

# GLARE: Repository-Level Code Optimization Agents for Heterogeneous CPU-GPU Systems

Junfeng Gong<sup>1,2</sup>, Cheng Liu<sup>3\*</sup>, Junying Chen<sup>4</sup>, Xiaowei Li<sup>1,2</sup>, Huawei Li<sup>1,2</sup>

<sup>1</sup>SKLP, Institute of Computing Technology, Chinese Academy of Sciences

<sup>2</sup>School of Computer Science and Technology, University of Chinese Academy of Sciences

<sup>3</sup>School of Control and Computer Engineering, North China Electric Power University

<sup>4</sup>School of Software Engineering, South China University of Technology

\*Corresponding author: [liucheng@ncepu.edu.cn](mailto:liucheng@ncepu.edu.cn)

## Abstract

Recent LLM-based GPU code generation methods have shown promising results on CUDA and Triton kernels, but they typically operate on pre-isolated operators. Such a pre-isolated setting falls short in real-world CPU-GPU heterogeneous optimization, where the target computation is embedded in a larger repository and must be discovered, contextualized, selectively offloaded, and integrated. Such optimization requires repository-level context including data structure definitions, caller patterns, build configurations, and profiling signals, which is largely absent from operator-level tasks and benchmarks. In this work, we propose GLARE, a profiling-guided framework for repository-level code optimization targeting at heterogeneous CPU-GPU systems. GLARE anchors repository context in dynamic profiling signals, builds a multi-granular code dependency graph, extracts hotspot-centered active subgraphs, and uses an agentic loop to decide CPU-GPU partitioning and refine heterogeneous patches with execution feedback. On a GPU-acceleration subset of GSO with 63 tasks, GLARE improves OPT@1 from 74.6% to 87.3% over the strongest baseline under GPT-5.2, and achieves an 11.6% average end-to-end speedup. Ablation studies confirm that profiling-guided context construction is the primary driver of these gains.

## 1 Introduction

Recent advancements in LLM-based GPU code generation show significant promise in synthesizing CUDA and Triton kernels. As benchmarks like KernelBench and TritonBench evaluate the correctness and efficiency of these generated kernels (Ouyang et al., 2025; Li et al., 2025a), recent systems are pushing the boundaries of kernel synthesis through model specialization, reinforcement learning, and automated refinement (Zaeed et al., 2025; Lei et al., 2025; Fisches et al., 2025; Woo

et al., 2025; Li et al., 2025b). These methods typically operate in an operator-level setting, where the target computation is already separated from the surrounding program and the task is to generate a correct and efficient CUDA or Triton kernel. However, this pre-isolated setting abstracts away the repository-level challenges of real-world CPU-GPU heterogeneous optimization. In a production codebase, the computation to accelerate is often embedded in a larger software system. Hence, the agent must first discover performance hotspots, recover the data structures and caller patterns that determine input shapes and memory layout, reason about whether GPU offloading is worthwhile under data-transfer overhead, and integrate the generated kernel with host-side code, build scripts, tests, and fallback paths. These requirements depend on execution-driven repository context rather than merely on the local operators.

Existing work has studied parts of this problem in isolation, but does not bridge repository-level context construction with GPU kernel generation. Kernel-level methods such as Opal, PRAGMA, KernelLLM, and AutoTriton focus on generating CUDA or Triton code for pre-isolated hotspots, and therefore do not address hotspot localization, cross-file dependency recovery, or project-level integration (Zaeed et al., 2025; Lei et al., 2025; Ouyang et al., 2025; Fisches et al., 2025; Woo et al., 2025; Li et al., 2025b; Gong et al., 2026). In parallel, recent repository-level code agents and optimization benchmarks have enabled progress on realistic software engineering and performance tasks (Jimenez et al., 2024; Wang et al., 2025a; Xia et al., 2025; Yang et al., 2024; Shetty et al., 2025; He et al., 2025; Bai et al., 2025; Peng et al., 2026; Garg et al., 2025; Zhang et al., 2024), but they mainly target CPU-side rewrites or general performance fixes. Their context construction strategies largely rely on semantic retrieval or dependency-based expansion (Zhang et al., 2023; Shrivastava

et al., 2023; Liu et al., 2024; Phan et al., 2025), which work well when relevant code is textually or structurally close to the target location. In heterogeneous optimization, however, relevant context is determined by many factors such as runtime cost, hot functions, upstream callers, data loaders, memory layout, build configurations, and input characteristics, which may be scattered across the repository without textual similarity. As a result, repository-level CPU-GPU optimization remains an open challenge and it requires profiling-guided context construction, GPU offloading decisions, kernel generation, and execution-grounded integration in a single loop (Gong et al., 2025).

To this end, we propose **GLARE**, a profiling-guided framework for repository-level code optimization on heterogeneous CPU-GPU systems. GLARE’s core insight is that runtime profiling signals anchor optimization context more effectively than semantic similarity. Given a repository, GLARE builds a multi-granular code dependency graph (MCDG), identifies hotspot seeds from profiling traces, and performs bidirectional graph expansion with heuristic scoring and pruning to extract a compact performance-aware active subgraph that serves as the optimization context for LLM reasoning. Using this context, GLARE performs agentic CPU-GPU partitioning, generates CUDA or Triton kernels, and iteratively refines patches through compilation, correctness testing, and performance evaluation. Our contributions are summarized as follows:

- We empirically show that runtime-cost-aware context construction better captures optimization-relevant dependencies than file-level, function-only, and whole-repo context strategies.
- We propose GLARE, a profiling-guided optimization framework that achieves performance-aware repository understanding via multi-granular dependency graphs and active subgraph extraction, agentic CPU-GPU partitioning, kernel generation, and execution-grounded iterative refinement.
- We evaluate on 63 GPU-acceleration tasks from GSO, showing that GLARE outperforms existing baselines and that profiling-guided context construction is the key driver of the improvement.

## 2 Related Work

**Repository-Level Code Agents.** Recent LLM-based software engineering systems have moved from isolated code generation to repository-level modification. SWE-bench (Jimenez et al., 2024) evaluates whether models can resolve real GitHub issues, while agentic systems such as SWE-agent (Yang et al., 2024), AutoCodeRover (Zhang et al., 2024), and OpenHands (Wang et al., 2025a) interact with files, execute commands, localize bugs, edit code, and validate patches. Moreover, workflow-style methods such as Agentless (Xia et al., 2025) decompose repository-level repair into localization, editing, and validation stages. These works demonstrate the promise of repository-level agents, but primarily target functional correctness rather than performance optimization.

**Repository-Level Code Understanding and Retrieval.** Another line of work studies repository-level context retrieval for code generation. RepoCoder (Zhang et al., 2023) and RepoFusion (Shrivastava et al., 2023) retrieve or learn to use cross-file context for completion, while GraphCoder (Liu et al., 2024) and RepoHyper (Phan et al., 2025) exploit code graphs or semantic repository graphs to improve context selection. RLCoder further trains repository-level retrievers with reinforcement learning signals (Wang et al., 2025b), and CodeRAG improves retrieval-augmented repository completion through query construction, multi-path retrieval, and preference-aligned reranking (Zhang et al., 2025b). However, these methods mainly retrieve semantically or syntactically relevant context, without considering runtime behavior or performance-critical execution paths.

**LLMs for Code Performance Optimization.** Early LLM-based code optimization studies mainly focus on isolated programs or function-level rewriting. PIE introduces performance-improving edits as a benchmark for learning faster code transformations (Shypula et al., 2024). Subsequent methods study execution-feedback or search-based optimization, including EffiLearner, which uses execution overhead profiles for self-optimization (HUANG et al., 2024), PerfCodeGen, which incorporates runtime feedback into self-refinement (Peng et al., 2025), and SBLLM, which formulates code optimization as an iterative search process (Gao et al., 2025). Recent work also explores training paradigms for code acceleration, such as BITE’s

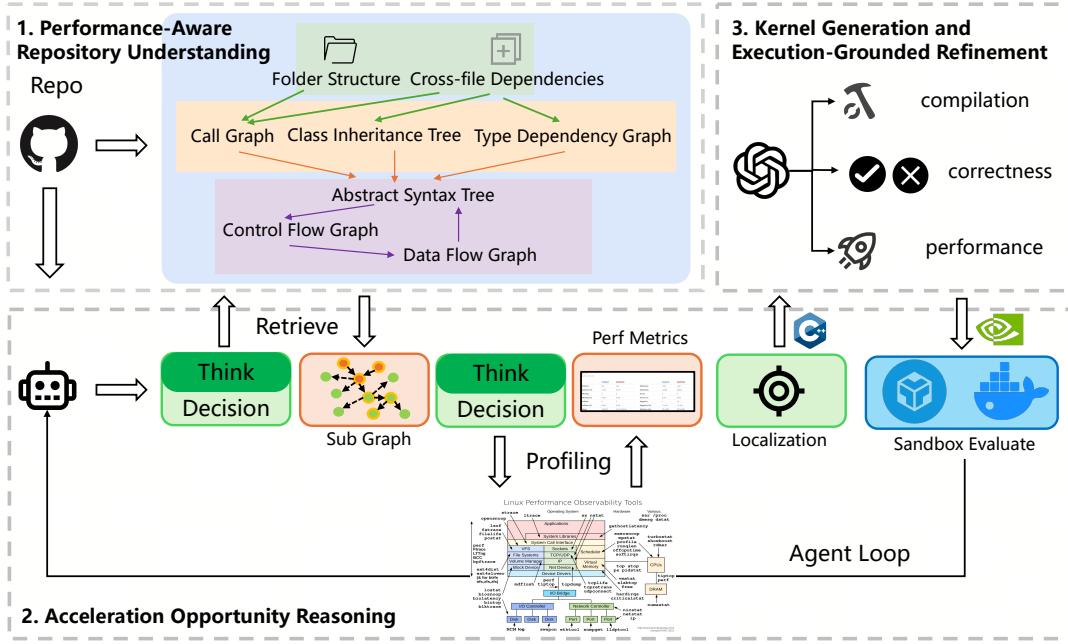


Figure 1: Overview of the proposed GLARE framework.

bidirectional tree editing (Zhang et al., 2025a), and efficiency-oriented generation frameworks such as LLM4Effi (Ye et al., 2025). Recent work has begun to study whether LLM agents can optimize real-world codebases. GSO (Shetty et al., 2025) introduces challenging repository-level software optimization tasks for evaluating SWE agents, and SWE-Perf (He et al., 2025) constructs performance optimization tasks from real performance-improving pull requests. These benchmarks reveal that existing agents still struggle with hotspot localization, cross-file reasoning, and generating patches with reliable speedup. POLO (Bai et al., 2025) further proposes a project-level LLM optimization framework that combines profiling, program structure analysis, and LLM-based rewriting. GLARE is closely related to this line of work, but focuses on CPU-GPU heterogeneous optimization, which additionally requires deciding whether a hotspot is suitable for GPU acceleration, reasoning about data transfer overhead, and integrating generated GPU code into the original repository.

### 3 GLARE

We propose **GLARE**, a profiling-guided, agentic code generation framework designed for repository-level optimization on heterogeneous CPU-GPU systems. As depicted in Figure 1, the GLARE pipeline begins by parsing an input software repository to construct a multi-granular code dependency graph, capturing structural, cross-file, and function-

level relationships. It then maps dynamic profiling signals onto this graph to isolate a compact active subgraph around performance-critical regions. Driven by this active subgraph, GLARE pinpoints suitable targets for GPU acceleration and formulates a comprehensive CPU-GPU partitioning plan. In the final stage, LLMs generate heterogeneous optimization patches, iteratively optimizing them through continuous feedback from compilation, correctness testing, and performance profiling.

Before context construction, GLARE runs the benchmark workload once with cProfile for Python or perf for C/C++. Execution time, call frequency, and accumulated cost are mapped to function-level MCDG nodes through symbol resolution and available source locations, making the profiles workload-dependent. These scores select hotspot seeds and bias graph expansion and pruning together with structural proximity, semantic importance, and redundancy penalty. Patched executions later provide additional traces for diagnosing low-speedup candidates.

#### 3.1 Code Repository Understanding

In repository-level optimization, relevant context inherently spans multiple levels of abstraction. Hotspot functions frequently depend on external data structures, modular helper functions, or upstream call chains that lack textual similarity to the hotspot, making standard semantic retrieval unreliable. To address this, GLARE introduces a two-

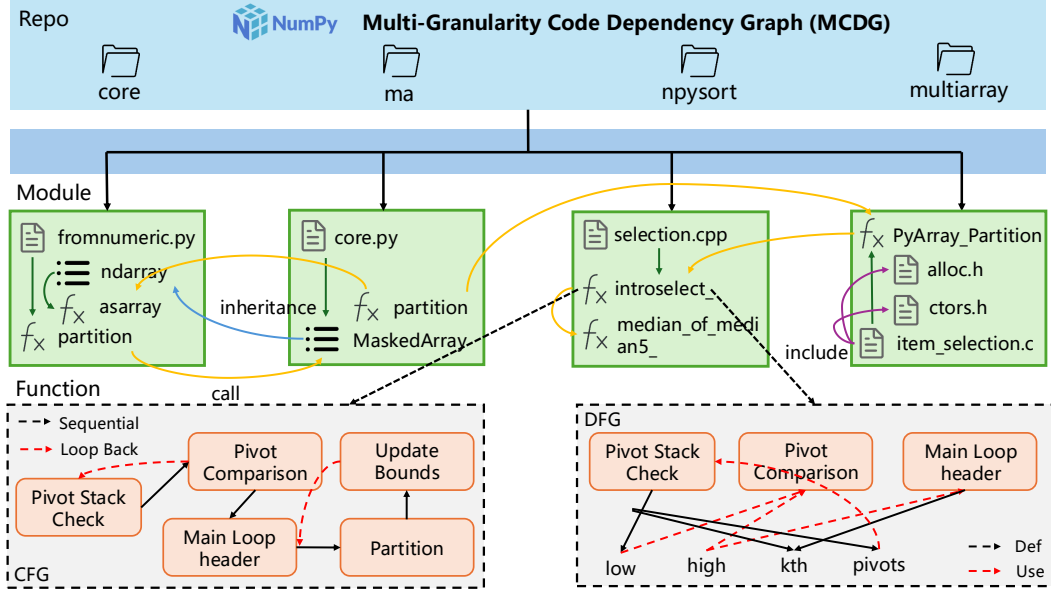


Figure 2: MCDG Construction. GLARE extracts code entities at repository, module, file, function, and optional block levels, and connects them with typed relations such as containment, import, call, reference, and local structural dependencies.

stage, performance-aware repository understanding mechanism. First, it establishes a multi-granular code dependency graph (MCDG) as a structured foundation. Second, it extracts a compact active subgraph encompassing profiling-identified hotspots. While drawing inspiration from prior methods that structure cross-file contexts using code graphs (Liu et al., 2024; Phan et al., 2025; Hu et al., 2025), GLARE uniquely anchors its graph expansion in dynamic profiling signals rather than semantic or lexical similarity.

### 3.1.1 MCDG Construction

Figure 2 illustrates the construction process of MCDG. Given a repository, GLARE first parses its directory hierarchy, source files, and configurations to obtain repository-, module-, and file-level entities. It then extracts function-level entities together with cross-file dependencies, including imports, function calls, and references to types, constants, and global variables. When fine-grained program structure is available, GLARE further incorporates local code regions such as loops and basic blocks. These entities and relations are finally unified into a directed typed graph, which captures both the hierarchical organization and semantic dependencies of the repository.

Formally, MCDG is a directed typed graph  $G = (V, E)$ , where nodes represent code entities at repository, module, file, function, and optional block levels, and edges represent containment, im-

port, call, reference, and local structural relations. This representation preserves both the hierarchical organization of the repository and the semantic dependencies required for downstream optimization.

The edge set  $\mathcal{E}$  contains several relations:

$$\mathcal{E} = \mathcal{E}_{contain} \cup \mathcal{E}_{import} \cup \mathcal{E}_{call} \cup \mathcal{E}_{ref} \cup \mathcal{E}_{local}.$$

Containment edges connect repository, module, file, function, and block-level nodes. Import edges represent file-level dependencies such as imports, includes, or package references. Call edges encode caller-callee relations. Reference edges connect functions to related data structures, types, constants, or global variables. Local edges optionally encode intra-function structure, such as control-flow or data-flow relations.

This graph provides a structured representation of the repository and serves as the substrate for context construction. Instead of passing entire files or flat retrieval results to the LLM, GLARE uses  $\mathcal{G}$  to reason about which code entities are likely to be relevant to the optimization target.

### 3.1.2 Active Subgraph Extraction

While the MCDG provides comprehensive repository structure, passing the entire graph to an LLM is infeasible under context-window constraints. The useful optimization context is typically a compact but dependency-rich region around performance-critical functions. GLARE extracts this region

through three steps including hotspot identification, bidirectional topology expansion, and heuristic pruning. Algorithm 1 and Figure 3 summarize the procedure.

**Hotspot Identification.** GLARE identifies hotspot seed nodes  $V_h = \{v \in V \mid \text{PERFScore}(v, P) \geq \theta\}$  from profiling results, where  $\text{PERFScore}(v, P)$  reflects each node’s runtime contribution (execution time, call frequency, accumulated cost). These seeds anchor subsequent graph expansion.

**Bidirectional topology expansion.** Optimizing hotspots often requires context beyond the hotspot itself: callers determine input shapes, callees contain reusable logic, and dependency nodes define data structures required for correct transformation. GLARE performs bidirectional topology expansion from  $V_h$  for up to  $d$  hops, collecting predecessors, successors, and dependency-related nodes to produce a candidate set  $V_c$ , with induced edges retained when both endpoints belong to  $V_c$ .

**Heuristic scoring and pruning.** To prune redundant nodes, GLARE scores each candidate  $v \in V_c$  with a weighted combination of four factors: profiling priority  $\phi_p(v)$  (runtime relevance), structural proximity  $\phi_s(v)$  (graph distance to seeds), semantic importance  $\phi_m(v)$  (utility for optimization), and a redundancy penalty  $\phi_r(v)$ :

$$\text{KEEPSCORE}(v) = \alpha\phi_p(v) + \beta\phi_s(v) + \gamma\phi_m(v) - \delta\phi_r(v).$$

Nodes with  $\text{KEEPSCORE}(v) \geq \tau$  are retained; hotspot seeds  $V_h$  are always kept. The active subgraph  $G_a = (V_a, E_a)$  is obtained by inducing edges  $E_a = \{(u, v) \in E \mid u, v \in V_a\}$  from the original graph.

The resulting active subgraph  $G_a$  provides a compact, performance-aware understanding of the repository: it excludes irrelevant code while retaining the entities and dependencies—hotspots, data structures, callers, callees, build configuration—that downstream stages require for acceleration opportunity reasoning and kernel generation.

### 3.2 Acceleration Opportunity Reasoning

With this performance-aware repository understanding, GLARE further decides how each candidate hotspot should be optimized. This step is

#### Algorithm 1 Performance-aware Active Subgraph Construction

---

**Require:** Multi-granular code dependency graph  $G = (V, E)$ , profiling results  $P$ , hotspot threshold  $\theta$ , expansion depth  $d$ , pruning threshold  $\tau$

**Ensure:** Active subgraph  $G_a = (V_a, E_a)$

*// Hotspot seed extraction*

1:  $V_h \leftarrow \{v \in V \mid \text{PERFScore}(v, P) \geq \theta\}$

*// Bidirectional topology expansion*

2:  $V_c \leftarrow V_h$

3: **for**  $l = 1$  to  $d$  **do**

4:    $V_{\text{new}} \leftarrow \emptyset$

5:   **for all**  $v \in V_c$  **do**

6:      $V_{\text{new}} \leftarrow V_{\text{new}} \cup \text{PRED}(v) \cup \text{SUCC}(v) \cup \text{DEP}(v)$

7:   **end for**

8:    $V_c \leftarrow V_c \cup V_{\text{new}}$

9: **end for**

*// Heuristic scoring and pruning*

10:  $V_a \leftarrow \emptyset$

11: **for all**  $v \in V_c$  **do**

12:    $\phi_p(v) \leftarrow \text{PERFPRIORITY}(v, P)$

13:    $\phi_s(v) \leftarrow \text{STRUCTPROXIMITY}(v, V_h)$

14:    $\phi_m(v) \leftarrow \text{SEMANTICIMPORTANCE}(v)$

15:    $\phi_r(v) \leftarrow \text{REDUNDANCYPENALTY}(v, V_c)$

16:    $\text{KEEPSCORE}(v) \leftarrow \alpha\phi_p(v) + \beta\phi_s(v) + \gamma\phi_m(v) - \delta\phi_r(v)$

17:   **if**  $\text{KEEPSCORE}(v) \geq \tau$  **then**

18:      $V_a \leftarrow V_a \cup \{v\}$

19:   **end if**

20: **end for**

21:  $V_a \leftarrow V_a \cup V_h$

22:  $E_a \leftarrow \{(u, v) \in E \mid u \in V_a \wedge v \in V_a\}$

23: **return**  $G_a = (V_a, E_a)$

---

crucial because a runtime hotspot is not necessarily a profitable GPU offloading target. Some regions dominate execution time due to sequential dependencies, irregular memory accesses, complex control flow, or frequent small-scale host-side operations, where direct GPU migration may introduce more overhead than benefit. Therefore, GLARE formulates hardware–software partitioning as an agentic planning problem over the active subgraph.

Given an active subgraph  $G_a = (V_a, E_a)$ , profiling results  $P$ , and a set of candidate hotspot regions  $\mathcal{H} = \{h_i\}$ , GLARE predicts a partitioning label for each candidate:

$$y_i \in \{\text{GPU}, \text{CPU}, \text{HYBRID}\},$$

where GPU denotes regions suitable for CUDA acceleration, CPU denotes regions better optimized on the host side, and HYBRID denotes regions that should be split into GPU-accelerated compute kernels and CPU-side orchestration logic. In addition to the partition label, GLARE also produces an evidence summary that explains the decision, including profiling signals, dependency context, parallelization opportunities, and potential integration risks.

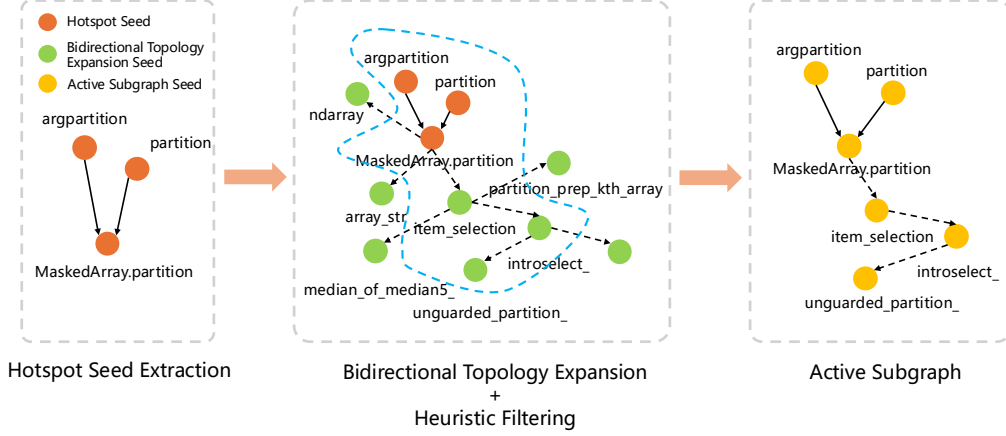


Figure 3: Performance-aware repository understanding via active subgraph extraction. GLARE combines the multi-granular code dependency graph (MCDG) with dynamic profiling results to identify hotspot-centered regions, expand dependency-relevant context, and filter low-value nodes, yielding a compact, performance-aware representation of the repository for CPU-GPU heterogeneous optimization.

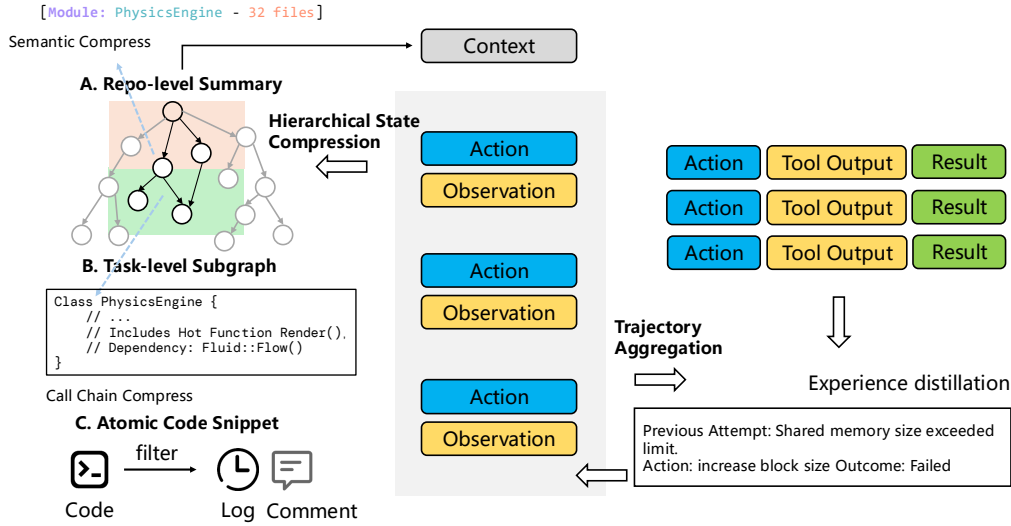


Figure 4: Iterative process of performance bottleneck localization by the GLARE agent in complex code repositories.

**Iterative agentic planning.** As shown in Figure 4 and detailed in Algorithm 2 in Appendix C, GLARE does not rely on a single-pass decision over the candidate hotspot. Instead, it performs an iterative agentic planning process that repeatedly analyzes repository context, invokes diagnostic tools, updates its hypothesis, and refines the localization result. At iteration  $t$ , the agent maintains a compact state  $s_t$  and generates a hypothesis about the current candidate region, such as whether the hotspot is caused by regular data-parallel computation or by host-side control overhead. It then selects an analysis action  $a_t$ , invokes static or dynamic analysis tools, receives an observation  $o_t$ ,

and updates its state:

$$a_t \sim \pi_\theta(s_t), \quad s_{t+1} = \text{UPDATE}(s_t, a_t, o_t).$$

The action space includes inspecting call relations, checking loop structures, analyzing memory access patterns, querying profiling statistics, estimating data transfer cost, and examining interface dependencies. Through this feedback-driven process, GLARE gradually narrows down the candidate region and forms a more reliable hardware–software partitioning decision.

### 3.3 Kernel Generation and Execution-Grounded Refinement

Given the partition plan  $\mathcal{P}$  and the active subgraph  $G_a$ , GLARE generates heterogeneous optimization patches for each hotspot labeled GPU or HYBRID. The generation is context-grounded: the LLM receives the hotspot source, data structure definitions, caller context, the evidence summary  $e_i$ , and relevant build configuration fragments extracted from  $G_a$ . For each hotspot, the LLM produces a patch containing the GPU kernel (CUDA/Triton), host-side integration code (memory transfer, kernel launch), and build file modifications. Hotspots labeled HYBRID additionally receive a split that keeps orchestration logic on the host while offloading compute-intensive inner loops to the GPU.

Each generated patch must pass three sequential verification gates before it is accepted. GLARE applies the patch, then executes:

**Compilation gate.** The patched repository is built with the same system. If compilation fails, the full compiler error output is fed back to the LLM for revision, bounded by  $K_c$  retries.

**Correctness gate.** If compilation succeeds, GLARE runs the project’s test suite. Test failures (assertion messages, stack traces) are fed back to the LLM for revision, bounded by  $K_t$  retries.

**Performance gate.** If correctness tests pass, GLARE executes the benchmark workload and measures runtime ( $R$  repetitions, arithmetic mean). If speedup falls below threshold  $\tau$ , the profiling trace of the patched execution is fed back to the LLM to diagnose and resolve the bottleneck (e.g., data transfer overhead, insufficient parallelism), bounded by  $K_p$  retries.

After exhausting all retry budgets, GLARE selects the patch with the highest verified speedup that passes both compilation and correctness gates. Tasks where no patch passes all three gates are reported as optimization failures.

## 4 Experiments

### 4.1 Experimental Setup

**Benchmark and Baselines.** We evaluate on a GPU-acceleration subset of GSO (Shetty et al., 2025), containing 63 tasks across 10 repositories (numpy, pandas, llama-cpp, and others). Tasks are selected for compute-intensity and GPU-offloading suitability; detailed criteria and repos-

itory statistics are provided in Appendix A. We compare GLARE against two baselines representing different paradigms of repository-level code modification: **Agentless** (Xia et al., 2025), a workflow-driven localization–repair–validation pipeline augmented with profiling-based bottleneck localization and CUDA-aware prompts; and **OpenHands** (Wang et al., 2025a), an agent-driven baseline that interactively explores the repository, inspects files, executes commands, and iteratively modifies code. All methods use the same backbone LLMs, repository access, and evaluation command.

**Backbone Models and Metrics.** We evaluate with three backbone LLMs spanning different capability levels: Claude-Opus-4-6, GPT-5.2, and DeepSeek-V3. Hardware configuration is detailed in Appendix A. We report patch apply rate (proportion of patches successfully applied), OPT@1 and OPT@5 (proportion of tasks where at least one of the top-1 or top-5 patches passes correctness tests and achieves speedup  $\geq 1.05\times$  over the original), and average end-to-end speedup (mean relative performance improvement over successfully optimized tasks).

### 4.2 Main Results

Table 1 presents the main results. GLARE consistently and substantially outperforms both baselines across all three backbone models. Several findings are noteworthy. (1) GLARE achieves the highest OPT@1 and OPT@5 across all models and metrics. With GPT-5.2, GLARE attains 87.3% OPT@1 and 90.5% OPT@5, improving over the best baseline (OpenHands) by 12.7% and 8.0% respectively. With Claude-Opus-4-6, GLARE reaches 84.1% OPT@1 and 11.2% average speedup. The gains are consistent across models: even with the weaker DeepSeek-V3, GLARE achieves 76.2% OPT@1 compared to 63.5% for OpenHands, demonstrating that GLARE’s structured context construction provides benefits complementary to raw model capability. (2) The performance gain remains stable across model scales. GLARE improves OPT@1 over OpenHands 12.7% across all three backbone models, suggesting that profiling-guided context construction provides benefits complementary to model capability. (3) The patch apply rate reveals a systematic gap in engineering robustness. GLARE achieves 74.6%–77.8% patch apply rates compared to 57.1%–63.5% for Agentless and 52.4%–63.5% for OpenHands. Since all methods attempt to gen-

Table 1: End-to-end performance comparison of GLARE and baseline methods on the GSO benchmark

| Model           | Method    | Patch Apply Rate | OPT@K (%) |       | Avg. End-to-End Speedup |
|-----------------|-----------|------------------|-----------|-------|-------------------------|
|                 |           |                  | OPT@1     | OPT@5 |                         |
| Claude-Opus-4-6 | Agentless | 57.1%            | 66.7%     | 74.6% | 8.3%                    |
|                 | OpenHands | 60.3%            | 71.4%     | 79.4% | 9.5%                    |
|                 | GLARE     | 74.6%            | 84.1%     | 87.3% | 11.2%                   |
| GPT-5.2         | Agentless | 60.3%            | 68.3%     | 76.2% | 8.6%                    |
|                 | OpenHands | 63.5%            | 74.6%     | 82.5% | 9.8%                    |
|                 | GLARE     | 77.8%            | 87.3%     | 90.5% | 11.6%                   |
| DeepSeek-V3     | Agentless | 47.6%            | 57.1%     | 66.7% | 6.2%                    |
|                 | OpenHands | 52.4%            | 63.5%     | 71.4% | 6.9%                    |
|                 | GLARE     | 66.7%            | 76.2%     | 79.4% | 7.4%                    |

erate patches for the same tasks, this gap reflects differences in patch quality: Agentless and OpenHands more frequently produce patches with merge conflicts, incorrect file paths, or malformed diffs—issues that GLARE mitigates by grounding generation in the active subgraph, which explicitly includes file-level and build-configuration context.

### 4.3 Ablation Study

To understand why GLARE outperforms existing approaches, we conduct ablation experiments that isolate the contributions of context construction strategy and agent decision mechanisms. All ablations use GPT-5.2 as the backbone.

#### 4.3.1 Context Construction Strategy

The central claim of this paper is that profiling-guided graph expansion provides more effective optimization context than alternative context construction strategies. To isolate this effect from hotspot detection accuracy, we adopt an **Oracle hotspot** setting: all methods are given the ground-truth hotspot function(s) for each task, and the comparison focuses purely on how each strategy constructs the supporting context around the known hotspot.

We compare GLARE’s full active subgraph construction (**Full MCDG + Active Subgraph**) against three alternative strategies: **File-level Retrieval**, which gathers all functions from the hotspot’s file and directly imported files without graph-based pruning; **Function-only Context**, which retains only the hotspot function body with no cross-file dependencies; and **Whole-repo Context**, which concatenates the entire repository (up to 32 K tokens) without filtering.

Table 2 reports the results. GLARE achieves the best balance: 92.4% Support Recall and 89.5% Path Coverage with 83.7% Precision in a compact

6.0k-token context, yielding the highest Oracle-OPT@1 (90.5%). Whole-repo Context achieves higher recall (96.1%) but at 24.7% precision and triple the context size (19.4k), degrading OPT@1 to 82.5%—more information is not better when it is mostly noise. Function-only Context is precise (88.1%) but starved (61.2% recall, 2.3k tokens), while File-level Retrieval misses cross-file dependency chains beyond direct imports (75.4% path coverage). These trends suggest that support recall and path coverage are more informative than raw context size for predicting optimization success.

#### 4.3.2 Agent Decision Mechanisms

We further ablate two mechanisms in GLARE’s acceleration opportunity reasoning stage: **Hierarchical State Compression** (HSC), which compresses multi-turn exploration history into hierarchical summaries to control per-turn context size, and **Trajectory Aggregation** (TA), which records analyzed regions and prior conclusions to avoid redundant re-exploration. We compare GLARE (Full) against four variants: removing HSC, removing TA, removing both, and Flat History (retaining all history without hierarchical organization).

Figure 5 shows that HSC mainly controls context growth, while TA reduces redundant exploration. Removing either component degrades both decision efficiency and optimization success, and removing both causes the largest drop. Flat History also underperforms GLARE, indicating that simply retaining more history is insufficient; the history must be compressed and organized to support multi-turn reasoning.

## 5 Conclusion

We presented GLARE, a profiling-guided framework for repository-level CPU-GPU heterogeneous optimization. GLARE grounds repository context

Table 2: Comparison of support-context quality and optimization effectiveness of different repository understanding strategies under the Oracle hotspot setting

| Metric                             | Full MCDG + Active Subgraph | File-level Retrieval | Function-only Context | Whole-repo Context |
|------------------------------------|-----------------------------|----------------------|-----------------------|--------------------|
| Support Recall (%) $\uparrow$      | 92.4                        | 84.6                 | 61.2                  | 96.1               |
| Support Precision (%) $\uparrow$   | 83.7                        | 68.9                 | 88.1                  | 24.7               |
| Path Coverage (%) $\uparrow$       | 89.5                        | 75.4                 | 56.8                  | 92.1               |
| Avg. Context Tokens $\downarrow$   | 6.0k                        | 8.1k                 | 2.3k                  | 19.4k              |
| Oracle-OPT@1 (%) $\uparrow$        | 90.5                        | 85.7                 | 77.8                  | 82.5               |
| Conditional Speedup (%) $\uparrow$ | 11.9                        | 10.8                 | 9.5                   | 10.2               |

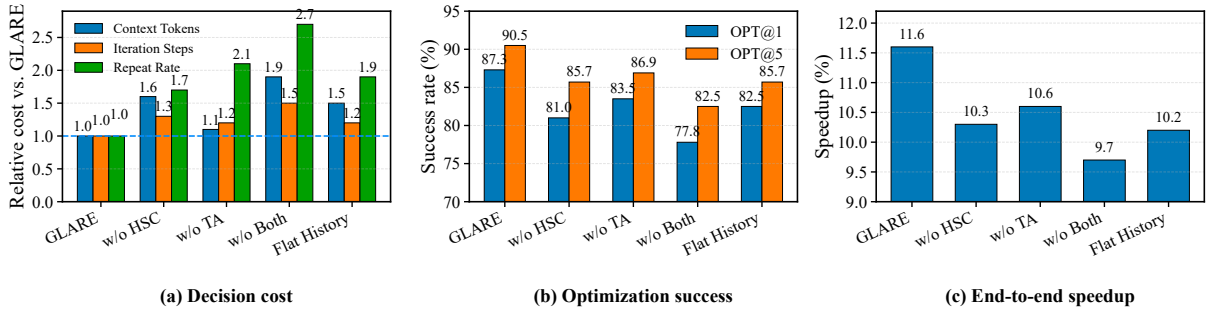


Figure 5: Relative degradation of efficiency and optimization outcomes when removing HSC and TA. GLARE (Full) serves as the baseline.

construction in runtime profiling signals, extracts hotspot-centered active subgraphs from multi-granular code dependency graphs, and uses them for agentic partitioning and execution-grounded patch refinement. Experiments on 63 GPU-acceleration tasks from GSO show that GLARE outperforms workflow-based and agent-based baselines, with ablations confirming the importance of profiling-guided context construction. These results suggest that performance-aware context selection is a promising direction for LLM-based code optimization.

## Limitations

While GLARE demonstrates the value of profiling-guided context construction for heterogeneous optimization, several limitations should be noted. First, GLARE’s reliance on dynamic profiling traces means that its context quality depends on workload representativeness: if the profiling workload exercises different execution paths than the deployment workload, the active subgraph may miss performance-critical context and misdirect the partitioning stage. Second, the current verification loop uses pure feedback-driven retry without structured error categorization; integrating compiler-level or runtime-level diagnostic tools could enable more targeted revision and reduce the retry budget needed for convergence. Third, GLARE

currently targets a single-GPU setting and does not address multi-GPU partitioning or distributed execution, which limits its applicability to large-scale, multi-node scientific computing workloads. Fourth, our evaluation is conducted on a GPU-acceleration subset of GSO curated by the authors; while we apply systematic selection criteria, the benchmark construction process may introduce biases that affect generalizability. Finally, GLARE inherits the known limitations of LLM-based code generation, including the potential for generating plausible-looking but semantically incorrect code, and the sensitivity of generation quality to prompt design and model capability. We release our benchmark subset and evaluation scripts to support reproducibility and future work.

## Acknowledgments

This work was supported in part by the Strategic Priority Research Program of the Chinese Academy of Sciences under Grant XDB0660103; the National Natural Science Foundation of China under Grants 62674224 and U2570205; the State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences under Grants CLQ202601 and CLQ202403; and the Guangdong Basic and Applied Basic Research Foundation under Grant 2024A1515030128.

## References

- Muhammad Awad, Cole Ramos, and Keith Lowery. 2025. [IntelliPerf: LLM-powered autonomous GPU performance engineer](#). Software release.
- Jiameng Bai, Ruoyi Xu, Sai Wu, Dingyu Yang, Junbo Zhao, and Gang Chen. 2025. Polo: An llm-powered project-level code performance optimization framework. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, pages 7319–7328.
- Zacharias V. Fisches, Sahan Paliskara, Simon Guo, Alex Zhang, Joe Spisak, Chris Cummins, Hugh Leather, Gabriel Synnaeve, Joe Isaacson, Aram Markosyan, and Mark Saroufim. 2025. [Kernelllm: Making kernel development more accessible](#).
- Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael R. Lyu. 2025. [Search-based llms for code optimization](#). In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, page 578–590. IEEE Press.
- Spandan Garg, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. 2025. [Perfbench: Can agents resolve real-world performance bugs?](#) *Preprint*, arXiv:2509.24091.
- Jingzhi Gong, Vardan Voskanyan, Paul Brookes, Fan Wu, Wei Jie, Jie Xu, Rafail Giavrimis, Mike Basios, Leslie Kanthan, and Zheng Wang. 2025. [Language models for code optimization: Survey, challenges and future directions](#). *Preprint*, arXiv:2501.01277.
- Junfeng Gong, Zhiyi Wei, Junying Chen, Cheng Liu, and Huawei Li. 2026. [From large to small: Transferring cuda optimization expertise via reasoning graph](#). In *International Conference on Learning Representations*, volume 2026, pages 121082–121107.
- Xinyi He, Qian Liu, Mingzhe Du, Lin Yan, Zhijie Fan, Yiming Huang, Zejian Yuan, and Zejun Ma. 2025. [Swe-perf: Can language models optimize code performance on real-world repositories?](#) *Preprint*, arXiv:2507.12415.
- Yuntong Hu, Zhihan Lei, Zheng Zhang, Bo Pan, Chen Ling, and Liang Zhao. 2025. [Grag: Graph retrieval-augmented generation](#). *Preprint*, arXiv:2405.16506.
- Dong HUANG, Jianbo Dai, Han Weng, Puzhen Wu, Yuhao QING, Heming Cui, Zhijiang Guo, and Jie Zhang. 2024. [Effilearner: Enhancing efficiency of generated code via self-optimization](#). In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. [SWE-bench: Can language models resolve real-world github issues?](#) In *The Twelfth International Conference on Learning Representations*.
- Kelun Lei, Hailong Yang, Huaitao Zhang, Xin You, Kaige Zhang, Zhongzhi Luan, Yi Liu, and Depei Qian. 2025. [Pragma: A profiling-reasoned multi-agent framework for automatic kernel optimization](#). *Preprint*, arXiv:2511.06345.
- Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, Jianrong Wang, Xu Han, Zhiyuan Liu, and Maosong Sun. 2025a. [TritonBench: Benchmarking large language model capabilities for generating triton operators](#). In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 23053–23066, Vienna, Austria. Association for Computational Linguistics.
- Shangzhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, Jianling Li, Yonggang Hu, Wanxiang Che, Xu Han, Zhiyuan Liu, and Maosong Sun. 2025b. [Autotriton: Automatic triton programming with reinforcement learning in llms](#). *Preprint*, arXiv:2507.05687.
- Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. 2024. [Graphcoder: Enhancing repository-level code completion via coarse-to-fine retrieval based on code context graph](#). In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 570–581, New York, NY, USA. Association for Computing Machinery.
- Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. [Kernelbench: Can llms write efficient gpu kernels?](#) In *Proceedings of the 42nd International Conference on Machine Learning*.
- Huiyun Peng, Arjun Gupte, Ryan Hasler, Nicholas John Eliopoulos, Chien-Chou Ho, Rishi Mantri, Leo Deng, Konstantin Läufer, George K. Thiruvathukal, and James C. Davis. 2026. [Sysllmatic: Large language models are software system optimizers](#). *Preprint*, arXiv:2506.01249.
- Yun Peng, Akhilesh Deepak Gotmare, Michael R. Lyu, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2025. [Perfcodegen: Improving performance of llm generated code with execution feedback](#). In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*, pages 1–13.
- Huy Nhat Phan, Hoang Nhat Phan, Tien N. Nguyen, and Nghi D. Q. Bui. 2025. [Repohyper: Search-expand-refine on semantic graphs for repository-level code completion](#). In *Proceedings of the 2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (FORGE)*, pages 14–25.
- Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. 2025. [Gso: Challenging software optimization tasks for evaluating swe-agents](#). In *Advances in Neural Information Processing Systems: Datasets and Benchmarks Track*.

- Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. 2023. [Re-pofusion: Training code models to understand your repository](#). *Preprint*, arXiv:2306.10998.
- Alexander G Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2024. [Learning performance-improving code edits](#). In *The Twelfth International Conference on Learning Representations*.
- Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2025a. [Openhands: An open platform for AI software developers as generalist agents](#). In *The Thirteenth International Conference on Learning Representations*.
- Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2025b. [RLcoder: Reinforcement learning for repository-level code completion](#). In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, page 1140–1152. IEEE Press.
- Jiin Woo, Shaowei Zhu, Allen Nie, Zhen Jia, Yida Wang, and Youngsuk Park. 2025. [Tritonrl: Training llms to think and code triton without cheating](#). *Preprint*, arXiv:2510.17891.
- Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. [Demystifying llm-based software engineering agents](#). *Proc. ACM Softw. Eng.*, 2(FSE).
- John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: agent-computer interfaces enable automated software engineering. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA. Curran Associates Inc.
- Tong Ye, Weigang Huang, Xuhong Zhang, Tengfei Ma, Peiyu Liu, Jianwei Yin, and Wenhai Wang. 2025. [Llm4effi: Leveraging large language models to enhance code efficiency and correctness](#). *Preprint*, arXiv:2502.18489.
- Mohammad Zaeed, Tanzima Z. Islam, and Vladimir Indić. 2025. [Opal: A modular framework for optimizing performance using analytics and llms](#). *Preprint*, arXiv:2510.00932.
- Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. [RepoCoder: Repository-level code completion through iterative retrieval and generation](#). In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore. Association for Computational Linguistics.
- Longhui Zhang, Jiahao Wang, Meishan Zhang, Gaoxiong Cao, Ensheng Shi, Yuchi Ma, Jun Yu, Honghai Liu, Jing Li, and Min Zhang. 2025a. [Speed up your code: Progressive code acceleration through bidirectional tree editing](#). In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 28563–28576, Vienna, Austria. Association for Computational Linguistics.
- Sheng Zhang, Yifan Ding, Shuquan Lian, Shun Song, and Hui Li. 2025b. [CodeRAG: Finding relevant and necessary knowledge for retrieval-augmented repository-level code completion](#). In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 23278–23288, Suzhou, China. Association for Computational Linguistics.
- Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. [Autocoderover: Autonomous program improvement](#). In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 1592–1604, New York, NY, USA. Association for Computing Machinery.

## A Detailed Experimental Setup

**Hardware Configuration.** We conduct all repository-level end-to-end optimization experiments on a single workstation with an Intel Xeon CPU, 128 GB DDR4 host memory, and an NVIDIA RTX 5090 GPU based on the Blackwell architecture with 32 GB VRAM. The software stack consists of CUDA 13.0, GCC 13.1.0, and NVCC 12.9. All baseline methods and GLARE are executed under identical hardware and software conditions to ensure fair comparison.

**Benchmark Construction.** We construct the GPU-acceleration subset of GSO by applying the following selection criteria: (1) the performance bottleneck is compute-intensive rather than I/O-bound; (2) the target code region exhibits data-parallel computation patterns amenable to GPU offloading; (3) the repository provides executable correctness and performance evaluation scripts; (4) the task is not dominated by caching, configuration, database indexing, or network-level optimization. Task filtering was performed before running any evaluated method, and was not based on the success or failure of GLARE or baselines. Table 3 reports the structural statistics of each repository after MCDG construction.

**Baseline Configuration.** **Agentless** is augmented with a profiling-based localization step (using the same profiling traces available to GLARE)

Table 3: Structural statistics of each repository in GSO after MCDG construction

| Repository       | #Nodes  | #Edges  | #Func./Meth. | #Nested Loops | Index Time (s) |
|------------------|---------|---------|--------------|---------------|----------------|
| transformers     | 332,756 | 630,894 | 94,420       | 3,658         | 36.1           |
| pandas           | 159,731 | 356,834 | 49,348       | 469           | 15.4           |
| numpy            | 120,337 | 330,862 | 36,007       | 1,857         | 12.4           |
| pydantic         | 30,287  | 48,057  | 8,118        | 90            | 1.5            |
| datasets         | 19,151  | 34,647  | 5,277        | 208           | 1.6            |
| tornado          | 21,396  | 36,634  | 6,543        | 47            | 1.7            |
| Pillow           | 18,299  | 38,475  | 5,431        | 245           | 3.3            |
| pillow-simd      | 16,652  | 34,439  | 4,913        | 236           | 0.9            |
| llama-cpp-python | 12,024  | 22,154  | 3,320        | 220           | 0.8            |
| tokenizers       | 2,626   | 3,804   | 768          | 15            | 0.2            |

and a CUDA-aware repair prompt that instructs the LLM to consider GPU offloading opportunities during patch generation. **OpenHands** is configured with file browsing, command execution, and code editing capabilities; it receives the same task description, repository path, and evaluation scripts as GLARE. For all methods, the maximum context window is set to the backbone model’s native limit.

**Evaluation Protocol.** For each candidate patch, we execute the following pipeline: (1) apply the patch to the repository using `git apply`; (2) build the patched repository using the project’s build system; (3) run the correctness test suite; (4) if all tests pass, execute the benchmark workload and measure runtime. Each runtime measurement is repeated  $R = 5$  times after 3 warm-up runs, and we report the arithmetic mean. Speedup is computed as the ratio of mean original runtime to mean patched runtime. **OPT@1** and **OPT@5** require a speedup  $\geq 1.05\times$  to count as successful. The intermediate metrics in Table 1 are defined as follows: patch apply rate = proportion of generated patches successfully applied; OPT@1 / OPT@5 = proportion of tasks where at least one candidate among top-1 / top-5 passes all tests and achieves the required speedup.

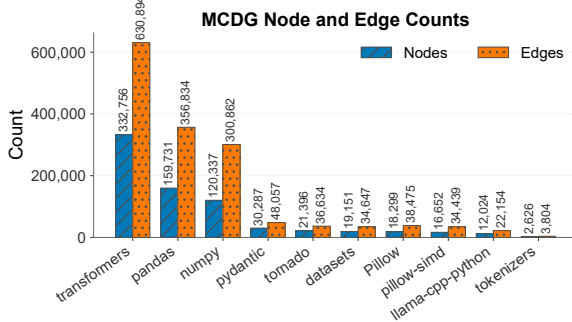


Figure 6: Distribution of the numbers of nodes and edges in MCDG across different repositories

## Implementation and Hyperparameters.

GLARE’s MCDG is constructed using tree-sitter for AST-based parsing. Static call graphs, import relationships, and type references are extracted via static analysis of each repository and unified into the typed graph  $\mathcal{G}$ . Runtime profiles are collected by executing the benchmark workload with standard profiling tools (cProfile for Python, perf for C/C++) and mapped to function-level graph nodes via symbol resolution. Active subgraph construction uses expansion depth  $d = 3$ , scoring weights  $\alpha = 0.4$ ,  $\beta = 0.3$ ,  $\gamma = 0.2$ ,  $\delta = 0.1$ , and pruning threshold  $\tau = 0.5$ , following a coarse grid search on a held-out validation set of 5 tasks. Context budget is capped at  $B = 8,000$  tokens for the serialized active subgraph. Retry budgets for the verification loop are  $K_c = 3$  (compilation),  $K_t = 3$  (correctness), and  $K_p = 3$  (performance). For all LLM inference, temperature is set to  $T = 0.2$  with top- $p = 0.95$ .

## B MCDG Construction Quality and Structural Analysis

To validate MCDG’s effectiveness as a repository-level structured representation, we analyze the graph construction quality from three perspectives: overall scale (node and edge counts), connectivity density (edge-to-node ratio), and edge-type

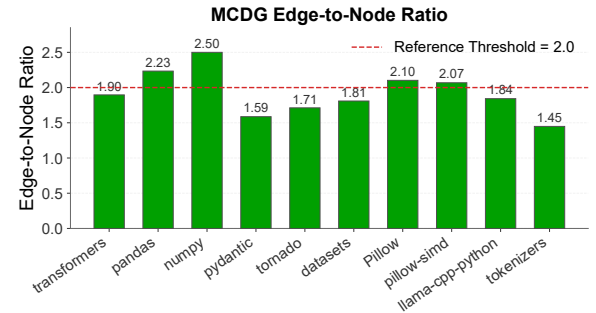


Figure 7: Edge-to-node ratio of MCDG across different repositories

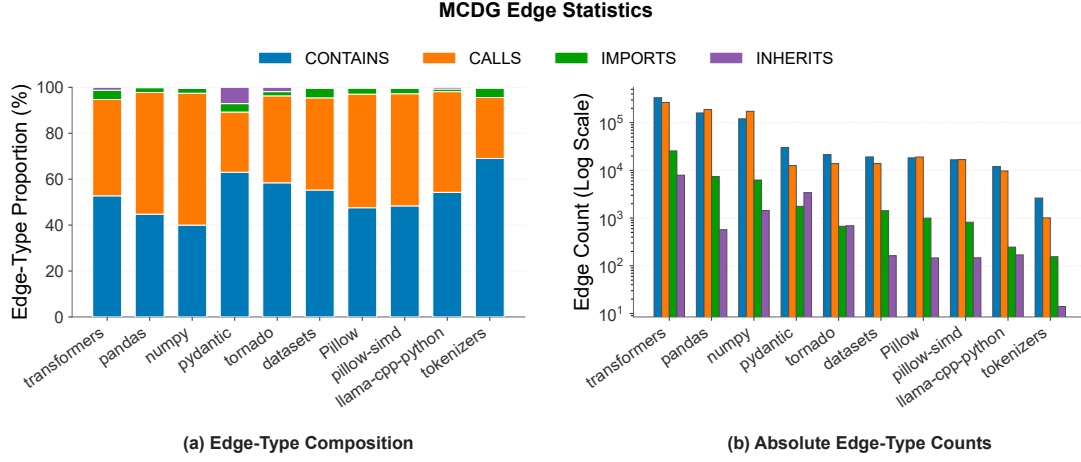


Figure 8: Statistical analysis of edge types in MCDG: (a) composition proportions of different edge types; (b) distribution of absolute counts of different edge types

Table 4: Per-repository breakdown of end-to-end optimization results (backbone: GPT-5.2)

| Repository   | #Tasks | Agentless OPT@1(%) | OpenHands OPT@1(%) | GLARE OPT@1(%) | GLARE Avg. Speedup |
|--------------|--------|--------------------|--------------------|----------------|--------------------|
| numpy        | 21     | 71.4               | 76.2               | 90.5           | 12.4%              |
| pandas       | 14     | 71.4               | 78.6               | 85.7           | 10.8%              |
| pillow-simd  | 7      | 71.4               | 85.7               | 100.0          | 12.4%              |
| tokenizers   | 3      | 66.7               | 66.7               | 100.0          | 12.5%              |
| transformers | 4      | 50.0               | 75.0               | 75.0           | 10.2%              |
| datasets     | 2      | 50.0               | 50.0               | 100.0          | 9.8%               |
| llama-cpp    | 2      | 100.0              | 100.0              | 100.0          | 14.2%              |
| Pillow       | 3      | 66.7               | 66.7               | 66.7           | 11.6%              |
| tornado      | 3      | 66.7               | 66.7               | 66.7           | 10.3%              |
| pydantic     | 4      | 50.0               | 50.0               | 75.0           | 10.1%              |
| Overall      | 63     | 68.3               | 74.6               | 87.3           | 11.6%              |

composition. These statistics characterize how MCDG captures multi-granular structural information across repositories of varying size, complexity, and coupling patterns.

**Node and Edge Distribution.** Figure 6 reports the number of nodes and edges in MCDG across the 10 GSO repositories. The scale varies substantially with repository size, module organization complexity, and cross-file dependency density. For small repositories (e.g., tokenizers, 2,626 nodes), MCDG yields a compact graph structure; for large, module-rich repositories with complex call relations (e.g., transformers, 332,756 nodes), MCDG naturally scales while preserving core structural relationships. This confirms that MCDG adaptively reflects the granularity of real-world codebases rather than being constrained to a fixed scale or topology.

**Edge-to-Node Ratio.** To assess whether multi-granular, multi-relation modeling leads to uncontrolled structural expansion, we compute the edge-to-node ratio for each repository. As shown in

Figure 7, the ratio ranges from 1.45 to 2.50 across repositories, with a mean of approximately 1.92. Critically, for larger repositories the number of edges increases substantially but the edge-to-node ratio does not rise linearly, indicating that MCDG’s multi-granular modeling strategy controls redundant relation proliferation as the graph scales. This stable connectivity provides a tractable foundation for downstream hotspot-seeded local expansion and heuristic pruning.

**Edge-Type Composition.** Beyond overall scale and density, MCDG’s key distinguishing feature is its use of multiple edge types to explicitly model different levels of code semantics. Figure 8 reports both the proportional composition and absolute counts of major edge types. The dominant edge types are CONTAINS, CALLS, and IMPORTS, which together form the backbone of repository-level code understanding—representing directory containment, function call relations, and cross-file dependencies, respectively. Lower-frequency but semantically important edge types such as INHER-

ITS provide necessary supplements for recovering object-level relationships and local program semantics. This “structural-backbone + semantic-supplement” organization ensures global connectivity while preserving information valuable for downstream performance bottleneck analysis, hotspot context recovery, and heterogeneous optimization decisions.

Overall, MCDG produces well-formed graphs of moderate scale, stable connectivity, and strong representational capacity across all evaluated repositories. The node and edge counts adapt to repository complexity without uncontrolled densification, and the edge-type distribution confirms that multi-grained, multi-semantic code graph representation is an effective modeling foundation for repository-level heterogeneous parallelization optimization.

## C Iterative Acceleration Opportunity Planning

Algorithm 2 specifies the planning state, action–observation loop, and stopping criteria used to assign each hotspot a CPU, GPU, or HYBRID label. The state retains the active subgraph, profiling signals, explored dependencies, candidate labels, evidence, and prior execution feedback. For each hotspot, GLARE iteratively selects an analysis action, incorporates its observation, and refines the candidate labels and evidence summary until the decision is confident, the budget is exhausted, or no GPU/Hybrid candidate remains viable. The algorithm returns the partition plan together with its evidence summaries.

## D Per-Repository Analysis

To further assess GLARE’s applicability and stability across diverse real-world codebases, we conduct a per-repository breakdown of results on the GSO subset. Figures 9, 10, and 11 compare methods along OPT@1, OPT@5, and average end-to-end speedup, respectively, for each repository. This finer granularity reveals how method performance varies with code organization, module coupling, and hotspot characteristics, providing a more complete picture of GLARE’s cross-repository generalizability.

**OPT@1.** GLARE achieves the best OPT@1 on most repositories, indicating that its advantage generalizes across project types (Figures 9, 10 and Table 4). On repositories with clear computational kernels—numpy, pandas, pillow-simd—GLARE

## Algorithm 2 Iterative Acceleration Opportunity Planning

**Require:** Active subgraph  $G_a$ , profiles  $P$ , hotspot candidates  $\mathcal{H}$ , planning budget  $K_a$

**Ensure:** Partition plan  $\mathcal{P}_{\text{plan}}$  and evidence summaries  $\mathcal{E}$

```

1:  $\mathcal{P}_{\text{plan}} \leftarrow \emptyset, \mathcal{E} \leftarrow \emptyset$ 
2:  $s_0 \leftarrow (G_a, P, \mathcal{H}, \mathcal{D}, \mathcal{Y}, \mathcal{E}, \mathcal{F})$ 
    $\mathcal{D}$ : explored dependency paths;  $\mathcal{Y}$ : candidate labels
    $\mathcal{F}$ : patch and execution-feedback history
3: for all  $h_i \in \mathcal{H}$  do
4:    $Y_i \leftarrow \{\text{CPU, GPU, HYBRID}\}, e_i \leftarrow \emptyset$ 
5:    $s_1 \leftarrow s_0$ 
6:   for  $t = 1$  to  $K_a$  do
7:      $a_t \leftarrow \text{SELECTACTION}(s_t)$ 
       Inspect calls, loops, memory access, profiles,
       transfer overhead, or interface/build dependencies
8:      $o_t \leftarrow \text{EXECUTE}(a_t)$ 
       Static analysis, profiling, or dependency evidence,
       and compiler/test/performance feedback
9:      $s_{t+1} \leftarrow \text{UPDATE}(s_t, a_t, o_t)$ 
10:     $(Y_i, e_i) \leftarrow \text{REFINE}(Y_i, e_i, s_{t+1})$ 
11:    if  $\text{CONFIDENT}(Y_i)$  or  $t = K_a$  then
12:      break
13:    else if  $Y_i \cap \{\text{GPU, HYBRID}\} = \emptyset$  then
14:      break
15:    end if
16:  end for
17:   $y_i \leftarrow \text{FINALLABEL}(Y_i, e_i)$ 
18:   $\mathcal{P}_{\text{plan}} \leftarrow \mathcal{P}_{\text{plan}} \cup \{(h_i, y_i)\}$ 
19:   $\mathcal{E} \leftarrow \mathcal{E} \cup \{(h_i, e_i)\}$ 
20: end for
21: return  $\mathcal{P}_{\text{plan}}, \mathcal{E}$ 

```

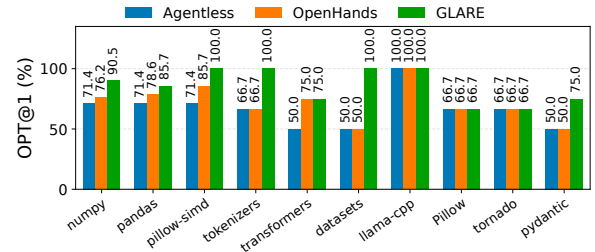


Figure 9: Comparison of OPT@1 across different code repositories.

attains 90.5%, 85.7%, and 100.0%, substantially exceeding Agentless (71.4%–71.4%) and OpenHands (76.2%–85.7%). On architecturally complex repositories like transformers and pydantic, GLARE scores 75.0% and 75.0%, still surpassing Agentless (50.0%) and matching or exceeding OpenHands (75.0%, 50.0%).

**OPT@5.** Under  $K = 5$  (Figure 10), GLARE remains best overall. On numpy, pandas, pillow-simd, GLARE’s OPT@5 reaches 95.2%, 92.9%, 100.0% vs. Agentless (81.0%–85.7%) and OpenHands (85.7%–100.0%). Gains are more modest on edge cases: tornado sees all methods at 66.7%, and pydantic has GLARE and OpenHands both at 75.0%. These results confirm that

Table 5: Comparison between GLARE and IntelliPerf.

| Dimension                | GLARE                                                                           | IntelliPerf                                                                       |
|--------------------------|---------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| Primary setting          | Repository-level CPU–GPU heterogeneous optimization                             | Performance engineering for existing GPU applications                             |
| Input assumption         | CPU-oriented repository; the hotspot need not have a GPU implementation         | Runnable application with existing GPU code under a ROCm workflow                 |
| Main problem             | Discover, partition, generate, and integrate GPU acceleration                   | Diagnose and optimize bottlenecks in existing GPU code                            |
| Profiling target         | CPU-side and end-to-end workload hotspots mapped to MCDG nodes                  | GPU kernels and GPU-side bottlenecks                                              |
| Localization and context | Profiling-guided MCDG active subgraph with cross-file dependencies              | Profiler and instrumentation localization to relevant source lines                |
| Optimization action      | CPU/GPU/HYBRID partitioning, CUDA/Triton generation, and host/build integration | Rewrite existing GPU code for bank conflicts, memory access, or atomic contention |
| Integration              | Repository-level patch compatible with original interfaces, build, and tests    | Operates on an existing buildable GPU application                                 |
| Validation               | Repository build, original tests, and end-to-end workload speedup               | GPU-output correctness and kernel/application performance                         |
| Evaluated ecosystem      | NVIDIA CUDA/Triton                                                              | AMD ROCm/HIP                                                                      |

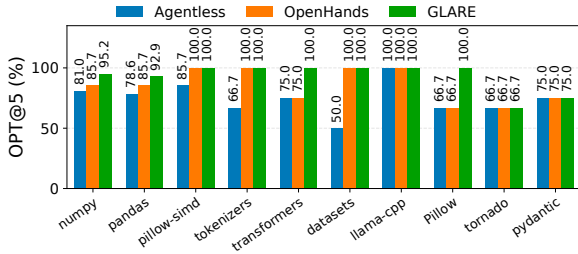


Figure 10: Comparison of OPT@5 across different code repositories.

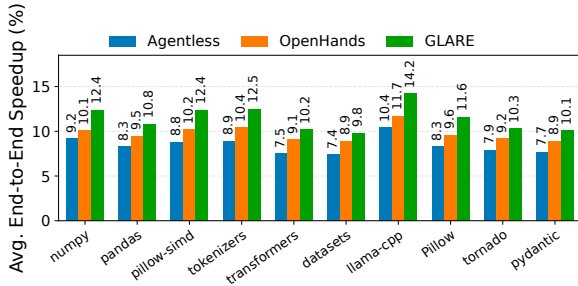


Figure 11: Comparison of average end-to-end speedup across different code repositories.

GLARE’s advantage reflects systematic candidate quality improvement, not single-generation luck.

**Speedup.** GLARE consistently outperforms both baselines in per-repository speedup (Figure 11). On compute-intensive repos with clear data-parallelism—llama-cpp, numpy, pillow-simd—GLARE achieves 14.2%, 12.4%, 12.4% speedup vs. Agentless (8.8%–10.4%) and OpenHands (10.1%–11.7%). On orchestration-heavy repos—datasets, transformers, pydantic—GLARE achieves 9.8%–10.2%,

still above Agentless (7.4%–7.7%) and OpenHands (8.9%–9.1%). These results confirm that GLARE exploits high-parallelism opportunities more fully while maintaining reliable optimization in complex projects.

## E Comparison with IntelliPerf

Table 5 compares GLARE with IntelliPerf (Awad et al., 2025). IntelliPerf profiles and tunes an existing ROCm GPU path, whereas GLARE starts from a CPU-oriented repository and performs hotspot discovery, MCDG context recovery, heterogeneous partitioning, GPU generation, and repository-level integration. Their optimization settings therefore differ substantially: IntelliPerf is complementary rather than a drop-in end-to-end baseline for our benchmark, and could tune a GPU path generated by GLARE.