2026 IEEE 50th Annual Computers, Software, and Applications Conference (COMPSAC) | 979-8-3315-4497-3/26/$31.00 ©2026 IEEE | DOI: 10.1109/COMPSAC69091.2026.00062

HGARena: Budgeted Test-Driven Multi-Agent Repository Issue Resolution with Heterogeneous Graph-Augmented Retrieval

Yuchen Cao, Jacky Wai Keung, Yifei Wang*, Yicheng Sun, and Zhenyu Mao
City University of Hong Kong, Hong Kong
Emails: {cynthcao2-c, ywang4748-c, yicsun2-c, zhenyumao2-c}@my.cityu.edu.hk, Jacky.Keung@cityu.edu.hk
*Corresponding author

***Abstract*—Real GitHub issue resolution requires a system to read issue reports, search large repositories, edit one or more files, and check the patch with executable tests. We present *HGARena*, a role-structured multi-agent framework that combines heterogeneous graph-augmented retrieval with budgeted iterative patch refinement. HGARena builds a typed RepoGraph over files, symbols, tests, and edit hunks. The framework retrieves graph-grounded context and runs a submitter–reviewer–executor loop where Docker test execution is the only final acceptance check, and reviewers only guide search. Using GPT-4o under a budget of 20 LLM calls, HGARena achieves a 25.3% resolve rate on SWE-bench Lite. Controlled variants show that iteration is essential, RepoGraph helps control token cost in iterative settings, and reviewer guidance improves the RR-token trade-off. Cross-backbone runs overall show similar resolve rates but different token costs across three commercial backbones.**

***Index Terms*—Retrieval-Augmented Generation, Graph retrieval, Program repair, Repository-level issues**

I. Introduction

Repository-level issue resolution is fundamentally different from function-level code generation. Given a repository snapshot and a natural-language issue description, a system must locate the relevant code, generate a patch, and rely on executable tests to determine whether the patch is correct [1]. This setting is closer to real software maintenance than self-contained coding benchmarks. It requires fault localization across many files, coordinated edits under repository dependencies, and reasoning from sparse binary feedback produced by test execution.

A central obstacle is that the evidence needed to fix an issue is inherently *heterogeneous*. Useful signals may come from issue text, source files, functions or classes, call sites, failing tests, stack traces, and previous edit attempts. If a repository is treated as a flat text corpus, these signals are mixed together and their relations are weakened. For example, a failing test may point to a covered symbol, a file may define several relevant functions, and a symbol may call another symbol that is closer to the actual bug. We therefore model the repository as a lightweight heterogeneous graph with typed nodes and typed edges. The graph serves as a retrieval scaffold: it helps the system follow code and test-related relations without acting as a hard controller over the entire search space.

Despite rapid progress in LLM-based code generation, many widely used benchmarks still focus on settings that are much simpler than repository maintenance. HumanEval mainly evaluates single-function completion [2], and contest-style benchmarks emphasize isolated problem solving [3]. These settings are useful for measuring local code generation ability, but they do not require repository navigation, multi-file editing, or tight interaction with an executable oracle. Classical test-based automated program repair (APR) studies test-driven patch search and learned repair heuristics [4]–[6], while neural repair methods learn bug-fix transformations from mined code changes [7], [8]. However, applying these ideas to real GitHub issues remains difficult because the system must retrieve evidence, use tools, and localize the right edit target under a limited budget.

Recent software engineering agents move closer to this setting by allowing LLMs to operate inside a development environment. SWE-agent formalizes an agent-computer interface and lets the model iteratively search, edit, and test [9]. Agentless studies which agent components are essential and simplifies the overall stack [10], while AutoCodeRover explores structured localization and iterative editing [11]. SwingArena further introduces competitive dynamics for long-context issue solving [12]. Even with this progress, two bottlenecks remain: the system must find useful evidence from thousands of files, and it must decide which edit to try next under a strict interaction budget. From this view, repository-level issue resolution is a constrained search problem over a large and heterogeneous edit space, where weak context selection or poorly guided revisions can quickly waste the available attempts [1], [10], [13].

Retrieval-augmented generation (RAG) is a natural way to ground generation on external evidence [14]. Prior work improves retrieval through query expansion and synthetic query generation [15], and surveys summarize design choices such as chunking, hybrid retrieval, reranking, and evaluation [16]–[18]. Graph-based retrieval further shows that structural links can help expand from local evidence to broader context [19], and LightRAG uses lightweight graph indexing with dual-level retrieval for efficiency [20]. However, repository-level issue resolution requires a more code-specific form of retrieval: files, symbols, tests, edit hunks, and failure traces play different roles, and their relations directly affect where a patch should be made. This motivates a typed repository graph that treats these

2836-3795/26/$31.00 ©2026 IEEE
DOI 10.1109/COMPSAC69091.2026.00062

artifacts as first-class nodes and uses relation-aware signals to control context selection and cost.

We propose *HGARena*, a budgeted framework for repository-level issue resolution with Heterogeneous Graph-Augmented Retrieval (HGAR) and a submitter–reviewer–executor loop. HGARena builds a lightweight heterogeneous repository graph to organize files, symbols, tests, and patch hunks. HGAR retrieves compact evidence through lexical-first candidate recall, lightweight graph reranking, and conservative typed neighborhood expansion. The framework separates three functions: a *submitter* proposes a patch, a *reviewer ensemble* provides three complementary perspectives in a single interaction, and an *executor* applies the patch and runs tests. Each candidate patch is instantiated as a shared patch-hunk node, so reviewer feedback, executor output, and later retrieval steps refer to the same edited region.

HGARena does not use reviewers as correctness judges. LLM-only judging can suffer from systematic biases, including position and formatting effects [21]–[23]. In our framework, the reviewer ensemble only steers the search by identifying likely failure causes, risky edits, and promising follow-up locations. Final success is determined by test execution. This design keeps the benefits of structured reviewer guidance while avoiding an LLM-only acceptance criterion. Our goal is not to claim a new leaderboard result, but to test whether typed retrieval and reviewer-guided search improve controlled baselines under a fixed call budget.

Our primary contributions are:

- **Budgeted execution-grounded repair.** We introduce a submitter–reviewer–executor framework where reviewers guide search and executable tests are the sole acceptance oracle.
- **Lightweight heterogeneous RepoGraph.** We build a typed graph that connects files, symbols, tests, and patch hunks through relations such as DEFINES, IMPORTS, CALLS, COVERS, and TOUCHES.
- **Graph-guided retrieval under budget.** HGAR combines lexical-first recall, graph reranking, and conservative typed expansion to prioritize promising files, symbols, and follow-up edit locations.
- **Controlled evaluation.** We evaluate HGARena on SWE-bench Lite under a fixed LLM-call budget, with ablations that isolate graph retrieval and reviewer feedback.

II. RELATED WORK

A. Repository-Level Benchmarks for Issue Resolution

SWE-bench introduces a repository-level benchmark where patches are validated by executing tests in containerized environments [1]. SWE-bench Lite reduces cost while preserving the same execution-based oracle [24], and SWE-bench Verified provides a human-validated subset to reduce evaluation ambiguity [25]. Beyond Python-centric benchmarks, Multi-SWE-bench extends issue-resolution evaluation to multiple languages [26]. These benchmarks complement earlier code-generation evaluations such as HumanEval [2] and contest-style synthesis tasks [3], but emphasize repository navigation, multi-file edits, and executable correctness.

TABLE I: Comparison of representative approaches related to repository-scale issue resolution.

Method	Repo-level	Exec-oracle	Hetero-Graph	Arena
HumanEval / CodeGen	×	×	×	×
GenProg / APR	✓	✓	×	×
SWE-agent	✓	✓	×	×
Agentless	✓	✓	×	×
AutoCodeRover	✓	✓	×	×
SwingArena	✓	✓	×	✓
GraphRAG / LightRAG	×	×	✓	×
HGARena (Ours)	✓	✓	✓	✓

Table I summarizes key design differences between HGARena and representative prior lines of work.

B. LLM-Based Software Engineering Agents

Software engineering agents enable an LLM to operate a development environment through tools such as search, file access, and test execution. Recent code-generation work also explores specialized multimodal settings such as chart-to-code translation [27]; HGARena instead targets repository-level issue repair where patches must be validated by executable tests. SWE-agent establishes an agent-computer interface, and ReAct-style agents interleave reasoning and tool actions for iterative search, edit, and test workflows [9], [28]. Agentless studies which agent components matter and reports strong results with a simplified stack [10], while AutoCodeRover explores structured localization and iterative repair [11]. SwingArena introduces competitive dynamics for long-context issue solving [12]. HGARena is complementary: rather than modeling adversarial patch-vs-test competition, it uses a reviewer ensemble as auxiliary search guidance, with lightweight graph-guided retrieval and execution as the sole oracle.

C. Automated Program Repair and Patch Generation

Test-based APR has a long history, including GenProg, Prophet, and Genesis-style code transformations [4], [5], [29]. Surveys summarize recurring APR challenges such as patch overfitting and benchmark bias [6]. Data-driven repair learns bug-fix patterns from mined changes, including DeepFix for compiler errors [7], neural machine translation for bug-fixing patches [8], and retrieval-augmented patch generation [30]. HGARena inherits the execution-grounded spirit of APR but targets repository-scale issues where localization and context selection are first-order concerns.

D. Retrieval-Augmented Generation for Code and Cost-Aware Retrieval

RAG combines parametric generation with non-parametric retrieval to ground outputs on external evidence [14]. HyDE shows that synthetic queries can improve dense retrieval without relevance labels [15]. Surveys summarize design choices and trade-offs in chunking, hybrid retrieval, reranking, latency, and cost [16]–[18]. RAG evaluation frameworks such as

RAGAS motivate explicit measurement of retrieval quality and answer faithfulness [31]. HGARena inherits these insights but focuses on *repository* evidence selection, where the context must be compact, actionable, and ultimately judged by test execution rather than textual faithfulness.

E. Graph-Augmented Retrieval and Heterogeneous Graph Modeling

GraphRAG argues for moving from local retrieved evidence to broader context by leveraging graph structure [19]. LightRAG advocates a lightweight graph index and dual-level retrieval for efficient end-to-end performance [20]. Graph learning studies scalable graph transformer designs [32], while graph-oriented LLM research translates graph structure into LLM-usable sequences [33], [34]. In contrast, repository-scale issue resolution is naturally *heterogeneous*: files, symbols, tests, and patch hunks are distinct entities, and relations such as definition, import, call, test coverage, and edit history carry different semantics. This motivates a *typed* repository graph, a setting related to relational and heterogeneous graph learning [35], [36]. HGARena adopts this view pragmatically: the graph is used mainly as a retrieval scaffold, not as heavyweight whole-program analysis.

F. LLM-as-a-Judge, Reviewer Guidance, and Agreement Measures

LLM-as-a-judge can reduce labeling cost but suffers from systematic biases, including position bias [21] and broader inconsistencies relative to human judgments [22]. A recent survey summarizes reliability risks and mitigation strategies [23]. HGARena therefore uses execution as the acceptance oracle and treats the reviewer ensemble as search guidance. Agreement measures such as Krippendorff's alpha can characterize reviewer stability when multiple perspectives are used [37], but such signals remain diagnostic rather than correctness-defining.

III. HGARena Framework

A. Problem Setting

We study repository-scale issue resolution under the SWE-bench formulation [1]. Each task provides (i) a natural-language issue description, (ii) a repository snapshot at a fixed commit, and (iii) an executable evaluation script. A system must produce a patch (possibly multi-file) such that the test suite passes under the provided evaluation protocol.

B. Overview and Design Choices

HGARena treats issue resolution as a *budgeted, execution-grounded search* over edits. In this view, RepoGraph and HGAR define the evidence expansion policy that determines the next search step before each submitter attempt.

Figure 1 illustrates three core components:

(1) RepoGraph (heterogeneous index). We build a lightweight *heterogeneous* repository graph whose typed nodes represent files, symbols, tests, and patch hunks, and whose typed edges represent semantic relations (definition, import, call/*reference dependency*), execution feedback (test evidence), and edit history (touched regions).

(2) HGAR (Heterogeneous Graph-Augmented Retrieval). Given the issue and the latest execution feedback, we retrieve a compact, structured context under a hard token budget using lexical-first candidate recall, lightweight graph reranking, and conservative typed neighborhood expansion.

(3) Arena loop (budgeted search with execution oracle). A submitter proposes patches; a reviewer ensemble produces three complementary perspectives in a single call and returns structured critiques, file-scope constraints, and ranked edit suggestions; an executor applies candidate patches and runs the official SWE-bench test suite in Docker as the sole correctness oracle. We avoid LLM-only acceptance to reduce judge-induced instability [23].

C. RepoGraph: A Lightweight Heterogeneous Repository Index

a) Heterogeneous schema.: Given a repository snapshot, HGARena constructs a typed directed multigraph $G = (V, E)$. Let $\mathcal{T}_V$ denote node types and $\mathcal{T}_E$ denote edge types. We use:

$$V = V^F \cup V^S \cup V^T \cup V^H, \qquad \mathcal{T}_V = \{F, S, T, H\}, \quad (1)$$

where V^F are **file nodes**, V^S are **symbol nodes** (functions/classes/methods), V^T are **test nodes** (test files and, when available, test cases), and V^H are **patch-hunk nodes** instantiated during the arena loop to represent candidate edited regions.

Edges are typed as:

$$\begin{aligned} E &= E^{\textsc{Defines}} \cup E^{\textsc{Imports}} \cup E^{\textsc{Calls}} \\ &\quad \cup E^{\textsc{Covers}} \cup E^{\textsc{Touches}}, \\ \mathcal{T}_E &= \{\textsc{Defines}, \textsc{Imports}, \textsc{Calls}, \textsc{Covers}, \textsc{Touches}\}. \end{aligned} \quad (2)$$

Edge directions (source/target types). To make the schema unambiguous in the paper, each directed edge type has a fixed source/target node type:

$$\begin{aligned} E^{\textsc{Defines}} &\subseteq V^F \times V^S, \qquad E^{\textsc{Imports}} \subseteq V^F \times V^F, \\ E^{\textsc{Calls}} &\subseteq V^S \times V^S, \qquad E^{\textsc{Covers}} \subseteq V^T \times (V^F \cup V^S), \\ E^{\textsc{Touches}} &\subseteq V^H \times (V^F \cup V^S). \end{aligned}$$

Concretely, $E^{\textsc{Defines}}$ points from a *defining file* to the *symbol defined in it*; $E^{\textsc{Imports}}$ points from an *importing file* to an *imported file/module*; $E^{\textsc{Calls}}$ points from a *caller symbol* to a *callee symbol* (best-effort static linking); $E^{\textsc{Covers}}$ points from a *failing test* to the *file/symbol implicated by execution evidence* (e.g., stack trace, import trace); and $E^{\textsc{Touches}}$ points from a *patch hunk* to the *file/symbol modified by the patch*. Table II summarizes the node/edge types (including source/target types) used throughout Section III-D.

b) What each node/edge means in practice.: The goal of RepoGraph is not to be a fully sound program analysis graph, but a *cheap, typed scaffold* that connects: (i) *where the code lives* (files), (ii) *what can be edited* (symbols), (iii) *what fails* (tests/log evidence), and (iv) *what has been tried* (patch

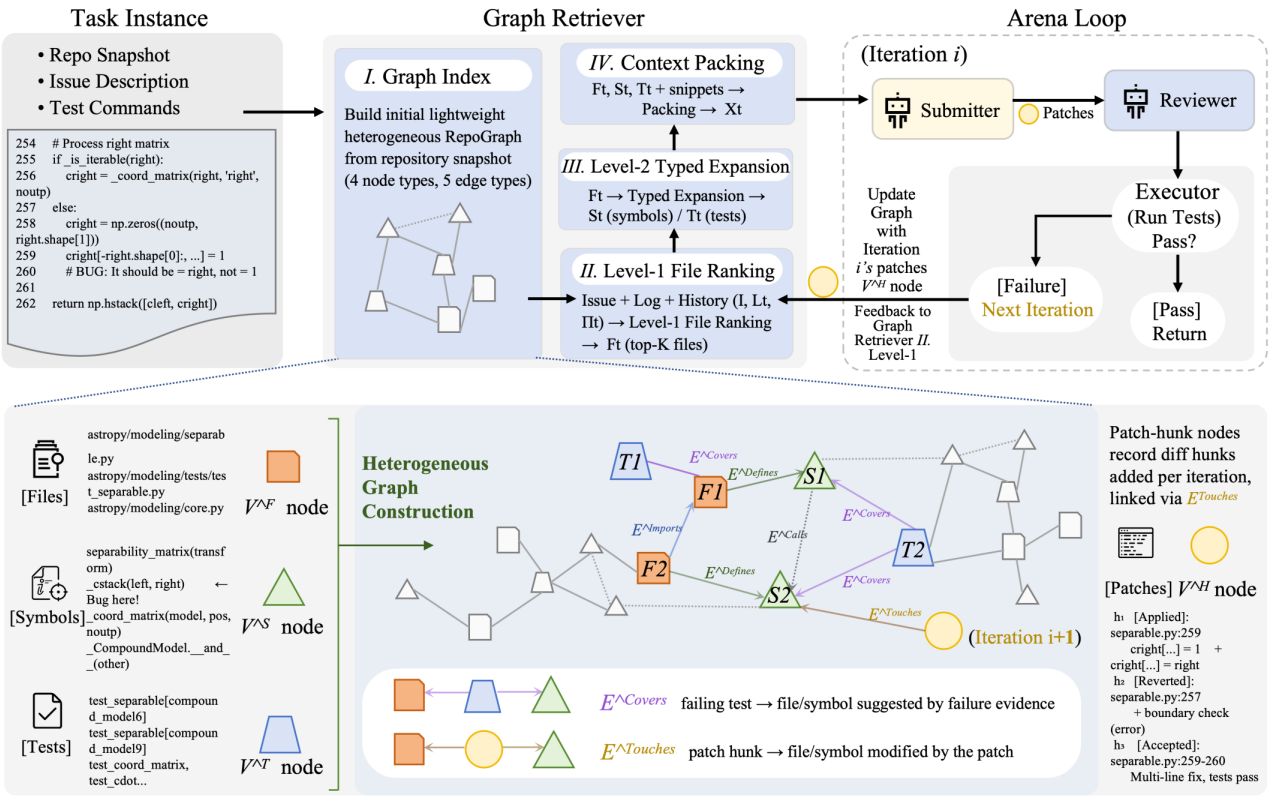

Fig. 1: HGARena overview. The system builds a lightweight *heterogeneous* repository graph (RepoGraph), retrieves type-aware graph-grounded contexts via HGAR, and runs a budgeted submitter–reviewer–executor loop. A candidate patch is instantiated as a shared patch-hunk node, which is then used by both the reviewer ensemble and the executor. The executor runs tests as the sole correctness oracle; the reviewer ensemble provides three complementary perspectives in one interaction: critique, file-scope guidance, and follow-up edit suggestions.

hunks/touches). A file node stores its path and lightweight text/embedding features; a symbol node stores its signature and span (start/end line ranges) so we can extract bodies for context packing; a test node stores failing test identifiers (when available) and minimal metadata for prioritization; and a patch-hunk node stores diff hunks produced by submitter attempts so that the graph explicitly records the search trajectory.

c) Why heterogeneity matters.: The heterogeneous schema makes execution feedback and edit history first-class signals. For example, a meta-path $\texttt{Test} \xrightarrow{\text{COVERS}} \texttt{Symbol} \xrightarrow{\text{CALLS}} \texttt{Symbol}$ captures failure propagation, while $\texttt{Hunk} \xrightarrow{\text{TOUCHES}} \texttt{Symbol}$ captures where the system has already searched. These typed relations enable type-aware expansion that is difficult to reproduce with flat chunk retrieval.

d) Construction is deliberately lightweight (realistic).: RepoGraph is designed to be implementable with inexpensive heuristics rather than heavyweight whole-program analysis: (i) DEFINES/IMPORTS edges can be extracted from language parsers (e.g., AST or tree-sitter); (ii) CALLS edges are approximated by best-effort dependency linking over parsed symbols, combining call-site hints and name-based reference matching (not perfectly sound); (iii) COVERS edges are derived from executor outputs when available (e.g., failing test identifiers, stack traces, import traces), and gracefully degrade to file-level matching if fine-grained signals are missing. This choice emphasizes stable behavior and low overhead under strict budgets.

D. Heterogeneous Graph-Augmented Retrieval (HGAR)

a) Formulation.: At iteration t, the arena maintains an issue context $q_t = (I, L_t, \Pi_t)$, where I is the issue text, L_t is the latest executor log, and Π_t is patch history (touched files/symbols). Retrieval is factorized into an indexing function $\phi(\cdot)$ and a retriever $\psi(\cdot)$:

$$G = \phi(\mathcal{R}), \qquad X_t = \psi(q_t; G), \tag{3}$$

where $\mathcal{R}$ is the repository snapshot and X_t is the packed context under a hard token budget B_{ctx}.

b) Notation (what the symbols mean).: q_t is the *current retrieval query state* summarizing what the system knows at iteration t: I provides the natural-language intent (what to

TABLE II: RepoGraph schema (node types and directed edge types).

Nodes			
Type		**Definition**	
V^F	File nodes (path, language, lightweight embedding/lexical features).		
V^S	Symbol nodes (function/class/method signatures and spans).		
V^T	Test nodes (test files/cases, framework, failure history if available).		
V^H	Patch-hunk nodes (diff hunks linking edits to file/symbol regions).		
Edges (directed)			
Edge type	**Src**	**Dst**	**Semantics**
E^{DEFINES}	V^F	V^S	defining file → symbol defined in the file.
E^{IMPORTS}	V^F	V^F	importing file → imported file/module dependency.
E^{CALLS}	V^S	V^S	caller symbol → callee symbol (best-effort call/*reference* linking).
E^{COVERS}	V^T	$V^F \cup V^S$	failing test → file/symbol implicated by failure evidence (logs/trace).
E^{TOUCHES}	V^H	$V^F \cup V^S$	patch hunk → file/symbol modified by the patch.

fix), L_t provides *execution evidence* (what failed and where it failed), and Π_t provides *search history* (what we already tried to edit or inspect). The output X_t is a *single bundled context* shown consistently to the submitter and reviewers (file headers/snippets, symbol bodies, and concise logs), constructed to fit within a strict budget B_{ctx}. Throughout, $\text{File}(s)$ maps a symbol node $s \in V^S$ to its defining file node in V^F.

c) Dual-level retrieval with type-aware expansion.: HGAR retrieves evidence in two stages: **(Level-1) candidate file recall and reranking** followed by **(Level-2) conservative typed neighborhood expansion**. We follow the RepoGraph schema in Table II to define candidate sets and expansion operators.

Level-1: file ranking. We score each candidate file node $f \in V^F$ by a hybrid function:

$$s_F(f) = \alpha\,\text{Sim}(I, f) + \beta\,\text{Sim}(L_t, f) + \gamma\,\text{Touched}(f; \Pi_t) + \delta\,\text{Centrality}(f). \tag{4}$$

We then select top-K_F files $\mathcal{F}_t$. In our implementation, lexical relevance dominates the initial candidate pool, while TOUCHED and CENTRALITY act as lightweight reranking factors rather than hard constraints. Here Sim can be lexical (e.g., keyword/BM25 overlap) or embedding similarity; $\text{Touched}(f; \Pi_t)$ increases the score of files repeatedly implicated by past attempts; and $\text{Centrality}(f)$ (computed on the import graph induced by E^{IMPORTS}) serves only as a weak preference for hub files.

Level-2: typed neighborhood expansion. Starting from $\mathcal{F}_t$, we expand a bounded neighborhood to collect symbol candidates using typed edges, while test nodes are instantiated directly from the latest executor log:

$$\begin{aligned} \mathcal{S}_t &= \text{Expand}\Big(\mathcal{F}_t;\, E^{\text{DEFINES}}\Big) \\ &\quad \cup\ \text{Expand}\Big(\text{Expand}\big(\mathcal{F}_t;\, E^{\text{DEFINES}}\big);\, E^{\text{CALLS}}\Big), \\ \mathcal{T}_t &= \text{FailTests}(L_t). \end{aligned} \tag{5}$$

Here $\text{FailTests}(L_t)$ extracts failing test identifiers from L_t and maps them to nodes in V^T (returning $\varnothing$ if the log does not expose test IDs). The typed edge E^{COVERS} is directed from these failing test nodes to implicated files/symbols and is used for proximity signals and context packing.

How to read the expansion equation. The first term $\text{Expand}(\mathcal{F}_t; E^{\text{DEFINES}})$ means: *take the top-ranked files and collect all symbols defined inside them*. The second term expands one more hop via E^{CALLS} starting from those symbols, meaning: *include symbols that are dependency-adjacent (called/referenced) to the initially retrieved symbols*. In the current implementation, this second-hop expansion is applied conservatively, prioritizing exact-match or failure-proximate symbol seeds to reduce graph-induced noise.

What the operator $\text{Expand}(\cdot)$ **does.** By default, expansion is limited to 1-hop to keep cost predictable. Formally, for a node set $\mathcal{A}$ and an edge type E^τ, $\text{Expand}(\mathcal{A}; E^\tau)$ returns the set of nodes reachable by following one outgoing edge of that type from any node in $\mathcal{A}$. (Optionally, 2-hop expansion can be enabled with decay, but we keep 1-hop as the default setting.)

Symbols are ranked by combining file relevance and proximity to failure evidence:

$$\begin{aligned} s_S(s) = s_F\big(\text{File}(s)\big) &+ \eta\,\text{Prox}(s, \mathcal{T}_t) \\ &+ \kappa\,\text{Match}(s, I, L_t) + \rho\,\text{Touched}(s; \Pi_t). \end{aligned} \tag{6}$$

What each term in $s_S(\cdot)$ **captures.** The first term $s_F(\text{File}(s))$ inherits file-level relevance. $\text{Prox}(s, \mathcal{T}_t)$ measures how close a symbol is to failing-test evidence (e.g., short typed path distance to a test node connected by COVERS); $\text{Match}(s, I, L_t)$ measures whether the symbol name/signature/body aligns with issue keywords or log identifiers (e.g., exception type, function name in stack trace); and $\text{Touched}(s; \Pi_t)$ biases toward symbols repeatedly touched or repeatedly implicated by failure evidence.

d) One-iteration walkthrough (step-by-step).: For clarity, one retrieval iteration proceeds as: (1) build a lexical-first file candidate pool and rerank it with lightweight graph signals to obtain $\mathcal{F}_t$ (top-K_F files); (2) expand $\mathcal{F}_t$ via DEFINES and conservatively via CALLS to get symbol candidates $\mathcal{S}_t$; (3) parse failing tests from L_t to obtain $\mathcal{T}_t$ and use COVERS relations as execution-evidence signals; (4) rank symbols by $s_S(s)$ and keep the highest-scoring signatures/bodies; and (5) pack everything into X_t under the budget B_{ctx}. This makes retrieval explicitly *type-aware*: files, symbols, and tests play different roles and are not mixed into a single flat chunk pool.

Context packing under a hard budget. We assemble the context bundle X_t under a token budget B_{ctx} by prioritizing: (i) top-ranked file headers and relevant regions, (ii) top-ranked symbols (signatures + bodies), (iii) failing tests and concise log snippets, and (iv) minimal dependency snippets needed for compilation. This packing strategy ensures the submitter and reviewers see consistent, evidence-grounded context rather than arbitrary long chunks.

e) Incremental updates.: Immediately after the submitter produces a candidate patch, HGARena instantiates a patch-hunk node $h_t \in V^H$ and adds provisional TOUCHES edges

linking the patch to modified files/symbols. After execution, we update only the subgraph induced by changed files Δ_t and their 1-hop neighbors in IMPORTS/CALLS, and refresh COVERS edges using the latest executor feedback. This prevents stale context across iterations without rebuilding the entire index.

E. Arena Loop: Submitter–Reviewer–Executor

At each iteration, HGARena maintains a state $S_t = \{X_t, h_t, p_t, r_t, out_t\}$, consisting of retrieved context X_t, the current patch-hunk node h_t, candidate patch p_t, reviewer aggregate r_t, and executor output out_t.

a) Submitter.: The submitter proposes a patch using X_t and the accumulated history (previous diffs, failures, and reviewer hints). In our implementation, the submitter is prompted to output a unified diff and to keep edits minimal but test-effective. Once produced, the candidate patch is instantiated as a shared patch-hunk node for the current iteration.

b) Reviewers (search guidance, not an oracle).: We use three reviewer perspectives: (i) failure analysis, which hypothesizes root causes from execution evidence; (ii) patch validity, which identifies risky edits and missing constraints; and (iii) test alignment, which highlights relevant validation targets. In the main HGARena configuration, these three perspectives are produced in a *single reviewer ensemble call* over the same candidate patch node. The reviewer aggregate outputs: (a) structured critique, (b) ranked edit suggestions, (c) a confidence indicator, and (d) optional file-scope guidance such as preferred edit files or single-file continuation. Crucially, reviewers do not decide acceptance.

c) Executor (the oracle).: The executor applies the same candidate patch and runs the evaluation script. Test execution solely determines success.

F. Candidate Ranking and Stopping

HGARena returns immediately when a test-passing patch is found. Otherwise, under a fixed interaction budget, we keep the best-effort candidate according to a lexicographic score:

$$\textsc{Key}(p) = \left\langle \begin{array}{c} \mathbb{I}[\textsc{Pass}(p)], \\ -\,\textsc{FailCount}(p), \\ \textsc{RevScore}(p), \\ -\,|\textsc{Diff}(p)| \end{array} \right\rangle. \tag{7}$$

where FAILCOUNT is the number of failing tests (or a large value if tests crash), REVSCORE is the ensemble-derived reviewer score used only when tests do not pass, and $|\textsc{Diff}|$ is a tie-breaker preferring smaller edits. We select the candidate with the maximum lexicographic key, ensuring that any passing patch dominates all non-passing ones, while still making progress measurable under failure.

G. Algorithm

Algorithm 1 summarizes HGARena with a fixed interaction budget B. We count each submitter call and each reviewer-ensemble call as one interaction (LLM API call); Docker execution runs after each attempt and is not counted toward B.

Algorithm 1 HGARena (Budgeted Arena Loop)

Require: Issue text I, repo snapshot $\mathcal{R}$, budget B, context budget B_{ctx}
Ensure: Patch $p^\star$ (or FAIL)

$G \leftarrow \phi(\mathcal{R})$ ▷ Build heterogeneous RepoGraph
$p^\star \leftarrow \varnothing$; $bestKey \leftarrow \langle 0, -\infty, -\infty, -\infty \rangle$
$L \leftarrow \varnothing$; $\Pi \leftarrow \varnothing$; $b \leftarrow 0$
while $b < B$ **do**
  $X \leftarrow \psi((I, L, \Pi); G)$ ▷ Lexical-first recall + graph reranking + typed packing
  $p \leftarrow \textsc{Submitter}(I, X, \Pi, L)$; $b \leftarrow b + 1$
  $h \leftarrow \textsc{InstantiatePatchNode}(p)$
  $\textsc{AddTouches}(G, h, p)$
  $r \leftarrow \textsc{ReviewerEnsemble}(I, X, h, p, L)$; $b \leftarrow b + 1$
  $out \leftarrow \textsc{Execute}(\mathcal{R}, p)$
  $key \leftarrow \textsc{Key}(p)$
  if $key > bestKey$ **then** $bestKey \leftarrow key$; $p^\star \leftarrow p$
  end if
  if out.pass $=$ **true then**
    return p
  end if
  $L \leftarrow out$.log; $\Pi \leftarrow \Pi \cup \textsc{Touched}(h)$
  $\textsc{UpdateGraph}(G, h, out)$ ▷ Refresh covers and patch outcome
end while
return $p^\star$

IV. EXPERIMENTS

We evaluate HGARena on the complete SWE-bench Lite benchmark, the standard 300-task setting used by prior repository-level issue-resolution systems. Our evaluation addresses three questions: (i) whether HGARena improves resolution quality, (ii) what makes HGARena effective compared with controlled internal variants, and (iii) how HGARena behaves across different LLM backbones.

A. Experimental Setup

Dataset. We evaluate on the complete **SWE-bench Lite benchmark** (300 tasks) [1]. SWE-bench Lite is a curated subset of real GitHub issues from Python repositories such as Django, Matplotlib, Scikit-learn, and SymPy. Each task provides an issue description, a fixed repository snapshot, executable tests, and a reference patch. We use Lite only because it is the common reporting basis for prior systems.

Evaluation Protocol. We use a fixed interaction budget of $B = 20$ **LLM API calls per task**. For HGARena, each iteration counts one submitter call and one reviewer-ensemble call; Docker execution is *not* counted toward B. A task is counted as resolved *if and only if* the generated patch passes the official SWE-bench Docker-based evaluation harness. All correctness results are computed from Docker execution, not from LLM self-judgment or reviewer preference.

Metrics. We report three metrics. **Resolve Rate (RR)** is the percentage of tasks resolved by the generated patch. **Budget**

TABLE III: Comparison with prior work on SWE-bench Lite. Prior systems are shown with their reported configurations. HGARena reports our final result under the Lite-only evaluation setting.

System	RR (%)	Tasks	Validation
SWE-agent [9]	18.0	300	Docker
AutoCodeRover [11]	19.0	300	Docker
Agentless [10]	27.3	300	Docker
HGARena (Ours, GPT-4o)	**25.3**	**300**	**Docker**

TABLE IV: Controlled variant analysis on the complete SWE-bench Lite benchmark (300 tasks, GPT-4o, official Docker validation). BE and token usage are computed from logged API usage.

Variant	RR (%)	BE (calls)	Tokens
SingleAgent + no-RepoGraph	5.3	1.0 / 20	4,474
SingleAgent + RepoGraph	0.0	1.0 / 20	3,056
IterAgent + no-RepoGraph	24.7	8.2 / 20	410,456
IterAgent + RepoGraph	24.0	9.4 / 20	211,299
HGARena	**25.3**	**13.0 / 20**	**204,995**

Efficiency (BE) is the average number of LLM API calls consumed per task under the 20-call budget. **Token Consumption** is measured from logged API usage and reflects the total input-output token footprint of each setting.

Controlled Variants. We compare three framework families. **SingleAgent** is a one-shot baseline that performs one retrieval-and-patch generation step followed by one Docker validation, without iterative refinement or reviewer guidance. **IterAgent** is an execution-guided iterative baseline that repeatedly regenerates patches based on Docker feedback under the same budget, but does not use reviewer guidance. **HGARena** extends iterative repair with HGAR retrieval and a reviewer ensemble. For SingleAgent and IterAgent, the "no-RepoGraph" variants use lexical retrieval only, while the "RepoGraph" variants use RepoGraph-based context organization.

Reviewer Ensemble and Arena Meaning. HGARena uses a reviewer ensemble as structured search guidance rather than as a correctness judge. Given the retrieved context, candidate patch, and latest execution feedback, the ensemble performs three verifier-style checks: failure analysis, patch validity, and test alignment. It returns structured critiques, ranked follow-up edit suggestions, and optional file-scope guidance. In this paper, "arena" refers to the role-structured submitter–reviewer–executor loop, not a model-versus-model competition.

Backbones. The main controlled comparison uses **GPT-4o**, which provides the cleanest setting for comparing internal variants under the same backbone. To examine whether the framework behavior is tied to a single model, we also evaluate HGARena with **Claude Sonnet 4.5** and **Gemini 2.5 Pro**. In these cross-backbone runs, the same model instantiates both the submitter and the reviewer ensemble under the same arena protocol.

Implementation Details. RepoGraph construction uses Tree-sitter parsers for Python, extracting file, symbol, test, and patch-hunk nodes, together with typed edges such as DEFINES, IMPORTS, CALLS, COVERS, and TOUCHES. HGAR uses lexical-first candidate recall, lightweight RepoGraph-based reranking, and conservative typed expansion to pack a compact context shared by the submitter and reviewer ensemble. The arena loop terminates when either Docker tests pass or the 20-call budget is exhausted.

B. RQ1: Does HGARena Improve Resolution Quality on SWE-bench Lite?

We first compare HGARena with representative systems reported on the standard SWE-bench Lite setting. The goal of this comparison is not to create a backbone-matched controlled study, since prior systems use their own reported configurations. Instead, it places HGARena in the same 300-task Docker-validated benchmark context used by prior work.

As shown in Table III, HGARena achieves **25.3%** RR on the complete SWE-bench Lite benchmark. It improves over SWE-agent by 7.3 percentage points and AutoCodeRover by 6.3 percentage points, while remaining 2.0 percentage points below Agentless. This result places HGARena in a competitive range among Docker-validated SWE-bench Lite systems, while keeping a distinct design focus on lightweight heterogeneous retrieval, reviewer-guided search, and execution-only acceptance under a fixed call budget.

Benchmark-level interpretation. The comparison shows that HGARena reaches a competitive solve rate on the same Lite benchmark used by prior issue-resolution agents. It does not claim a new state-of-the-art result; rather, it shows that typed repository retrieval and a role-structured arena loop can reach this performance band while preserving Docker execution as the final oracle.

The 25.3% result should therefore be interpreted as a benchmark-level effectiveness result under a fixed budget. Compared with systems that emphasize tool operation or agent simplification, HGARena emphasizes how evidence is selected and how failed attempts are guided. This distinction is important for repository-level repair, where a system must decide not only what patch to write, but also what code evidence to inspect, which failed attempt to trust, and where to continue the search.

C. RQ2: What Makes HGARena Effective?

We next study HGARena through controlled internal variants under the same SWE-bench Lite, GPT-4o, and Docker-validation setting. Unlike RQ1, which compares with external systems, this section isolates the roles of iteration, RepoGraph-based context organization, and reviewer-guided search. The purpose is to understand why HGARena obtains its final result, rather than only reporting the final score.

1) Iteration Is the Main Source of Repair Ability: The first pattern in Table IV is the large gap between one-shot and iterative repair. SingleAgent + no-RepoGraph solves only 5.3% of tasks, whereas IterAgent + no-RepoGraph reaches 24.7%. This indicates that execution-guided iteration is a major driver of repository-level repair performance. In many tasks, the first patch attempt is incomplete or localized to the wrong region; Docker feedback gives the system a chance to correct these mistakes in later attempts.

However, this gain comes with a cost. IterAgent + no-RepoGraph consumes 410,456 logged tokens, the highest token footprint among all controlled variants. This suggests that free-form iteration can be effective but expensive, especially when each round retrieves or carries a large amount of context. Therefore, the key challenge is not only to iterate, but to make iteration controlled and evidence-efficient.

A one-shot method may generate plausible code, but once the patch fails, it cannot revise localization, inspect adjacent code, or test another hypothesis.

2) RepoGraph Controls Context Cost but Is Not Sufficient Alone: The second pattern is that RepoGraph alone does not consistently increase RR. For SingleAgent, adding RepoGraph reduces RR from 5.3% to 0.0%. This does not mean that repository structure is useless; rather, it shows that structural filtering can hurt when the model has only one chance to generate a patch. If useful lexical evidence is filtered out in a one-shot setting, the system has no later opportunity to recover.

For IterAgent, RepoGraph keeps RR nearly unchanged but substantially reduces token usage. IterAgent + no-RepoGraph reaches 24.7% RR with 410,456 tokens, while IterAgent + RepoGraph reaches 24.0% RR with 211,299 tokens. Thus, RepoGraph is better understood as a context-control mechanism than as a standalone source of accuracy. It reduces the retrieval footprint and makes iterative search cheaper, even when the raw resolve rate remains similar.

This pattern is important for interpreting HGAR. The graph is not treated as a replacement for lexical retrieval or as a complete program-analysis oracle. Instead, HGAR uses RepoGraph as a lightweight retrieval scaffold: lexical recall keeps broad coverage, while graph reranking and conservative typed expansion reduce unnecessary context growth. This is why the current design emphasizes graph-lite behavior rather than aggressive graph-only search.

3) Reviewer Guidance Makes Iteration More Directed: HGARena further adds a reviewer ensemble on top of HGAR retrieval. The reviewer ensemble is not used to accept or reject patches; instead, it provides verifier-style guidance about likely failure causes, patch validity, and test alignment. This role matters because Docker feedback is often sparse: a failing test may identify that a patch is wrong without explaining which file, symbol, or assumption should be changed next. Reviewer guidance turns this sparse execution signal into more directed follow-up edits.

This additional guidance has a visible cost in calls: HGARena uses 13.0 calls on average, compared with 9.4 calls for IterAgent + RepoGraph and 8.2 calls for IterAgent + no-RepoGraph. However, HGARena also obtains the best RR (25.3%) while using slightly fewer tokens than IterAgent + RepoGraph and roughly half the tokens of IterAgent + no-RepoGraph. This indicates that the reviewer ensemble increases interaction count but does not simply inflate context size; instead, it helps make the additional iterations more targeted.

TABLE V: Backbone-level HGARena performance on SWE-bench Lite. All rows use the same role-structured arena protocol and official Docker validation.

Model	RR (%)	Budget	Tokens / Task
GPT-4o	25.3	13.0	204,995
Claude Sonnet 4.5	27.7	11.5	143,658
Gemini 2.5 Pro	24.3	18.5	356,646
Mean	**25.8**	**14.3**	**235,100**
Std Dev	$\pm$1.7	$\pm$3.7	$\pm$109,639
CoV	6.8%	25.7%	46.6%

A useful way to read Table IV is therefore as a cost-quality trade-off. SingleAgent is cheap but weak. IterAgent is strong but can become expensive when it freely accumulates context. RepoGraph makes iteration cheaper but does not by itself guarantee better patches. HGARena combines these effects: it keeps the repair ability of iterative search, controls the context footprint with HGAR, and uses reviewer feedback to steer the next edit.

4) Component Synergy and Trade-off: The component-level pattern is mixed but informative. Free iteration is effective but token-expensive. RepoGraph reduces context cost, but is not enough by itself and may hurt one-shot generation. HGARena combines iterative repair, RepoGraph-based context control, and reviewer-guided search.

In this sense, HGARena is not simply "IterAgent plus graph". Its value comes from the coupling between HGAR retrieval and the arena loop. HGAR decides what evidence enters the context, while the reviewer ensemble decides how that evidence should shape the next edit. This coupling explains why HGARena achieves the best RR-token balance among the controlled variants in Table IV.

D. RQ3: How Consistent Is HGARena Across Backbones?

A practical consideration for HGARena is whether the role-structured arena protocol depends strongly on a single LLM backbone. We therefore evaluate the full HGARena configuration with GPT-4o, Claude Sonnet 4.5, and Gemini 2.5 Pro under the same SWE-bench Lite setting. Each backbone instantiates both the submitter and reviewer ensemble under the same submitter–reviewer–executor protocol.

Table V shows that HGARena achieves broadly comparable resolve rates across the three backbones. The RR ranges from 24.3% to 27.7%, with a mean of 25.8% and a CoV of 6.8%. Claude Sonnet 4.5 obtains the highest RR and the lowest token usage, while Gemini 2.5 Pro obtains a comparable RR but consumes substantially more calls and tokens.

Framework-Level Consistency. The relatively small RR spread suggests that the HGARena protocol is not tied to a single backbone. Although the three models differ in generation style, verbosity, and instruction-following behavior, all remain within a 3.4 percentage-point band. This supports the view that HGARena contributes a framework-level structure—retrieval, reviewer guidance, and execution-grounded iteration—that can be instantiated with different LLMs.

This does not mean that all models behave identically. The same arena protocol can lead to different repair trajectories because each backbone may write patches with different granularity, interpret reviewer instructions differently, or consume context at different rates. Therefore, the cross-backbone result should be read as evidence of broad compatibility, not as proof that the framework eliminates model-specific behavior.

Token Variance and Cost Sensitivity. The stronger variation appears in cost rather than success rate. Budget usage ranges from 11.5 to 18.5 calls per task, and token usage ranges from 143,658 to 356,646 tokens per task. The token CoV is 46.6%, much larger than the RR CoV of 6.8%. This indicates that different backbones can achieve similar resolution quality while imposing very different cost profiles.

This cost variation matters for deployment. A slightly higher RR may not justify substantially more tokens or latency. In our results, Claude Sonnet 4.5 gives the strongest cost-performance trade-off, GPT-4o provides the main controlled setting, and Gemini 2.5 Pro remains competitive but more expensive.

E. Analysis and Discussion

1) Why HGARena Succeeds: Three Coupled Factors: The results suggest that HGARena's effectiveness comes from three coupled factors. First, execution-guided iteration is necessary because repository-level repair rarely succeeds in one shot. Second, RepoGraph and HGAR reduce the context footprint of iteration by organizing files, symbols, tests, and touched regions. Third, reviewer guidance makes the next edit more directed when Docker feedback is sparse. Together, these factors explain why HGARena achieves the best RR-token balance among the controlled variants.

Unlike pure one-shot generation, HGARena can recover from failed initial patches. Unlike free-form iteration, it does not simply accumulate large amounts of context. Unlike RepoGraph-only variants, it uses reviewer guidance to turn structured evidence into concrete follow-up edits. This explains why the method's value is not captured by any single component alone.

2) Competitive Performance on SWE-bench Lite: HGARena reaches 25.3% RR on the complete SWE-bench Lite benchmark. This is higher than SWE-agent and AutoCodeRover, and within 2.0 percentage points of Agentless. Since prior systems use their own reported configurations, this comparison should be interpreted as benchmark-level context rather than a controlled model-matched comparison. Even so, it shows that HGARena is competitive under the standard Docker-validated Lite setting.

3) Model-Agnostic but Cost-Sensitive Framework Design: The cross-backbone results suggest that HGARena can be instantiated with different LLMs while maintaining similar RR. However, the cost profile changes substantially across backbones. This means that HGARena is not model-agnostic in the sense of identical cost, but it is relatively stable in terms of resolution quality. In practice, users may choose a backbone based on cost, latency, API availability, or organizational constraints.

4) Run-to-Run Variability: A further limitation is run-to-run variability in commercial LLM calls. Even under the same task, prompt family, and budget, differences in decoding behavior, backend variation, or intermediate patch trajectories can affect whether a task is solved. A representative example is `astropy__astropy-12907` under GPT-4o: we observed both a successful Docker-validated run and failed reruns under nearby settings. This indicates that the task is reachable for the framework, but deterministic reproduction across repeated API calls is not yet guaranteed.

5) Artifact Release and Future Work: To support inspection and qualitative analysis, we release the current codebase together with representative success/failure records, including the GPT-4o `astropy__astropy-12907` example, at https://github.com/YgcCoder/HGARena. Future work should extend the evaluation to open-source backbones, repeated SWE-bench Lite reruns, and multilingual issue-resolution benchmarks. Another immediate direction is to improve unified-diff generation and patch application stability, since reviewer guidance can only help when candidate patches are valid enough to be refined by execution feedback.

V. Conclusion

We presented HGARena, a role-structured arena-style framework for repository-level GitHub issue resolution. HGARena combines heterogeneous graph-augmented retrieval—inspired by graph-based RAG methods [19], [20]—with a budgeted submitter–reviewer–executor loop grounded by Docker-based test execution [1]; reviewers provide structured search guidance rather than judge correctness, mitigating LLM-as-a-judge instability [21]–[23]. By treating retrieval as budgeted search control, the typed RepoGraph helps prioritize files, symbols, failure evidence, and edits, making iterative repair more evidence-efficient under budget. Using GPT-4o, experiments on SWE-bench Lite achieve a 25.3% Docker-validated resolve rate under a 20-call budget; variants show iteration drives repair ability, RepoGraph reduces context cost, and reviewer guidance improves the RR-token trade-off. Cross-backbone results show comparable resolve rates across GPT-4o, Claude Sonnet 4.5, and Gemini 2.5 Pro, but different costs. Future work includes extending to multilingual benchmarks [26], open-source backbones, repeated SWE-bench Lite reruns, and more stable patch generation and application.

References

[1] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world

github issues?" in *International Conference on Learning Representations*, vol. 2024, 2024, pp. 54 107–54 157.
[2] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.
[3] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.
[4] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.
[5] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*, 2016, pp. 298–312.
[6] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.
[7] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the aaai conference on artificial intelligence*, vol. 31, no. 1, 2017.
[8] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.
[9] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," *Advances in Neural Information Processing Systems*, vol. 37, pp. 50 528–50 652, 2024.
[10] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Agentless: Demystifying llm-based software engineering agents," *arXiv preprint arXiv:2407.01489*, 2024.
[11] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1592–1604.
[12] W. Xu, J. Xiong, C. Zhao, Q. Chen, H. Wang, H. Shen, Z. Wan, J. Dai, T. Wu, H. Xiao *et al.*, "Swingarena: Competitive programming arena for long-context github issue solving," *arXiv preprint arXiv:2505.23932*, 2025.
[13] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," *Advances in neural information processing systems*, vol. 36, pp. 8634–8652, 2023.
[14] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in neural information processing systems*, vol. 33, pp. 9459–9474, 2020.
[15] L. Gao, X. Ma, J. Lin, and J. Callan, "Precise zero-shot dense retrieval without relevance labels," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2023, pp. 1762–1777.
[16] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, H. Wang, H. Wang *et al.*, "Retrieval-augmented generation for large language models: A survey," *arXiv preprint arXiv:2312.10997*, vol. 2, no. 1, p. 32, 2023.
[17] P. Zhao, H. Zhang, Q. Yu, Z. Wang, Y. Geng, F. Fu, L. Yang, W. Zhang, J. Jiang, and B. Cui, "Retrieval-augmented generation for ai-generated content: A survey," *Data Science and Engineering*, pp. 1–29, 2026.
[18] W. Fan, Y. Ding, L. Ning, S. Wang, H. Li, D. Yin, T.-S. Chua, and Q. Li, "A survey on rag meeting llms: Towards retrieval-augmented large language models," in *Proceedings of the 30th ACM SIGKDD conference on knowledge discovery and data mining*, 2024, pp. 6491–6501.
[19] D. Edge, H. Trinh, N. Cheng, J. Bradley, A. Chao, A. Mody, S. Truitt, D. Metropolitansky, R. O. Ness, and J. Larson, "From local to global: A graph rag approach to query-focused summarization," *arXiv preprint arXiv:2404.16130*, 2024.
[20] Z. Guo, L. Xia, Y. Yu, T. Ao, and C. Huang, "Lightrag: Simple and fast retrieval-augmented generation," *arXiv preprint arXiv:2410.05779*, vol. 2, no. 3, 2024.
[21] L. Shi, C. Ma, W. Liang, X. Diao, W. Ma, and S. Vosoughi, "Judging the judges: A systematic study of position bias in llm-as-a-judge," in *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*, 2025, pp. 292–314.
[22] G. H. Chen, S. Chen, Z. Liu, F. Jiang, and B. Wang, "Humans or llms as the judge? a study on judgement bias," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024, pp. 8301–8327.
[23] J. Gu, X. Jiang, Z. Shi, H. Tan, X. Zhai, C. Xu, W. Li, Y. Shen, S. Ma, H. Liu *et al.*, "A survey on llm-as-a-judge," *The Innovation*, 2024.
[24] SWE-bench Team, "SWE-bench Lite," https://huggingface.co/datasets/SWE-bench/SWE-bench_Lite, 2023, accessed: 2026-05-15.
[25] OpenAI, "Introducing SWE-bench Verified," https://openai.com/index/introducing-swe-bench-verified/, 2024, accessed: 2026-05-15.
[26] D. Zan, Z. Huang, W. Liu, H. Chen, S. Xin, L. Zhang, Q. Liu, L. Aoyan, L. Chen, X. Zhong *et al.*, "Multi-swe-bench: A multilingual benchmark for issue resolving," *Advances in Neural Information Processing Systems*, vol. 38, 2026.
[27] Y. Wang, J. Keung, Z. Mao, J. Zhang, and Y. Cao, "Chart2code-mola: Efficient multi-modal code generation via adaptive expert routing," *arXiv preprint arXiv:2511.23321*, 2025.
[28] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," *arXiv preprint arXiv:2210.03629*, 2022.
[29] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 727–739.
[30] W. Wang, Y. Wang, S. Joty, and S. C. Hoi, "Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 146–158.
[31] S. Es, J. James, L. E. Anke, and S. Schockaert, "Ragas: Automated evaluation of retrieval augmented generation," in *Proceedings of the 18th conference of the european chapter of the association for computational linguistics: system demonstrations*, 2024, pp. 150–158.
[32] L. Rampášek, M. Galkin, V. P. Dwivedi, A. T. Luu, G. Wolf, and D. Beaini, "Recipe for a general, powerful, scalable graph transformer," *Advances in Neural Information Processing Systems*, vol. 35, pp. 14 501–14 515, 2022.
[33] J. Tang, Y. Yang, W. Wei, L. Shi, L. Su, S. Cheng, D. Yin, and C. Huang, "Graphgpt: Graph instruction tuning for large language models," in *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2024, pp. 491–500.
[34] R. Chen, T. Zhao, A. Jaiswal, N. Shah, and Z. Wang, "Llaga: Large language and graph assistant," *arXiv preprint arXiv:2402.08170*, 2024.
[35] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *European semantic web conference*. Springer, 2018, pp. 593–607.
[36] X. Wang, H. Ji, C. Shi, B. Wang, Y. Ye, P. Cui, and P. S. Yu, "Heterogeneous graph attention network," in *The world wide web conference*, 2019, pp. 2022–2032.
[37] K. Krippendorff, "Computing krippendorff's alpha-reliability," 2011.