SELECTION

We implement EFFIPAIR as an iterative propose–evaluate–refine pipeline. Given a task description d, we first generate an initial pool of candidates,

$$C_0 = \{p_0^{(j)}\}_{j=1}^{N},$$

by independently prompting the LLM with d without any additional guiding signals and requesting a code solution. Sampling the model N times produces diverse candidate implementations that can subsequently serve both as potential solutions and as contrasts for refinement. At round t, each candidate in the current pool is evaluated for functional correctness and execution efficiency. For each candidate p, we additionally construct an abstract syntax tree (AST) representation $A(p)$, which is used to measure structural similarity between programs. We first select the most efficient correct candidate as the reference program p^+:

$$p^+ = \arg\min_{p \in C_t} e(p) \quad \text{s.t.} \quad \forall i,\ s_i(p) = 1.$$

We then construct a less efficient but structurally similar partner p^-. Let

$$\mathrm{sim}_{\mathrm{AST}}(p, q) = \cos\big(A(p), A(q)\big)$$

denote the AST similarity between two programs. Given a similarity threshold τ, we define the set of sufficiently similar correct candidates

$$\mathcal{N}(p^+) = \left\{ q \in C_t : q \neq p^+,\ \forall i,\ s_i(q) = 1,\ \mathrm{sim}_{\mathrm{AST}}(p^+, q) \geq \tau \right\}.$$

Among these candidates, we select the least efficient program:

$$p^- = \arg\max_{q \in \mathcal{N}(p^+)} e(q).$$

This selection produces a pair that is structurally close but separated in execution efficiency: the similarity constraint limits unrelated implementation differences, while choosing the slowest candidate within the neighborhood provides a sufficiently strong performance contrast for constructing DEF. The formulation applies when at least two functionally correct candidates are available. As a correctness-recovery fallback, when only one correct candidate is available and a standard correct–correct DEF pair cannot be constructed, we retain the correct candidate as Candidate A and select a structurally similar incorrect candidate as Candidate B, along with its failure information. In this setting, the resulting feedback is used primarily to preserve correctness while searching for an improved implementation, rather than to infer an efficiency contrast between the two candidates. If no correct candidate is available, two incorrect candidates and their failure information are provided to the model, which is instructed to restore correctness before optimizing efficiency.

3.3 ITERATIVE REFINEMENT

Given the selected pair (p^+, p^-), EFFIPAIR constructs DEF from their implementation and execution differences. Rather than providing performance information for either program in isolation, DEF provides the model with a contrastive signal between the two candidates, relating the implementation choices on which they differ to their observed difference in execution efficiency. This provides the LLM with a directional signal about which parts of the less efficient implementation can be changed while preserving the behavior of the efficient reference. Let $\Delta(p^+, p^-)$ denote the resulting DEF signal. Given the task description d, the selected pair (p^+, p^-), and their differential feedback, the refinement step produces a new candidate

$$\tilde{p} = R\big(d, p^+, p^-, \Delta(p^+, p^-)\big),$$

where R denotes the LLM-based refinement procedure. The model is instructed to preserve functional correctness while using the differences between the two implementations to identify performance-improving changes. The refined candidate $\tilde{p}$ is then evaluated for correctness and efficiency and added back to the candidate pool. Pair selection and refinement are repeated on the updated pool in subsequent rounds, allowing newly generated candidates to become either stronger reference programs or informative contrast candidates. Because previously generated candidates are retained, an unsuccessful refinement does not discard the best correct solution already found.

4 EXPERIMENTS

4.1 SETUP

We evaluate EFFIPAIR on a suite of code-generation benchmarks designed to assess not only functional correctness but also the execution efficiency of generated programs. Our analysis emphasizes benchmark-specific metrics which represent relative improvement, rather than absolute execution times, since raw latency is sensitive to hardware, operating system scheduling, and library versions. Specifically, we study the effect of EFFIPAIR on three code-efficiency benchmarks: Mercury (Du et al., 2024), ENAMEL (Qiu et al., 2025), and EvalPerf (Liu et al., 2024). EvalPerf targets performance-challenging tasks and reports **DPS**, which measures a solution's efficiency relative to the nearest slower reference solution, with the normalized version $\textbf{DPS}_{\text{norm}}$. Mercury uses the **Beyond** metric, which normalizes runtime percentiles over each task's runtime distribution to enable hardware-robust comparisons. ENAMEL reports **eff@1**, a weighted score based on the worst execution time across test cases of different difficulty levels; higher values indicate better efficiency, and values above 1 mean the generated code outperforms the expert solution. We also report **Pass@1** for all runs. We use a diverse set of language models to examine whether EFFIPAIR can improve systems that already have strong efficiency and correctness performance. Specifically, our evaluation includes GPT-5.6 Luna, GPT-5 mini, GPT-4o mini (OpenAI, 2026; 2024; 2025), Claude Sonnet 5 (Anthropic, 2026), DeepSeek-V4.1 Flash (DeepSeek, 2026), and Gemini 3.8 Flash (Google DeepMind, 2026). This setting provides a stringent test of whether EFFIPAIR remains effective even when the underlying model is already highly capable.

EFFIPAIR uses $N = 3$ initial candidates and $T = 2$ refinement rounds by default, which provides a practical balance between candidate diversity and inference cost. The three candidates are independent samples generated from the same prompt using different seeds. The similarity window is chosen by default to be $[0.85, 1.0]$ (see Appendix A.1 for further analysis) and byte-identical partners are excluded, as they carry no contrastive signal. The similarity metric, described in Section 3.2, is computed on abstract syntax trees. To reduce test-case overfitting, benchmark tests are not included in the prompt for our method (EFFIPAIR); instead, the model receives only compact execution and profiling feedback from `cProfile` (Python Software Foundation), which is a built-in Python profiler. We compact the profiler output into a short summary; implementation details are provided in Appendix A.2.1, and the rationale for condensed profiling is discussed in Appendix A.2.2. To avoid selection bias from runtime noise, candidate selection and final evaluation use separate measurements. After EFFIPAIR selects the fastest correct candidate using its internal profiling measurements, the selected program is independently re-executed under the benchmark's official evaluation harness, and only this independent measurement is used for reported efficiency results. On EvalPerf, EFFIPAIR's

		EvalPerf			Mercury			ENAMEL		
Model	Method	DPS_{norm}	Pass@1	Tok.	Beyond@1	Pass@1	Tok.	eff@1	Pass@1	Tok.
GPT-4o mini	Baseline (instruct)	75.50	89.83	0.2	81.06	92.19	0.5	49.78	83.80	0.3
	EffiLearner-Gen	80.28	**88.98**	0.6	79.94	90.62	0.5	48.51	83.10	0.6
	ECCO nl2code	79.49	82.20	0.3	79.68	89.45	0.5	53.99	83.80	0.7
	ECCO exec-refine	74.88	67.80	0.7	80.32	89.84	1.2	47.99	75.35	1.3
	ECCO self-refine	74.59	61.02	2.1	74.49	85.16	2.6	36.22	60.56	2.9
	ECCO nl+exec-refine	73.49	50.85	2.0	76.22	86.33	2.7	38.76	68.31	2.9
	EffiLearner-Opt	74.34	86.44	6.3	82.84	91.80	9.7	54.80	83.80	5.6
	LLM4EFFI	81.08	72.88	59.1	76.85	85.16	79.6	42.52	71.13	78.3
	EFFIPAIR r_0	83.58	88.14	1.4	86.32	**95.18**	1.7	57.50	**86.62**	1.9
	EFFIPAIR r_2	**84.25**	**88.98**	3.0	**86.62**	**95.18**	3.9	**59.71**	**86.62**	5.1
DeepSeek-V4.1 Flash	Baseline (instruct)	81.48	88.14	0.2	89.17	**98.05**	0.5	56.38	93.66	0.3
	EffiLearner-Gen	81.88	88.14	0.4	88.98	97.27	0.5	58.70	**94.37**	0.7
	ECCO nl2code	84.36	82.20	0.3	89.51	95.70	0.5	68.45	90.85	0.9
	ECCO exec-refine	82.97	83.90	0.7	89.81	95.70	1.0	71.04	92.96	1.4
	ECCO self-refine	80.05	80.51	1.9	85.46	93.75	3.1	63.41	85.92	3.4
	ECCO nl+exec-refine	80.10	75.42	2.1	87.07	94.14	3.0	66.85	86.62	3.4
	EffiLearner-Opt	85.79	85.59	4.3	92.44	96.88	8.7	73.06	**94.37**	4.5
	LLM4EFFI	85.36	89.83	35.7	83.09	88.28	42.5	75.86	88.73	44.8
	EFFIPAIR r_0	87.59	88.98	1.1	92.57	97.79	1.5	71.16	**94.37**	2.2
	EFFIPAIR r_2	**89.50**	**91.53**	2.6	**94.24**	97.79	3.7	**77.58**	**94.37**	5.2
GPT-5 mini	Baseline (instruct)	79.56	86.44	0.2	89.78	96.48	0.6	56.19	94.37	0.4
	EffiLearner-Gen	82.34	86.44	0.4	88.31	95.70	0.6	57.82	**95.77**	0.4
	ECCO nl2code	82.35	85.59	0.4	88.32	95.31	0.6	61.48	92.25	0.6
	ECCO exec-refine	76.48	83.05	1.1	89.12	95.70	1.3	62.60	91.55	1.2
	ECCO self-refine	74.84	70.34	2.9	87.76	95.70	3.3	45.70	73.24	3.4
	ECCO nl+exec-refine	78.53	67.80	3.1	87.70	94.92	3.2	55.58	84.51	3.1
	EffiLearner-Opt	82.92	87.29	5.7	91.40	95.70	9.4	70.35	94.37	4.8
	LLM4EFFI	73.62	83.05	36.1	86.38	92.97	47.4	63.65	88.03	43.2
	EFFIPAIR r_0	85.42	**89.83**	1.0	91.33	**97.27**	1.8	69.74	95.07	1.2
	EFFIPAIR r_2	**85.95**	**89.83**	3.0	**92.32**	**97.27**	4.4	**72.47**	95.31	6.0

Table 1: EFFIPAIR versus prior efficiency-oriented methods, all reproduced with the same models, decoding settings and evaluation harnesses. Each benchmark reports its official efficiency metric and Pass@1 (%). r_0 is EFFIPAIR's selected draft before refinement and r_2 its output after two contrastive rounds; both are means over three independent runs, while other rows are single runs. Bold marks the best efficiency and Pass@1 within each model block. Tok. is thousands of tokens per task (prompt + completion) for that run; for EFFIPAIR it covers the three sampled drafts and, in the r_2 row, the two refinement rounds.

internal pool gate uses only the EvalPlus base suite (a mean of 3.9 inputs per task), while final scoring uses the official harness over base + plus (285 inputs per task); EffiLearner's optimization stage gates on the same base suite, and ECCO's execution-refining variants use at most three cases. On Mercury, EFFIPAIR validates against the benchmark's shipped `test_cases` through the same builder the official scorer uses, as do ECCO and EffiLearner. On ENAMEL, EFFIPAIR and ECCO's execution-refining variants share an identical gate—16 of the scorer's 20 cases under the official seeds, while EffiLearner draws the same number of cases under different seeds. LLM4EFFI touches no benchmark tests on any of the three and uses 20 self-made test cases per task. All experiments were run on dedicated AWS EC2 `c7i.2xlarge` instances (Intel Xeon Scalable "Sapphire Rapids", 3.2 GHz sustained; 4 physical cores / 8 logical threads, 16 GiB RAM, 80 GB gp3 EBS) in the `us-east-1` region, running Ubuntu 24.04 LTS (kernel `7.0.0-1012-aws`) with Python 3.12.3.

4.2 BASELINES AND COMPARISON METHODS

We compare EFFIPAIR against three published efficiency-oriented systems, each reproduced from its official implementation under the same models, decoding settings and evaluation harnesses. **ECCO** (Waghjale et al., 2024) refines a single generated program from feedback; we include all three of its refinement modes – natural-language self-critique, execution feedback, and natural-language

Method	DeepSeek-V4.1 Flash eff@1 / Pass@1	Gemini 3.8 Flash eff@1 / Pass@1	GPT-5.6 Luna eff@1 / Pass@1	Claude Sonnet 5 eff@1 / Pass@1
Generation				
Greedy Draft	55.73 / 93.66	60.88 / 97.18	55.29 / 90.14	56.01 / 94.37
Single Sampled Draft	65.30 ± 2.34 / 93.66	78.99 ± 1.34 / 96.71	64.36 ± 0.71 / 89.20	62.76 ± 0.96 / 93.66
Best-of-5	72.61 ± 1.99 / 93.19	85.23 ± 0.96 / **98.12**	71.58 ± 1.23 / **94.37**	69.09 ± 1.74 / **96.24**
Pool Selection (r_0)	71.52 ± 1.32 / 94.84	83.70 ± 0.81 / 97.42	69.91 ± 1.06 / 93.66	67.38 ± 1.20 / 94.60
+ *refinement*				
Isolated Refine	75.11 ± 1.52 / 95.54	84.81 ± 0.36 / 97.42	70.75 ± 0.53 / 93.90	70.64 ± 0.91 / 95.07
Random-Pair Refine	76.25 ± 1.80 / 96.24	87.65 ± 0.72 / 97.18	71.47 ± 1.48 / 93.66	69.09 ± 1.30 / 94.37
Matched-Pair Refine	77.95 ± 1.54 / 96.01	88.64 ± 0.28 / 97.42	71.70 ± 1.10 / 93.66	70.39 ± 0.58 / 94.60
+ *execution feedback*				
Isolated Refine + EF	79.09 ± 1.08 / 96.60	90.05 ± 0.40 / 97.18	75.18 ± 0.65 / 93.90	72.79 ± 2.10 / 94.37
Random-Pair Refine + EF	79.92 ± 1.61 / 96.01	90.18 ± 0.18 / 97.42	74.07 ± 1.41 / 93.90	73.86 ± 1.38 / 94.37
EFFIPAIR	**80.25** ± 1.27 / **96.71**	**90.93** ± 0.51 / 97.42	**75.34** ± 0.47 / 93.90	**73.93** ± 1.23 / 94.60

Table 2: Component ablation on ENAMEL (142 tasks), reporting the official eff@1 / Pass@1 (%) metrics across four model families. We compare generation-only baselines with isolated refinement, random-pair refinement, and matched-pair refinement, with and without the corresponding constructed execution feedback (EF). In the last row, DEF is implemented by combining matched-pair refinement with execution feedback to create EFFIPAIR. Best-of-5 is included as a generation-only control. All refinement methods operate on the same initial candidate pool and use the same correctness gate. Bold denotes the best reported value for each metric within each model.

reflection over execution feedback – alongside its direct *nl2code* generation. **EffiLearner** (Huang et al., 2025a) profiles the current program with line- and memory-level instrumentation, returns the overhead report to the model, and accepts a candidate only if it executes and lowers the measured memory integral; we report both its initial generation and its output after five self-optimization iterations. **LLM4EFFI** (Ye et al., 2025) instead searches: it formalizes the task, proposes several algorithms and implementations, synthesizes its own test cases to verify them, and selects a final program by model judgment. As a lower bound we include an instruction-only baseline at temperature 0, using LLM4EFFI's own direct-generation prompt and sanitizer. Together these span the design space around EFFIPAIR (single-call generation, feedback-driven refinement of one program, and search over many candidates) and they differ by more than two orders of magnitude in inference cost, which is why we report tokens per task alongside every efficiency number.

4.3 MAIN RESULTS

Table 1 establishes the end-to-end effectiveness of EFFIPAIR. EFFIPAIR achieves the highest official efficiency across nine settings after two refinement rounds (r_2), indicating that the benefit is not tied to a particular model family or efficiency metric. The comparison between r_0 and r_2 is particularly important since it isolates the contribution of DEF beyond the gains from initial candidate selection. Across all settings r_2 improves over r_0, which further reinforces EFFIPAIR's central claim that the refinement stage contributes beyond candidate selection: sampling three candidates and keeping the fastest correct one does not exhaust the available efficiency, and the subsequent rounds recover gains that selection alone leaves behind. Subsection 4.4 further isolates the impact of each component. Importantly, Table 1 shows that the efficiency gains do not come at the expense of functional correctness. Across the nine model–benchmark settings in Table 1, Pass@1 never decreases from r_0 to r_2: it is preserved in six settings and improves in three. Figure 2 provides further evidence for this behavior across refinement rounds: Pass@1 remains stable or improves as efficiency optimization proceeds. Together, these results show that EFFIPAIR can continue extracting efficiency gains while preserving the correctness established during candidate generation and selection.

4.4 ABLATION RESULTS

Table 2 disentangles the contributions of candidate selection, pairing, and execution feedback to EFFIPAIR. With the initial pool and correctness gate held fixed, EFFIPAIR improves over *Pool*

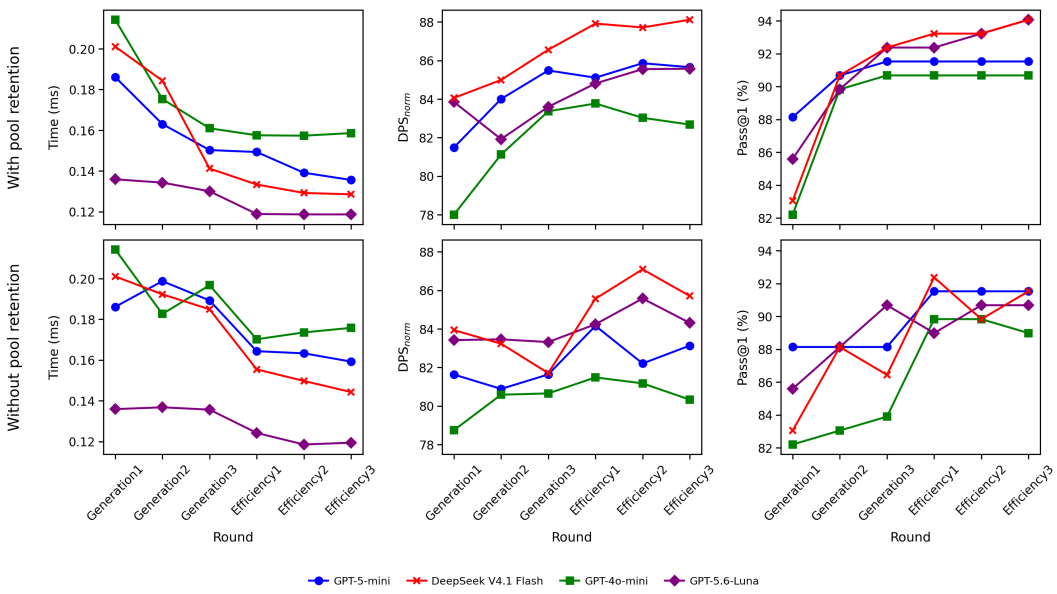

Figure 2: Iterative refinement behavior of EFFIPAIR on EvalPerf across evaluated models.

Selection ($r0$) by 5.43–8.73 eff@1 points across all four model families. It also outperforms *Best-of-5* in every case, showing that its gains cannot be explained by additional sampling alone. The ablations further show that both execution feedback and pair construction matter.

Adding execution feedback improves *Isolated Refine* for every model, while EFFIPAIR consistently exceeds *Random-Pair Refine + EF*. Relative to *Matched-Pair Refine* without execution feedback, the full method gains 2.29–3.64 eff@1 points. This suggests a complementary effect: execution feedback supplies the performance signal, while similarity-based pairing focuses that signal on localized implementation differences that are more useful for refinement. These efficiency gains are achieved without sacrificing correctness: EFFIPAIR matches or improves the $r0$ Pass@1 for all four models. Overall, Table 2 shows that EFFIPAIR benefits not merely from selecting strong candidates, but from turning relationships among candidates into actionable, execution-grounded refinement signals.

4.5 ITERATIVE REFINEMENT

Figure 2 examines the behavior of EFFIPAIR over successive generation and efficiency-refinement checkpoints on EvalPerf. The top row reports the standard setting with pool retention. Across all four models, execution time decreases substantially, DPS_{norm} improves over the initial generation, and Pass@1 is preserved or increased. Importantly, these gains extend beyond initial candidate selection, with later efficiency rounds continuing to improve runtime and, in most cases, DPS_{norm}. The magnitude and timing of improvement vary across models—some improve sharply in early rounds and then stabilize, while others continue to benefit from later refinement—but the overall pattern shows that EFFIPAIR extracts additional efficiency from an already strong candidate pool without sacrificing correctness. The bottom row isolates the role of pool retention by repeating the same analysis without retaining previously discovered candidates. The resulting trajectories are noticeably less stable. A refinement that improves a metric at one checkpoint can be followed by a worse rewrite at the next: execution time increases in several transitions, DPS_{norm} can fall after reaching a stronger value, and Pass@1 similarly exhibits regressions after earlier gains. In the standard EFFIPAIR loop, refined candidates are instead added back to the existing pool, so a successful candidate remains available even when a later refinement is unsuccessful. Pool retention therefore turns iterative refinement into a best-so-far process: once EFFIPAIR discovers a strong correct solution, subsequent rounds can improve upon it without requiring that solution to be discarded. The contrast between the two rows shows that retention is important not only for obtaining good candidates, but also for preventing later optimization steps from undoing gains already found.

Arm	Contrast source	n	Mean gain (%)
Best seed (no refinement)	none	125	1.21
Solo (profile only)	none	125	1.19
Same-site pair	resampled variant	125	1.24
History pair	own prior version	107	1.16
Sibling pair	similar function	79	0.85
History + sibling	both	119	1.06
Retrieved exemplar	BM25 perf-commit	125	1.17
ablation: no pool retention	none	120	0.76

Table 3: EFFIPAIR at repository scale on SWE-Perf (125 of 140 instances; exclusions in Appendix A.5). Each arm commits one patch per instance, the fastest correct member of its pool on EFFIPAIR's own internal clock, which is not the metric reported here, and all arms and the expert patch are then measured together in one scoring session; **Mean gain** is the significance-gated improvement under SWE-Perf's unmodified harness over that row's n instances.

4.6 REPOSITORY-LEVEL EXPERIMENTS

To test EFFIPAIR in a more realistic setting, we evaluate it at repository scale, where a candidate is a patch against a live codebase rather than a self-contained function. We use **SWE-Perf**, a benchmark of 140 instances derived from real performance-improving pull requests across nine widely used Python repositories. Each instance provides a repository snapshot at the pre-merge commit, a set of performance tests, the expert patch that was actually merged, and—in the *oracle* setting we adopt—the names of the functions that patch modified. We score every arm with SWE-Perf's own unmodified evaluation harness, which times each performance test over repeated runs in a fresh container and reports a gain only when a one-sided Mann–Whitney test distinguishes it from run-to-run noise. This setting changes what EFFIPAIR must do. At function level, the unit of generation, the unit of measurement, and the unit of edit coincide; in a repository they come apart, and a candidate must preserve the surrounding API and pass the project's own tests before its runtime is even measurable. It also changes where contrastive pairs can come from: rather than resampling the same function, a repository supplies its own history and its own structurally similar code. Our experiments hold the model, the instances, and the meter fixed and vary only the source of the contrast, isolating the contribution of *pair quality* from that of refinement itself.

5 LIMITATIONS

EFFIPAIR has several limitations. Its effectiveness depends on identifying informative candidate pairs: limited diversity, weak structural similarity, or noisy execution measurements can reduce the quality of the refinement signal. In addition, EFFIPAIR currently optimizes primarily for correctness and execution efficiency, without explicitly considering software qualities such as readability and maintainability. In some cases, performance-oriented refinements may produce more specialized or compact implementations that are harder to understand or modify, creating a trade-off between efficiency and code quality. EFFIPAIR also introduces additional generation, execution, and profiling overhead, making it most suitable when the optimized code is executed frequently enough to justify this cost. Future work could incorporate readability, maintainability, and other software-quality objectives into the refinement process, while also improving pair selection and adaptive similarity criteria.

6 CONCLUSION

We introduced EFFIPAIR, a training-free test-time framework that uses DIFFERENTIAL EXECUTION FEEDBACK (DEF) to improve the efficiency of LLM-generated code. By contrasting structurally similar programs with different execution behavior, EFFIPAIR turns candidate diversity into a useful signal for refinement. Across multiple benchmarks and language models, this approach consistently improves efficiency while preserving functional correctness. Overall, our results suggest that DEF provides a practical and general mechanism for guiding test-time code optimization.

AI USE STATEMENT

Generative AI tools were used as writing assistants to improve the clarity, grammar, phrasing, and overall readability of author-written text. All AI-assisted text was reviewed and edited by the authors, who take responsibility for the final content of the paper.

REFERENCES

Anthropic. Introducing claude sonnet 5. https://www.anthropic.com/news/claude-sonnet-5, June 2026. Accessed: 2026-09-22.

DeepSeek. Introducing deepseek-v4.1-flash: Smarter, faster, more efficient. https://www.deepseek.com/en/news/deepseek-v4-1-flash/, September 2026. Accessed: 2026-09-22.

Ryan Deng, Yuanzhe Liu, Bastian Lipka, Yao Ma, Xuhao Chen, Tim Kaler, and Jatin Ganhotra. Perfagent: Profiler-guided iterative refinement for repository-level code optimization, 2026. URL https://arxiv.org/abs/2607.19653.

Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. Mercury: A code efficiency benchmark for code large language models, 2024. URL https://arxiv.org/abs/2402.07844.

Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. Llm-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Transactions on Software Engineering*, 50(9):2254–2268, September 2024. ISSN 2326-3881. doi: 10.1109/tse.2024.3428972. URL http://dx.doi.org/10.1109/TSE.2024.3428972.

Leonidas Gee, Milan Gritta, Gerasimos Lampouras, and Ignacio Iacobacci. Code-optimise: Self-generated preference data for correctness and efficiency, 2025. URL https://arxiv.org/abs/2406.12502.

Google DeepMind. Introducing gemini 3.8 flash and 3.8 flash cyber. https://blog.google/innovation-and-ai/models-and-research/gemini-models/3-8-flash-and-3-8-flash-cyber/, September 2026.

Xinyi He, Qian Liu, Mingzhe Du, Lin Yan, Zhijie Fan, Yiming Huang, Yin Zheng, Zejian Yuan, and Zejun Ma. Swe-perf: Can language models optimize code performance on real-world repositories?, 2026. URL https://arxiv.org/abs/2507.12415.

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, 2025a. URL https://arxiv.org/abs/2405.15189.

Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M. Zhang. Effibench: Benchmarking the efficiency of automatically generated code, 2025b. URL https://arxiv.org/abs/2402.02037.

Dong Huang, Guangtao Zeng, Jianbo Dai, Meng Luo, Han Weng, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie M. Zhang. Efficoder: Enhancing code generation in large language models through efficiency-aware fine-tuning, 2025c. URL https://arxiv.org/abs/2410.10209.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–72, January 2026. ISSN 1557-7392. doi: 10.1145/3747588. URL http://dx.doi.org/10.1145/3747588.

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?, 2024. URL https://arxiv.org/abs/2310.06770.

Hannah Lin, Martin Maas, Maximilian Roquemore, Arman Hasanzadeh, Fred Lewis, Yusuf Simonson, Tzu-Wei Yang, Amir Yazdanbakhsh, Deniz Altinbüken, Florin Papa, Maggie Nolan Edmonds, Aditya Patil, Don Schwarz, Satish Chandra, Chris Kennelly, Milad Hashemi, and Parthasarathy Ranganathan. Eco: An llm-driven efficient code optimizer for warehouse scale computers, 2025. URL https://arxiv.org/abs/2503.15669.

Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. Evaluating language models for efficient code generation, 2024. URL https://arxiv.org/abs/2408.06450.

OpenAI. GPT-4o mini: Advancing cost-efficient intelligence. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/, July 2024. Accessed: 2026-09-22.

OpenAI. Introducing gpt-5. https://openai.com/index/introducing-gpt-5/, August 2025. Accessed: 2026-09-22.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition. https://openai.com/index/gpt-5-6/, July 2026. Accessed: 2026-09-22.

Yun Peng, Akhilesh Deepak Gotmare, Michael Lyu, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. Perfcodegen: Improving performance of llm generated code with execution feedback, 2024. URL https://arxiv.org/abs/2412.03578.

Python Software Foundation. The python profilers: cprofile and profile. https://docs.python.org/3/library/profile.html#module-cProfile. Accessed: 2026-09-22.

Ruizhong Qiu, Weiliang Will Zeng, James Ezick, Christopher Lott, and Hanghang Tong. How efficient is llm-generated code? a rigorous & high-standard benchmark, 2025. URL https://arxiv.org/abs/2406.06647.

Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. Gso: Challenging software optimization tasks for evaluating swe-agents, 2025. URL https://arxiv.org/abs/2505.23671.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits, 2024. URL https://arxiv.org/abs/2302.07867.

Siddhant Waghjale, Vishruth Veerendranath, Zora Zhiruo Wang, and Daniel Fried. Ecco: Can we improve model-generated code efficiency without sacrificing functional correctness?, 2024. URL https://arxiv.org/abs/2407.14044.

Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. Compilable neural code generation with compiler feedback. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (eds.), *Findings of the Association for Computational Linguistics: ACL 2022*, pp. 9–19, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-acl.2. URL https://aclanthology.org/2022.findings-acl.2/.

Tong Ye, Weigang Huang, Xuhong Zhang, Tengfei Ma, Peiyu Liu, Jianwei Yin, and Wenhai Wang. Llm4effi: Leveraging large language models to enhance code efficiency and correctness, 2025. URL https://arxiv.org/abs/2502.18489.

A Appendix

A.1 Similarity

A.1.1 Similarity Threshold

We use AST representations to measure the structural similarity between programs: each candidate is parsed with Python's `ast` module, after stripping comments and top-level assertions, and represented

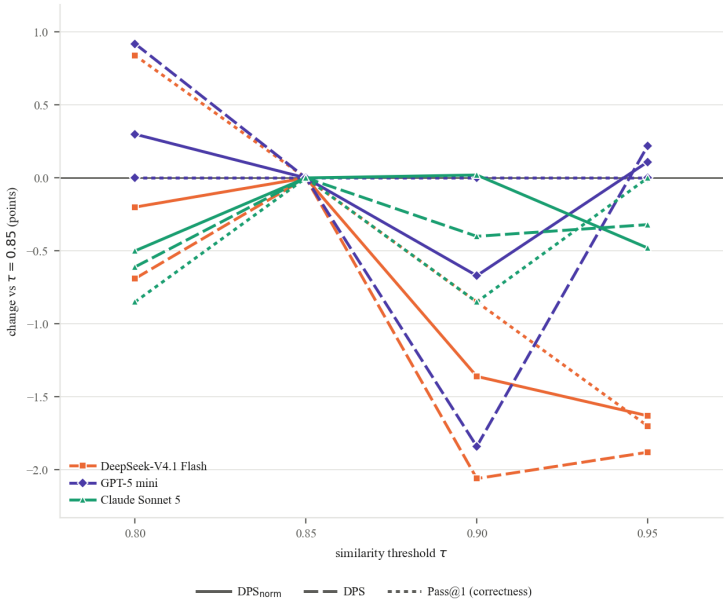

Figure 3: DPS, DPS_{norm}, and Pass@1 for various similarity thresholds on the EvalPerf benchmark.

as a 132-dimensional bag-of-node-types vector counting occurrences of every AST node class. These vectors are L_2-normalized and compared by cosine similarity, and the pairing window $[\tau, 1]$ is applied to that score. This representation is simple, inexpensive, and sufficient for capturing broad structural overlap without introducing heavy preprocessing. Figure 3 shows EFFIPAIR's performance with various τ values in different settings. Our default value, $\tau = 0.85$, seems to present a local optimum in terms of the balance between accuracy and efficiency across settings.

A.2 Profiling Method Analysis

A.2.1 Compressed Profiling Signal

We compact the profiler output to a short summary: the candidate function's own share of profiled CPU time – cProfile attributes time per function, so this is reported at the function's definition line – together with up to five of the hottest calls it makes, each accounting for at least 1% of profiled time. Profiler frames and harness scaffolding are excluded so they cannot appear as spurious hotspots.

A.2.2 Rationale for Compressed Profiling Signal

Our goal in profiling is not to recover perfectly precise line-by-line execution costs. Rather, we need a lightweight and reliable signal that identifies where most of the runtime is spent, so that refinement can focus on the most promising optimization opportunities. This is why a condensed profiler signal is sufficient for our setting.

Amdahl's Law. If a fraction p of total runtime is spent in one component, and that component is accelerated by a factor of k, then the overall speedup is

$$S_{\text{total}} = \frac{1}{(1-p) + \frac{p}{k}}.$$

Amdahl's law shows that the end-to-end benefit of an optimization depends primarily on how much of the total runtime is covered by the optimized component. In particular, larger values of p yield larger global gains. This formalizes the intuition to *optimize the hottest code first*: improving a small bottleneck cannot produce a large overall speedup, no matter how aggressively it is optimized.

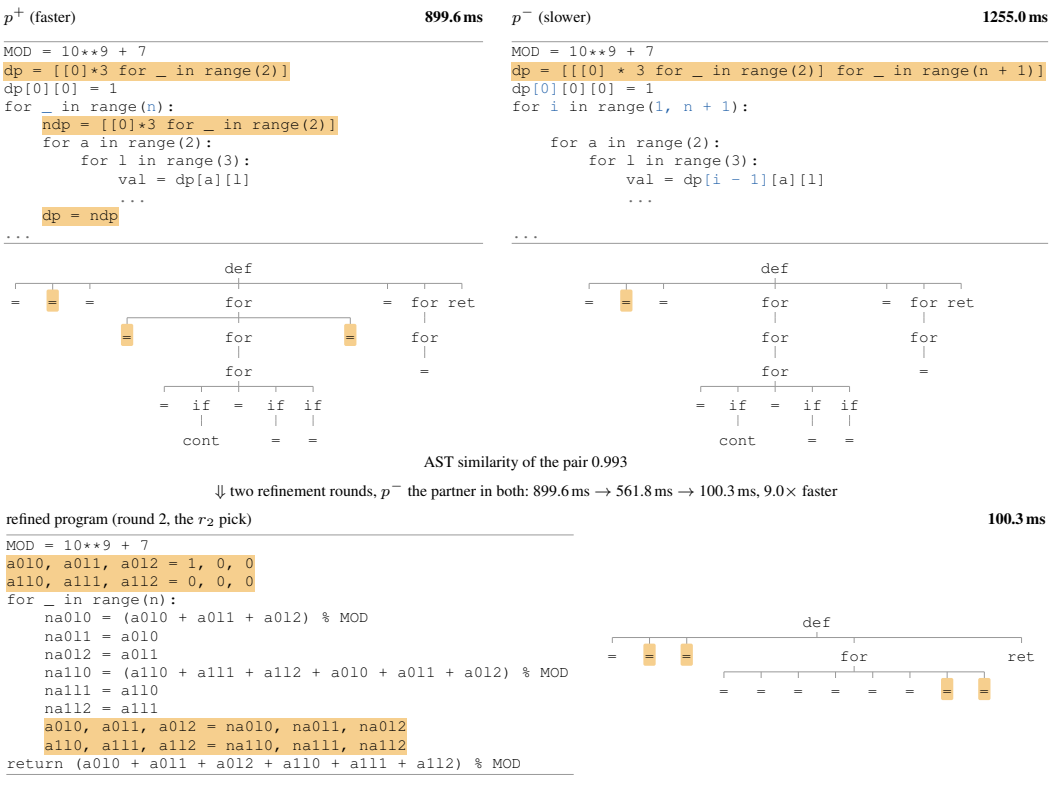

Figure 4: Mercury #552 (DeepSeek-V4.1 Flash): the round-1 pair and the final refined program, each with its statement-level AST. Up to cosmetic edits, the pair differs only in how the table is stored (shaded) and the indexing this forces (blue); the refined program keeps the six states in scalars. . . . marks omitted lines. Full case in Appendix A.4.

This observation is especially relevant for our use case. Since the refinement loop only needs to know which functions or lines dominate execution, approximate hotspot attribution is often enough.

We can also view each refinement step as a greedy optimization move. Let J denote the total execution cost. Choosing the hotspot with the largest estimated contribution to J is analogous to taking a local step in the direction of steepest decrease. Even when profiling provides only relative percentages rather than exact measurements, these percentages still supply a useful ordering over candidate bottlenecks.

A.3 Prompts

A.3.1 EffiPair Generation

EffiPair Generation (ENAMEL)

```
Complete the following function.

{prompt_stub}

Requirements:
- Keep the required function signature (top-level, no class).
- Return only a Python code block.
```

EffiPair Generation (EvalPerf)

```
{prompt_text}

Requirements:
```

```
- Implement the top-level function named '{entry_point}' (no class
    wrapper).
- Return only a Python code block.
```

EFFIPAIR Generation (Mercury)

```
Task:
{pretty_problem_content}

Code Stub:
{stub}

Requirements:
- Provide class Solution with the required method '{method_name}'.
- Return only a Python code block.
```

A.3.2 EFFIPAIR EFFICIENCY

```
Improve the following Python code with correctness as the top priority
    and efficiency as secondary.

Guidance:
{guidance}

{task_block}
Candidate A:
{code_a}

[optional] Profile A:
{profile_a}

[optional] Candidate B:

{code_b}

[optional] Profile B:
{profile_b}

Requirements:
{requirements}
```

A.3.3 EFFILEARNER GENERATION

HumanEval.

```
Please complete Python code based on the task description. # Task
    description:
{prompt}
#Solution:
```

MBPP.

```
Please complete Python code based on the task description and test
    cases. # Task description:
{prompt}
{tests}
#Solution:
```

A.3.4 EFFILEARNER EFFICIENCY

```
Optimize the efficiency of the following Python code based on the task,
     test case, and overhead analysis provided. Ensure the optimized
    code can pass the given test
case.

Task Description:
{task_description}

Test Case:
{test_case}

Original Code:

{completion}

Overhead Analysis:
{overhead_prompt}

Optimization Rules:

- Encapsulate the optimized code within a Python code block (i.e.,
    python\n[Your Code Here]\n).
- Do not include the test case within the code block.
- Focus solely on code optimization; test cases are already provided.
- Ensure the provided test case passes with your optimized solution.
```

A.4 Case study: Mercury task #552

This appendix follows one Mercury task, #552 (`student-attendance-record-ii`) as a representative example of how EffiPair turns differences between structurally similar programs into optimization guidance. The task uses the entry point `checkRecord` and is labeled LeetCode Hard and asks for the number of valid attendance sequences of length n with fewer than two absences and no three consecutive late days. Figure 4 summarizes the selected pair and the final refined implementation.

Each draw runs on Mercury's offline checker over the 8 test cases, and three clocks are recorded: the wall time of that correctness run, the workload time (in-process, candidate calls only, mean of 3 runs) and the profile time (the elapsed time under cProfile). In this run the pair is ordered by the wall time, the prompt reports the profile time and the committed pick uses the workload time. The naive draw enumerates all 3^n records, reaches the 30 s limit and is excluded from pairing (Table 4). The other two draws are the same dynamic program over (absences, current late streak): the *medium* draw keeps the full $(n{+}1) \times 2 \times 3$ table and the *fast* draw keeps two rolling 2×3 rows.

Table 4: Gate and clocks of the three draws (iteration 0).

draw	style	gate	wall (s)	workload (ms)	profile (ms)
`1d7bfd0f`	medium	ok	1.156	1085.6	1255.0
`eb72593a`	fast	ok	0.797	728.4	899.6
`09cc3033`	naive	timeout	30.031	–	–

A.4.1 Refinement

Round One. EffiPair selects the fastest correct candidate p^+ and the slowest correct candidate p^- within the similarity window. Here, $p^+ =$ `eb72593a` and $p^- =$ `1d7bfd0f`, with AST similarity 0.993. As shown in Figure 4, the programs implement the same six-state dynamic program and differ primarily in data layout: p^+ retains only two rolling 2×3 tables, whereas p^- stores the full $(n+1) \times 2 \times 3$ table. Their profile times are 899.6 ms and 1255.0 ms, respectively, providing a localized performance contrast between otherwise highly similar implementations. Conditioned on this pair and its execution feedback, the first refinement preserves the rolling-table representation but

removes the nested loops over the six states, replacing them with explicit state updates. The resulting candidate, `49b12e17`, passes the correctness gate and reduces profile time from 899.6 ms to 561.8 ms; its workload time decreases from 728.4 ms to 523.4 ms.

Round Two. The round-1 output becomes the new reference candidate p^+. The same slower candidate `1d7bfd0f` remains the selected partner p^-, with AST similarity 0.969. The second refinement further simplifies the representation by replacing the two 2×3 lists with six scalar state variables, eliminating per-iteration list allocation and indexing. This candidate, `a4522979`, also passes the correctness gate and reduces profile time from 561.8 ms to 100.3 ms, with workload time decreasing from 523.4 ms to 88.5 ms.

Across the two rounds, the selected implementation therefore progresses from a rolling-table dynamic program to explicit state transitions and finally to a scalar-state formulation. Profile time decreases from 899.6 ms to 100.3 ms while correctness is preserved. The example illustrates the intended role of DEF: the paired program supplies a nearby implementation contrast, while successive refinement can exploit that contrast to discover additional optimizations beyond simply copying the faster candidate.

Listing 1: Round-1 output `49b12e17` (the program inside the model's code block).

```python
class Solution(object):
    def checkRecord(self, n):
        MOD = 10**9 + 7

        # dp[a][l]
        # a = total absences so far: 0 or 1
        # l = current consecutive late streak length: 0, 1, or 2
        dp = [[0] * 3 for _ in range(2)]
        dp[0][0] = 1

        for _ in range(n):
            ndp = [[0] * 3 for _ in range(2)]

            # Unroll the 6 states to reduce loop overhead while preserving correctness.
            v = dp[0][0]
            if v:
                ndp[0][0] = (ndp[0][0] + v) % MOD       # P
                ndp[1][0] = (ndp[1][0] + v) % MOD       # A
                ndp[0][1] = (ndp[0][1] + v) % MOD       # L

            v = dp[0][1]
            if v:
                ndp[0][0] = (ndp[0][0] + v) % MOD       # P
                ndp[1][0] = (ndp[1][0] + v) % MOD       # A
                ndp[0][2] = (ndp[0][2] + v) % MOD       # L

            v = dp[0][2]
            if v:
                ndp[0][0] = (ndp[0][0] + v) % MOD       # P
                ndp[1][0] = (ndp[1][0] + v) % MOD       # A

            v = dp[1][0]
            if v:
                ndp[1][0] = (ndp[1][0] + v) % MOD       # P
                ndp[1][1] = (ndp[1][1] + v) % MOD       # L

            v = dp[1][1]
            if v:
                ndp[1][0] = (ndp[1][0] + v) % MOD       # P
                ndp[1][2] = (ndp[1][2] + v) % MOD       # L

            v = dp[1][2]
            if v:
                ndp[1][0] = (ndp[1][0] + v) % MOD       # P

            dp = ndp

        return (dp[0][0] + dp[0][1] + dp[0][2] + dp[1][0] + dp[1][1] + dp[1][2]) % MOD
```

A.4.2 Pool, committed picks and official score

Table 5 lists the whole pool. The committed pick after round r is the fastest candidate the pipeline marked correct, on the workload clock, among iterations $\leq r$. Each pick is scored with Mercury's procedure (median of 3 repeats, `PYTHONHASHSEED=0`, the 8 test cases), and Beyond@1 $= (t_{\max} - t)/(t_{\max} - t_{\min})$, clipped to $[0, 1]$, against the 16 reference runtimes (Table 6). The results tables report r_0 and r_2; the r_1 pick is included here for the round-by-round view. These three scores come from a separate scoring session, so their absolute runtimes differ from the table's by that session's

machine offset; the ordering is the point. The r_2 program is faster than 14 of the 16 reference solutions.

Table 5: Every entry of the pool with its clocks.

run	iter.	source	Cand. B	gate	wall (s)	workload (ms)	profile (ms)
`09cc3033`	0	naive draw	–	timeout	30.031	–	–
`1d7bfd0f`	0	medium draw	–	ok	1.156	1085.6	1255.0
`eb72593a`	0	fast draw	–	ok	0.797	728.4	899.6
`49b12e17`	1	refines `eb72593a`	`1d7bfd0f`	ok	0.593	523.4	561.8
`a4522979`	2	refines `49b12e17`	`1d7bfd0f`	ok	0.157	88.5	100.3

Table 6: Committed picks and their official Mercury scores.

pick	run	passes	runtime (s, median of 3)	Beyond@1
r_0	`eb72593a`	yes	0.3756	54.0
r_1	`49b12e17`	yes	0.2760	66.2
r_2	`a4522979`	yes	0.0508	94.0

A.4.3 Other repetitions and cost

Repetitions 1 and 2 are independent generations (seeds 136–138 and 146–148). In each repetition the naive draw enumerates all 3^n records and fails the gate (it reaches the 30 s limit in repetition 3 and fails in under 0.1 s in repetitions 1 and 2), and the other two draws are the same dynamic program. Two rounds take the best program from 0.90–1.26 s to 0.10–0.20 s on the profile clock (Table 7). In repetition 2 the pair again contrasts rolling rows with the full table; in repetition 1 both draws keep the full table. Table 8 gives the token cost of repetition 3.

Table 7: Profile time (ms) of the best correct program after each stage in the three repetitions, the partner (Candidate B) of both rounds, and the wall time (s) of the naive draw's gate run. *The repetition shown above.

rep.	best draw (r_0)	after round 1	after round 2	partner	naive (s)
1	`1cefd2c5` (medium) 1258.7	`bd58c82f` 999.2	`6b512ed5` 162.3	`aa9818e2`	0.071
2	`e543525f` (fast) 957.3	`31af93ac` 313.2	`c8225cda` 195.4	`0e89ba0d`	0.064
3*	`eb72593a` (fast) 899.6	`49b12e17` 561.8	`a4522979` 100.3	`1d7bfd0f`	30.031

Table 8: Tokens for this task in repetition 3.

stage	prompt	completion
generation, three draws (r_0)	1139	799
refinement round 1	1276	606
refinement round 2	1528	414
total (r_2)	3943	1819

A.5 Repository-Level Task Exclusion Details

We evaluate on 125 of SWE-Perf's 140 instances. Before running any arm we replayed the benchmark's gold patch three times per instance under the unmodified harness. Ten instances did not survive that check. On eight (`seaborn-2224`, `seaborn-2313`, `scikit-learn-13124`, `scikit-learn-12446`, `seaborn-2458`, `requests-1891`, `astropy-13734` and `scikit-learn-12246`) every repetition ran cleanly, but the expert patch measured a gain of 0.0% each time against the 2.9–20.8% the benchmark records, leaving no reference to compare an arm against. Two more, `sphinx-8537` and `sphinx-6422`, produced no timing at all in any repetition. The remaining five are limits of our patch builder rather than of

the benchmark, and all five reproduce normally. `sympy-27347`, `scikit-learn-12859` and `astropy-15228` carry an empty `patch_functions` field, so the oracle setting specifies no target functions to give the model. In `sympy-27666` and `scikit-learn-12567` every named target resolves to several definitions in its file, which our builder cannot splice unambiguously. No result in this paper is credited with any of the fifteen.