

InfCode-C++: Intent-Guided Semantic Retrieval and AST-Structured Search for C++ Issue Resolution

QINGAO DONG, Beihang University, China and Beijing Tokfinity Technology Co., Ltd., China

MENGFEI WANG, HENGZHI ZHANG, and ZHICHAO LI, Beijing Tokfinity Technology Co., Ltd., China

YUAN YUAN, MU LI, XIANG GAO, HAILONG SUN, CHUNMING HU, and WEIFENG LV, Beihang University, China

Large language model (LLM) agents have recently shown strong performance on repository-level issue resolution, but existing systems are almost exclusively designed for Python and rely heavily on lexical retrieval and shallow code navigation. These approaches transfer poorly to C++ projects, where overloaded identifiers, nested namespaces, template instantiations, and deep control-flow structures make context retrieval and fault localization substantially more difficult. As a result, state-of-the-art Python-oriented agents show a drastic performance drop on the C++ subset of MultiSWE-bench. We introduce InfCode-C++, the first C++-aware autonomous system for end-to-end issue resolution. The system combines two complementary retrieval mechanisms—semantic code-intent retrieval and deterministic AST-structured querying—to construct accurate, language-aware context for repair. These components enable precise localization and robust patch synthesis in large, statically typed C++ repositories. Evaluated on the MultiSWE-bench-CPP benchmark, InfCode-C++ achieves a resolution rate of 25.58%, outperforming the strongest prior agent by 10.85 percentage points and more than doubling the performance of MSWE-agent. Ablation and behavioral studies further demonstrate the critical role of semantic retrieval, structural analysis, and accurate reproduction in C++ issue resolution. InfCode-C++ highlights the need for language-aware reasoning in multi-language software agents and establishes a foundation for future research on scalable, LLM-driven repair for complex, statically typed ecosystems.

ACM Reference Format:

Qingao Dong, Mengfei Wang, Hengzhi Zhang, Zhichao Li, Yuan Yuan, Mu Li, Xiang Gao, Hailong Sun, Chunming Hu, and Weifeng Lv. 2025. InfCode-C++: Intent-Guided Semantic Retrieval and AST-Structured Search for C++ Issue Resolution. 1, 1 (November 2025), 17 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 INTRODUCTION

Automated software issue resolution stands as a critical challenge in software engineering, promising to significantly reduce developer workload and accelerate maintenance cycles. The emergence of comprehensive benchmarks, most notably SWE-bench [1], which evaluates agents on thousands of real-world GitHub issues, has been pivotal in measuring progress. On this benchmark, which is predominantly composed of Python repositories, state-of-the-art LLM-powered agents like Trae Agent have demonstrated impressive capabilities, achieving resolution rates as high as 75.20% [2]. However, this success in memory-safe, dynamically-typed languages masks a significant performance gap in other critical domains. Specifically, resolving issues in large-scale,

Authors' Contact Information: Qingao Dong, Beihang University, Beijing, China and Beijing Tokfinity Technology Co., Ltd., Beijing, China; Mengfei Wang; Hengzhi Zhang; Zhichao Li, Beijing Tokfinity Technology Co., Ltd., Beijing, China; Yuan Yuan; Mu Li; Xiang Gao; Hailong Sun; Chunming Hu; Weifeng Lv, Beihang University, Beijing, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM XXXX-XXXX/2025/11-ART

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

high-performance systems written in C++ remains a formidable, unsolved challenge. Recent evaluations on multilingual benchmarks, such as MultiSWE-bench [3], highlight this disparity; a leading agent configuration, Mopenhands paired with Claude 3.7 Sonnet, achieved a resolution rate of only 14.7% on C++ tasks [4]. This stark performance drop underscores that current methodologies are ill-equipped to handle the distinct complexities of C++, necessitating a shift from general-purpose approaches to language-specific, structurally-aware solutions.

Ambiguous Context Localization without Precise Artifacts. A primary challenge in C++ issue resolution is ambiguous context localization, particularly when the issue description lacks precise artifacts such as a stack trace. This problem is exacerbated for non-crashing bugs, such as logical errors or performance bottlenecks, where no immediate fault signal is available. Unlike a crash bug which pinpoints a faulting location, these issues require the agent to deduce the relevant code context from a natural language description alone. In a sprawling, million-line C++ codebase, this task becomes a “needle in a haystack” problem. Existing approaches [4–6] often resort to rudimentary heuristics, attempting to guess relevant file, class, or function names based on superficial lexical cues in the issue report. This approach is inefficient, unreliable, and frequently retrieves irrelevant context, forcing the LLM to analyze hundreds or even thousands of lines of unrelated code, thus leading to incorrect or incomplete patches [7].

Insufficiency of Lexical Search in Complex Code Structures. This localization difficulty is compounded by the inadequacy of standard lexical search tools. Many contemporary agents rely on ‘grep’ utilities for context retrieval [2, 4, 6, 8]. While effective for simple pattern matching, these tools are fundamentally limited and ill-suited for C++. The structural complexity of C++, characterized by features such as namespaces, function and operator overloading, intricate template metaprogramming, and deep inheritance hierarchies—renders text-based search insufficient. For example, a ‘grep’ query for a function named ‘update’ cannot semantically disambiguate between ‘UI::update()’, ‘Database::update()’, and a similarly named method in a base class. It also fails to resolve virtual function calls or understand template instantiations. This lexical ambiguity pollutes the context provided to the LLM with a high volume of irrelevant and misleading information, obscuring the true semantic relationships and call graphs, and ultimately hindering its ability to perform robust bug resolution.

To address these specific challenges in C++ repositories, we propose a novel, two-pronged approach that enhances an LLM agent’s ability to understand and navigate complex C++ codebases. Our method combines a high-level, semantic retrieval strategy based on code intent with a low-level, precise structural search mechanism based on the Abstract Syntax Tree (AST).

First, we introduce a semantic context retrieval framework based on *code intent*. This method moves beyond simple file-based retrieval by associating disparate software artifacts, such as files, classes, and functions, that collectively implement a specific, high-level feature. This association allows an LLM agent to first query the codebase using natural language to identify relevant functional modules (e.g., “locate components responsible for data serialization”). Once this high-level context is established, the agent can execute more granular queries to inspect the specific code blocks necessary for the fix, effectively narrowing the search space from the entire repository to a few relevant components.

Second, to “go beyond grep” and overcome the limitations of lexical search, we develop an AST-based structured query engine. This engine parses the C++ codebase into its abstract syntax tree, enabling precise queries based on code structure rather than text. We expose this capability to the LLM agent through a suite of robust tools, including ‘FindClass’, ‘FindFunction’, and ‘GetInheritanceChain’. These tools empower the agent to navigate C++’s complex structures deterministically, find precise definitions, and understand critical relationships (e.g., class hierarchies or virtual function overrides) that are invisible to text-based tools.

We thus present **InfCode-C++**, an autonomous agent that integrates these two capabilities to systematically resolve C++ issues. Starting from a natural language issue description, **InfCode-C++** first employs the code-intent semantic retrieval to perform a high-level semantic search, identifying a small set of candidate functional modules relevant to the issue. Subsequently, the agent deploys its AST-based search tools to conduct a precise, structural analysis within these modules. This two-stage process—broad semantic filtering followed by deep structural validation—allows the agent to resolve ambiguities like function overloading and namespace conflicts. This procedure provides the LLM with a concise, accurate, and semantically-rich context that is not merely lexically relevant but structurally correct, facilitating the construction of a robust patch.

The main contributions of this work are as follows:

- **Semantic Code-Intent RAG:** We propose a novel semantic retrieval framework that maps high-level functional descriptions (the intent) to relevant, multi-file code artifacts. This allows an agent to bypass ambiguous lexical search and instead retrieve context based on feature-level understanding.
- **Structural-Aware AST Querying:** We develop a set of agent-usable tools that operate directly on the C++ AST. This mechanism, featuring tools like ‘FindClass’ and ‘GetInheritanceChain’, overcomes the limitations of text-based search by providing deterministic, semantically-correct navigation of complex C++ structures.
- **High-Efficacy Agent and Public Artifact:** We design and implement **InfCode-C++**, an agent integrating our two-pronged retrieval system. Our tool shows high efficacy, solving **25.58%** of issues on the C++ subset of MultiSWE-bench [3], significantly outperforming prior state-of-the-art. To support the research community and facilitate future work, we release **InfCode-C++** and our evaluation benchmark as an open-source project at <https://github.com/Tokfinity/InfCode>.

2 Motivation Example

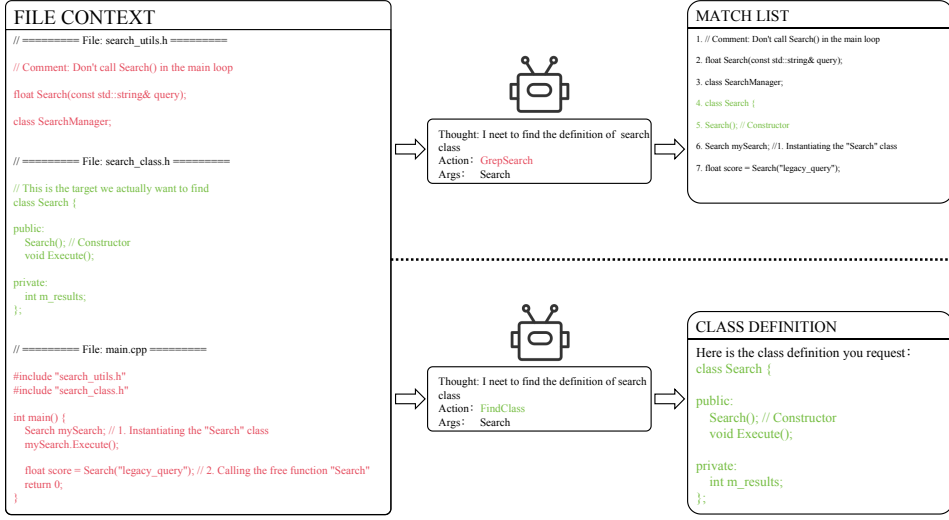
We present a motivating example that highlights the core challenges of C++ issue resolution and the necessity of the retrieval mechanisms introduced in **InfCode-C++**. The example is adapted from a real-world multi-file C++ project, and its structure is illustrated in Figure 1.

Lexical retrieval fails in the presence of overloaded identifiers. The file context shown on the left side of the figure contains three files: `search_utils.h`, which defines a free function `Search`; `search_class.h`, which defines a class `Search` with a constructor and a member function; and `main.cpp`, which uses both the class `Search` and the free function `Search`. This mixture of identically named entities is common in C++ due to overloaded functions, namespace scoping, forward declarations, and project-specific naming conventions.

When an agent attempts to locate the definition of “the `Search` class,” a lexical retrieval tool such as `grep` matches every appearance of the string “`Search`,” producing the heterogeneous match list in the top-right of Figure 1. The list includes comments, forward declarations, free functions, class names, constructor invocations, and call sites. Crucially, lexical search cannot distinguish between:

- a free function `Search(const std::string&)`,
- a forward declaration `class SearchManager`,
- the actual class definition `class Search { ... }`, and
- uses of the class constructor or variable instantiations.

As a result, lexical tools provide an over-approximate and noise-dominated context. The agent must manually filter a large number of irrelevant matches to locate the true class definition.



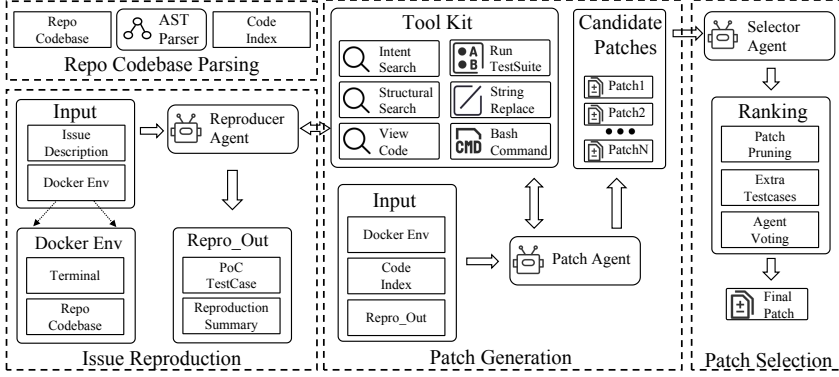


Fig. 2. Overall system architecture of InfCode-C++. The framework consists of three collaborative agents: (1) the Reproducer Agent, which reconstructs the failure and produces a concrete reproducing test; (2) the Patch Agent, which performs semantic intent retrieval, AST-structured querying, and multi-step patch synthesis using a rich tool kit; (3) the Selector Agent, which prunes, validates, and ranks candidate patches through regression testing, extra test generation, and agent voting.

This task is especially challenging in C++ repositories due to structural complexity, overloaded identifiers, namespace shadowing, templates, and deep inheritance hierarchies, which significantly hinder reliable context retrieval and localization. InfCode-C++ addresses these challenges through a structured, multi-agent design.

3.2 System Overview

InfCode-C++ is implemented as a modular, autonomous multi-agent framework comprising three cooperating agents: (1) the *Reproducer Agent*, (2) the *Patch Agent*, and (3) the *Selector Agent*.

The system architecture includes four main stages—*Repository Parsing*, *Issue Reproduction*, *Patch Generation*, and *Patch Selection*—which correspond directly to the workflow illustrated in Figure 2.

- **Repository Parsing:** Prepares the repository by constructing an AST-based structural index and a semantic code-intent index to support subsequent retrieval.
- **Issue Reproduction:** Converts the natural-language issue description into an executable proof-of-concept (PoC) test case and a reproduction summary.
- **Patch Generation:** Performs C++-aware context retrieval—combining semantic intent search with AST structural queries—to localize faults and synthesize candidate patches.
- **Patch Selection:** Performs a hierarchical patch selection process that begins with patch pruning, which merges semantically equivalent candidates and removes formatting-only or comment-only edits; proceeds to behavioral validation, where the Selector Agent generates additional test cases to filter out fault-preserving patches; and concludes with an agent-voting mechanism that ranks the remaining candidates and selects the final patch.

This pipeline allows InfCode-C++ to combine high-level semantic reasoning with low-level structural precision, enabling robust fault localization and patch synthesis in complex C++ repositories.

3.3 Repository Parsing

Before issue resolution begins, the repository is statically analyzed to build two retrieval structures essential for C++ code navigation:

AST-Based Structural Index. The repository is parsed into an abstract syntax tree:

$$\mathcal{T}_C = (\mathcal{N}, \mathcal{E}),$$

where nodes \mathcal{N} represent syntactic constructs (e.g., classes, methods, template instantiations) and edges \mathcal{E} capture structural relations such as inheritance, call edges, containment, and overload groups. This index enables deterministic structural queries that avoid ambiguities inherent to lexical search.

Semantic Code-Intent Index. We further construct a semantic embedding index that maps feature-level intents to code artifacts:

$$\mathcal{I}_{\text{intent}} : \text{Intent} \mapsto \{A_1, A_2, \dots, A_k\},$$

where each A_i is a file, class, or function participating in the implementation of a high-level feature. This index enables intent-driven retrieval of relevant subsystems.

Together, these two indices form the core of INFCode-C++'s C++-aware retrieval toolkit.

3.4 Issue Reproduction

The *Reproducer Agent* operationalizes the issue description by producing an executable test case t_D that deterministically reproduces the failure. This requires extracting execution conditions, inputs, expected and observed behaviors, and environmental dependencies.

The agent interacts with a sandboxed Docker environment containing the repository snapshot and produces:

- **PoC Test Case** capturing the erroneous behavior.
- **Reproduction Summary** including execution traces and environment details.

This stage provides a concrete behavioral oracle for validating and ranking candidate patches in later stages.

Formally, the reproducer defines a partial behavioral specification:

$$\text{Exec}(C, t_D) = \text{Fail}.$$

3.5 Patch Agent: Context Retrieval and Patch Generation

The *Patch Agent* is responsible for fault localization and candidate patch generation. Unlike Python-focused agents, which often rely on lexical retrieval, INFCode-C++ integrates two complementary retrieval mechanisms specifically designed for C++.

3.5.1 Semantic Retrieval via Code Intent. Given the issue description D , the agent constructs a query q and invokes:

$$\text{QueryCodeIntent}(q) \rightarrow C_{\text{intent}},$$

where C_{intent} is a semantically coherent subset of the repository. This step identifies the subsystem implementing the relevant feature(s). Since:

$$|C_{\text{intent}}| \ll |C|,$$

this significantly eliminates irrelevant modules before structural reasoning.

3.5.2 Structural Retrieval via AST Querying. Within C_{intent} , the agent performs precise structural analysis by issuing queries such as:

- `FindClass(className)` (locates class definitions across nested namespaces)
- `FindFunction(spec, name)` (locates function definitions within the specified scope and resolves overloads)
- `GetInheritanceChain(className)` (extracts base and derived class hierarchies)
- `GetFunctionCalls(className, functionName)` (computes intra-class and inter-module call graphs)

Because these operations rely on AST structure rather than surface tokens, they handle overloaded identifiers, template instantiation, and nested namespaces—scenarios where lexical baselines fail.

The intersection of semantic and structural evidence yields the localized fault region:

$$L = I_{\text{intent}}(D) \cap \mathcal{G}_{\text{bug}},$$

where \mathcal{G}_{bug} is the structurally derived defect subgraph.

The Patch Agent then synthesizes candidate patches $\{p_1, \dots, p_n\}$ conditioned on (D, L, C_L) .

3.6 Patch Selection

The *Selector Agent* performs a hierarchical refinement procedure to identify the final patch from the set of candidates $\{p_1, \dots, p_n\}$ generated by the Patch Agent. This procedure consists of three stages: patch pruning, behavioral validation, and agent voting.

Patch Pruning. Given a candidate patch p_i , the Selector first applies a pruning function $\text{Prune}(\cdot)$ that removes edits that do not affect program behavior. This includes formatting-only changes, comment-only modifications, and semantically equivalent patches that differ only in syntactic form. The pruning stage reduces redundancy and eliminates candidates that cannot contribute to behavioral correction. Formally, pruning yields:

$$\widehat{p}_i = \text{Prune}(p_i),$$

and only behaviorally distinct patches are retained for further evaluation.

Behavioral Validation. For each pruned candidate \widehat{p}_i , the Selector applies the patch to the repository:

$$C'_i = C \oplus \widehat{p}_i.$$

The agent then validates C'_i using both the reproducer test t_D and additional Selector-generated test cases $\mathcal{T}_{\text{extra}}$:

$$t_D(C'_i) = \text{PASS} \quad \wedge \quad \forall t \in \mathcal{T}_{\text{extra}}, t(C'_i) = \text{PASS}.$$

Candidates failing any test are discarded. The surviving set is denoted $\mathcal{P}_{\text{valid}}$.

Agent Voting. If multiple candidates remain, the Selector triggers an LLM-based voting mechanism that scores each patch according to its semantic alignment with the issue description D , its minimality, and its consistency with the localized defect region. The final patch is selected as:

$$p_{\text{final}} = \arg \max_{p_i \in \mathcal{P}_{\text{valid}}} \text{VoteScore}(p_i, D).$$

This hierarchical design ensures that non-impactful edits are removed early, fault-preserving patches are filtered through behavioral validation, and the final decision reflects semantically grounded reasoning rather than syntactic similarity alone.

Algorithm 1 End-to-End Workflow of INFCode-C++

Require: Software codebase C , Issue description D , Regression test suite T

Ensure: Validated patch p_{final} satisfying D and preserving T

```

1: function REPRODUCE( $C, D$ )
2:   Parse  $D$  to extract input configuration, triggering behavior, and expected outcome
3:   Generate executable reproducer test  $t_D$ 
4:   Verify Exec( $C, t_D$ ) = Fail
5:   return  $t_D$ 

6: function PATCH( $C, D, t_D$ )
7:    $C_{\text{intent}} \leftarrow \text{QUERYCODEINTENT}(D)$                                 ▶ Semantic retrieval
8:    $\mathcal{T}_C \leftarrow \text{PARSETOAST}(C_{\text{intent}})$                             ▶ AST construction
9:    $L \leftarrow \text{LOCATEBUG}(\mathcal{T}_C, D)$                                     ▶ Structural localization
10:   $\{p_1, \dots, p_n\} \leftarrow \text{GENERATEPATCHES}(L, D)$ 
11:  return  $\{p_1, \dots, p_n\}$ 

12: function SELECT( $C, T, t_D, \{p_1, \dots, p_n\}$ )
13:   $\mathcal{P}_{\text{valid}} \leftarrow \emptyset$ 
14:  for  $i \leftarrow 1$  to  $n$  do
15:     $p_i \leftarrow p_i$                                                     ▶ Candidate patch
16:     $C'_i \leftarrow C \oplus p_i$ 
17:    if  $t_D(C'_i) = \text{PASS}$  and  $\forall t \in T : t(C'_i) = t(C)$  then
18:       $\mathcal{P}_{\text{valid}} \leftarrow \mathcal{P}_{\text{valid}} \cup \{p_i\}$ 
19:    if  $\mathcal{P}_{\text{valid}} = \emptyset$  then
20:      return FAILURE
21:    else
22:       $p_{\text{final}} \leftarrow \arg \min_{p_i \in \mathcal{P}_{\text{valid}}} \text{Complexity}(p_i)$ 
23:      return  $p_{\text{final}}$ 

24:  $t_D \leftarrow \text{REPRODUCE}(C, D)$ 
25:  $\{p_1, \dots, p_n\} \leftarrow \text{PATCH}(C, D, t_D)$ 
26:  $p_{\text{final}} \leftarrow \text{SELECT}(C, T, t_D, \{p_1, \dots, p_n\})$ 
27: return  $p_{\text{final}}$ 

```

3.7 End-to-End Execution Flow

The complete workflow of INFCode-C++ follows a deterministic pipeline:

$$(C, D) \xrightarrow{\text{Reproducer Agent}} t_D \xrightarrow{\text{Patch Agent}} \{p_1, \dots, p_n\} \xrightarrow{\text{Selector Agent}} p_{\text{final}}.$$

By combining semantic intent analysis, AST-based structural reasoning, and hierarchical patch refinement, INFCode-C++ achieves robust issue resolution for structurally complex C++ repositories.

The complete procedure of INFCode-C++ is summarized in Algorithm 1.

Table 1. Repository statistics of the MultiSWEbenchCPP benchmark.

Repository	Files	Locs	Instances
catchorg/Catch2	399	58.0k	12
fmtlib/fmt	25	36.4k	41
nlohmann/json	477	124.7k	55
simdjson/simdjson	455	229.7k	20
yhirose/cpp-httpplib	33	50.9k	1

4 EXPERIMENTAL SETUP

4.1 Benchmark

We evaluate INFCode-C++ on the C++ subset of the MultiSWE-bench benchmark, denoted as MultiSWEbenchCPP. As shown in Table 1, the benchmark contains 129 real-world GitHub issues from actively maintained C++ repositories. Each instance provides a natural-language issue description, a code snapshot, and a regression test suite. Agents must synthesize a patch that satisfies the behavioral requirements described in the issue report while preserving all existing behaviors validated by the test suite.

All 129 instances are drawn from five actively maintained C++ libraries, whose statistics are summarized in Table 1. The repositories range from `fmtlib/fmt` with 25 files and 36.4k lines of code to `nlohmann/json` and `simdjson/simdjson` with 477 and 455 files and 124.7k and 229.7k lines of code respectively, while `catchorg/Catch2` and `yhirose/cpp-httpplib` contribute additional large, header-heavy codebases. This scale and modularity substantially increase the search space for localization compared to typical Python benchmarks. All instances include ground-truth patches, enabling objective and reproducible correctness evaluation. While some issue descriptions provide stack traces or partial execution logs, such information is inconsistent and often incomplete in these C++ projects (due to inlining, template instantiation, and namespace resolution), so agents cannot rely on auxiliary execution signals and must localize faults primarily from the natural-language description and the codebase itself.

4.2 Baselines

We compare INFCode-C++ with three state-of-the-art agent-based repair systems: Mopenhands [4], MSWE-Agent [6], and MAgentless [5]. These systems represent the strongest publicly available baselines on MultiSWE-bench, covering tool-driven, hybrid, and model-centric repair strategies.

4.3 Evaluation Protocol

All methods are evaluated using the official MultiSWE-bench pipeline. A patch is considered correct if and only if it passes both the issue-specific reproducer test and the full regression test suite. Each issue is evaluated independently, and each agent is given a single attempt per instance to ensure strict comparability.

4.4 Agent Configuration

INFCode-C++ is evaluated under two backend large language models: GPT-5-20250807 [9] and DeepSeek-V3 [10]. The system adopts a three-agent configuration consisting of the Reproducer Agent, the Patch Agent, and the Selector Agent.

Reproducer Agent. This agent converts the natural-language issue description into an executable reproducer script. We set a maximum interaction budget of 20 LLM turns to balance generation complexity and stability.

Patch Agent. The Patch Agent performs context retrieval and patch synthesis using the full pipeline in Section 3, which integrates semantic code-intent retrieval and AST-structured searching. We allow up to 50 LLM turns for this agent, enabling multi-step reasoning across semantic and structural contexts. During test-time scaling, the agent generates 10 candidate patches conditioned on the localized context. These candidates are subsequently filtered, pruned, and validated in later stages of the pipeline.

Selector Agent. The Selector Agent receives all candidate patches from the Patch Agent and conducts behavioral validation. It executes candidate patches against the reproducer test and regression suite, prunes semantically equivalent or comment-only edits, and applies agent voting to identify the final patch. The Selector Agent does not require additional LLM turns beyond those used for ranking and semantic comparison during patch selection.

Baselines. We directly adopt the published results for all competing systems without modifying their agent configurations, toolchains, prompt formats, or execution constraints. This ensures that our comparisons faithfully reflect the behavior of each system under the official MultiSWE-bench evaluation protocol.

4.5 Metric

We report *issue resolution rate*, defined as the proportion of issues for which an agent produces a correct patch under the evaluation protocol outlined above. This is the primary metric used by MultiSWE-bench and reflects the end-to-end repair capability of each system.

4.6 Research Questions

We structure our empirical study around the following three research questions.

RQ1: How effectively does INFCode-C++ resolve C++ issues on the MultiSWE-bench-CPP benchmark?

This research question evaluates the end-to-end repair capability of INFCode-C++ under the official MultiSWE-bench evaluation protocol. It measures whether the proposed retrieval and reasoning mechanisms improve issue resolution rates relative to existing state-of-the-art agents. This RQ establishes the overall effectiveness of our system in real-world C++ repair settings.

RQ2: What is the contribution of each component in INFCode-C++'s retrieval and reasoning pipeline?

This research question examines the impact of the system's key components through ablation experiments. We isolate the effects of semantic code-intent retrieval and AST-structured querying, and assess the performance of the Reproducer Agent and Patch Agent under restricted configurations. The goal is to determine which capabilities are essential for achieving robustness on C++ repositories and to quantify their individual contributions.

RQ3: What are the behavioral characteristics of INFCode-C++ across its internal stages, specifically in terms of reproduction success and localization accuracy?

This research question investigates the internal behavior of the system by examining two key metrics. The first is the reproduction success rate, which measures the ability of the Reproducer Agent to construct an executable test that triggers the issue. The second is the localization success rate, which quantifies whether the Patch Agent identifies the correct defect region before generating

Table 2. Issue resolution rates on MultiSWEbenchCPP. The InfCode models are compared with leading baselines using Claude 3.7 Sonnet and DeepSeek-V3. For settings where difficulty-level breakdown is not provided, we report “–”.

Model	Overall (%)	Easy (%)	Medium (%)	Hard (%)
InfCode + GPT-5	25.58	50.00	25.42	9.52
InfCode + DeepSeek-V3	13.20	–	–	–
<i>Baselines using Claude 3.7 Sonnet</i>				
Mopenhands + Claude-3.7-Sonnet	14.73	32.14	15.25	2.38
MSWE-agent + Claude-3.7-Sonnet	11.63	28.57	11.86	0.00
MAgentless + Claude-3.7-Sonnet (Oct)	3.88	10.71	3.39	0.00
<i>Baselines using DeepSeek-V3</i>				
Mopenhands + DeepSeek-V3	7.75	10.71	10.17	2.38
MSWE-agent + DeepSeek-V3	7.75	17.86	8.47	0.00
MAgentless + DeepSeek-V3	1.55	3.57	0.00	0.00

a patch. These measurements provide insight into where failures occur in the repair pipeline and help explain the end-to-end performance observed in RQ1.

4.7 RQ1: Overall Issue Resolution Performance

Table 2 reports the end-to-end issue resolution rates of InfCode-C++ and all baselines on MultiSWEbenchCPP. InfCode-C++ equipped with GPT-5 achieves the highest overall performance, resolving 25.58% of all issues, which represents a substantial improvement over prior systems. The strongest baseline, MOpenHands + Claude-3.7 Sonnet, resolves 14.73%, while MSWE-agent and MAgentless achieve 11.63% and 3.88%, respectively. Thus, InfCode-C++ exceeds the strongest Claude-3.7-based baseline by 10.85 absolute percentage points and achieves more than twice the resolution rate of MSWE-agent.

When using DeepSeek-V3, InfCode-C++ obtains 13.20% resolution accuracy, again surpassing all DeepSeek-based baselines by a clear margin. MOpenHands, MSWE-agent, and MAgentless with DeepSeek-V3 achieve 7.75%, 7.75%, and 1.55%, respectively. This demonstrates that InfCode-C++ consistently improves C++ repair performance across different LLM backends.

A difficulty-level analysis further highlights the robustness of InfCode-C++. With GPT-5, InfCode-C++ resolves 50.00% of easy issues, 25.42% of medium issues, and 9.52% of hard issues, all of which exceed the best-performing baselines by substantial margins.

For example, on hard problems, the most structurally complex C++ issues involving deep template instantiation chains or multi-file dependency interactions—MOpenHands + Claude-3.7 achieves only 2.38%, whereas InfCode-C++ achieves 9.52%.

Beyond aggregate resolution rates, we further analyze the instance-level repair coverage of InfCode-C++ compared with representative baselines. Figure 3 presents a five-way Venn diagram spanning InfCode (GPT-5 and DeepSeek-V3) and three Claude-3.7-based agents (MOpenHands, MSWE-agent, and MAgentless).

InfCode + GPT-5 not only subsumes the majority of issues repaired by all baselines but also contributes a non-trivial set of uniquely solved instances, 6 issues are repaired exclusively by InfCode + GPT-5, and none of the other four systems are able to fix them. This demonstrates that InfCode-C++ expands the solvable region of the benchmark rather than merely overlapping with existing capabilities.

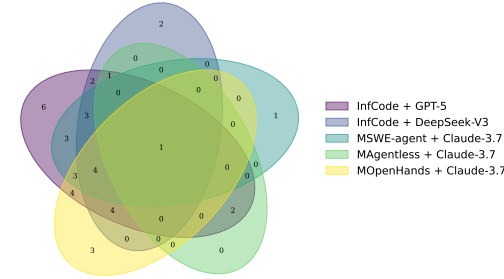


Fig. 3. Instance-level overlap of resolved issues on MultiSWEbenchCPP across InfCode and representative baselines.

Table 3. Ablation study of INFCode-C++ on MultiSWEbenchCPP. All configurations retain the Patch Agent; individual retrieval, reasoning, and validation components are removed to quantify their contributions.

Configuration	Resolution (%)
Full System (GPT-5)	25.58
<i>Ablations (Patch Agent retained)</i>	
w/o Code-Intent Retrieval	19.37%
w/o AST-Structured Querying	17.05%
w/o Reproducer Agent	20.16%
w/o Selector Agent	22.48%

Overall, the results demonstrate that INFCode-C++ provides significant and consistent improvements over existing agent-based repair systems. This confirms that the proposed semantic and structural retrieval mechanisms are effective for resolving real-world C++ issues, which pose substantially greater localization and reasoning challenges than their Python counterparts.

4.8 RQ2: Contribution of Retrieval and Reasoning Components

Table 3 presents the ablation study for INFCode-C++ on MultiSWEbenchCPP. All variants retain the Patch Agent so that the differences in performance reflect the contribution of the retrieval, reproduction, and validation mechanisms rather than the underlying patch synthesis capability.

Removing semantic code-intent retrieval reduces the resolution rate from 25.58% to 19.37%, an absolute drop of 6.21 percentage points. This demonstrates that high-level semantic filtering is crucial for constraining the search space in large C++ repositories. Since many issues lack direct fault indicators, the absence of semantic retrieval deprives the Patch Agent of coherent feature-level context, significantly weakening its ability to identify the relevant defect region.

Disabling AST-structured querying yields an even larger absolute reduction, with performance dropping from 25.58% to 17.05%, a decrease of 8.53 points. This confirms that structural code search is a core requirement for C++ repair. C++ projects contain deeply nested class hierarchies, overloaded member functions, template instantiations, all of which introduce structural ambiguity that lexical search cannot resolve. AST-based querying provides deterministic disambiguation of these constructs, enabling precise localization and yielding a larger performance impact than semantic retrieval alone.

Removing the Reproducer Agent decreases performance to 20.16%, a 5.42 points drop. Without a faithful executable test derived from the issue description, the system loses the behavioral specification that constrains the localization process and validates patch candidates. This leads to higher rates of both under-constrained patch generation and false positives that would otherwise be rejected.

Finally, removing the Selector Agent reduces resolution accuracy to 22.48%, an absolute loss of 3.10 points. The Selector Agent performs structured patch pruning by removing redundant edits, filtering semantically equivalent or comment-only modifications, and applying LLM-based ranking to choose the most behaviorally consistent candidate. The degradation shows that these verification and ranking steps meaningfully increase the likelihood that the final patch is correct.

Overall, the ablation results indicate that the performance improvements reported in RQ1 arise from the combined effect of semantic retrieval, structural analysis, reproducer construction, and

selector-guided filtering. Among these, semantic intent retrieval and AST-structured querying contribute the largest absolute gains, demonstrating their necessity for robust repair in complex C++ codebases.

Beyond correctness, we also examine the effect of removing retrieval components on the reasoning efficiency of the Patch Agent. We measure efficiency using the number of LLM iterations required to synthesize a patch. In the full system, the Patch Agent requires an average of 28.1 turns. Removing semantic code-intent retrieval increases the average iteration count to 35.3 turns, while removing AST-structured querying leads to a substantial increase to 45.3 turns.

These results indicate that both retrieval mechanisms not only improve repair accuracy but also significantly reduce the reasoning burden on the LLM. Without semantic retrieval, the agent must explore a substantially larger portion of the repository before reaching a plausible localization hypothesis. Without AST-based structural analysis, the agent increasingly relies on trial-and-error reasoning to resolve ambiguities in overloaded members, template instantiations, and cross-file dependencies. Thus, retrieval-guided reasoning is essential not only for accuracy but also for maintaining tractable search trajectories in C++ repair tasks.

4.9 RQ3: Behavioral Analysis of Internal Pipeline Stages

To understand where failures occur within the repair pipeline, we analyze the internal behavior of INFCode-C++ across three stages: (1) reproduction of the issue, (2) localization of the defect, and (3) final end-to-end resolution. Table 4 summarizes the results.

Reproduction Success Rate. The Reproducer Agent successfully constructs an executable failing test for 28.81% of issues. This indicates that more than half of the C++ issues cannot be reproduced from the textual description alone. Compared to Python-based benchmarks, C++ issues often rely on build-system configurations, platform-specific behaviors, or cross-module runtime states that are not explicitly described in the issue report. Consequently, missing environment assumptions and implicit dependencies make reproduction failures a major limiting factor in C++ repair.

Localization Success Rate. Localization performance is analyzed at two granularities. At the file level, the Patch Agent identifies the correct file containing the defect for 55.10% of issues. At the function level, accuracy decreases to 42.10%, reflecting the increased difficulty of pinpointing defects in fine-grained C++ program structures. This gap highlights the inherent ambiguity in C++ codebases, where overloaded member functions, template instantiations, namespace shadowing, and deep inheritance hierarchies obscure semantic relationships that are more explicit in languages like Python. Even with semantic code-intent retrieval and AST-structured querying, mislocalization remains the most common failure source after reproduction.

End-to-End Success. Ultimately, INFCode-C++ resolves 25.58% of issues, as reported in RQ1. By correlating the three behavioral metrics, we observe a clear pattern: (1) issues that fail to reproduce cannot proceed further in the pipeline, and (2) issues that reproduce but mislocalize seldom yield correct patches, even when patch synthesis succeeds locally. Thus, the primary failure modes lie in reconstructing the execution context and identifying the correct structural region of the defect, rather than in the patch generation step.

Overall, the behavioral analysis demonstrates that C++ repair presents fundamental challenges that differ significantly from Python-based settings. Reproduction requires inferring complex runtime contexts, while localization requires resolving structural ambiguities intrinsic to the C++ language. The results confirm that INFCode-C++'s retrieval and structural reasoning components directly target the most difficult stages of the pipeline, enabling substantial improvements in end-to-end repair capability.

Table 4. Behavioral analysis of INFCode-C++ on MultiSWEbenchCPP. We report the success rate of each internal stage in the repair pipeline.

Metric	Rate (%)
Reproduction Success Rate	28.81
File-Level Localization Success Rate	55.10
Function-Level Localization Success Rate	42.10
End-to-End Resolution Rate	25.58

5 RELATED WORK

5.1 Automatic Software Issue Resolution

Automatic software issue resolution has received significant attention in recent years, leading to a growing body of work centered around LLM-driven agents [2, 4–6, 8, 11–20]. Early agent frameworks such as openhands [4] introduced a tool-augmented planning mechanism that enables iterative interaction with the environment through file editors, terminals, and search tools. SWE-agent [6] extends this idea by providing a dedicated agent–computer interface tailored for repository-level modification, allowing the agent to inspect, edit, and validate code through structured operations.

Subsequent systems focus on improving retrieval and localization. Moatless [21] enhances issue resolution by incorporating code search utilities and retrieval strategies designed to identify potentially relevant locations within a Python codebase. AutoCodeRover [12] introduces a structural representation of source code based on abstract syntax trees (ASTs), enabling more precise localization compared with lexical search. Building on this design, SpecRover [17] integrates function-level summarization and automatic reproduction test generation to improve localization and verification fidelity. Agentless [5] adopts a fixed operational pipeline—fault localization, patch generation, and patch validation—without requiring tool-based exploration, while MarsCode Agent [22] employs code knowledge graphs combined with LLM reasoning to support localization and candidate patch generation.

Beyond retrieval-oriented approaches, several recent systems explore alternative mechanisms to enhance agent robustness and patch quality. SWE-EXP [15] and EXPERepair [13] incorporate long-horizon historical memory, enabling agents to accumulate and leverage prior interaction trajectories to guide future repair attempts. SWE-Debate [14] introduces a multi-agent debate protocol in which multiple LLMs generate competing reasoning chains and critique each other, improving the likelihood that the system converges on the correct repair trajectory. Trae [2] further advances this direction by combining test-time scaling [19] with a selector agent that evaluates candidate patches using behavioral signals, achieving state-of-the-art results on the SWE-bench-verified [23] benchmark. These techniques highlight the growing shift toward iterative, ensemble-based, or interaction-driven reasoning; however, they remain optimized for Python environments and do not address the structural and semantic complexities inherent to C++ codebases.

Despite architectural differences, all of these systems share a fundamental characteristic: they have been designed, evaluated, and optimized almost exclusively for *Python* repositories. The benchmarks used to validate them, most notably SWE-bench [1], are dominated by dynamically typed, memory-safe Python projects. Under these conditions, state-of-the-art systems achieve strong performance; for example, MOpenHands combined with Claude 3.7 Sonnet [24] resolves 52.2% of SWE-bench Python issues. However, when evaluated on the C++ subset of MultiSWE-bench, the same configuration achieves only 14.73%. This sharp degradation demonstrates that

general-purpose agent designs do not transfer effectively to statically typed, structurally complex languages.

The difficulty arises from the nature of C++ codebases. Identifier overloading, namespace shadowing, template instantiation, deep inheritance hierarchies, and cross-module control flow cause simple lexical retrieval—even when paired with powerful LLMs—to produce ambiguous or misleading context. Python-oriented agents typically rely on heuristics such as keyword matching, directory-level search, or shallow name-based filtering. These retrieval mechanisms are inadequate for C++, where the same identifier may denote distinct functions, templates, or methods across multiple nested namespaces, and where the syntactic and semantic relationships required for fault localization cannot be recovered from surface-level cues.

Furthermore, the majority of prior work assumes the availability of explicit execution traces or easily inferable failure points. In C++, non-crashing logic bugs frequently lack observable fault indicators; without symbolic traces or stack information, agents must infer the fault location solely from the semantic structure of the repository. Existing Python-oriented systems are not equipped to perform this form of structural reasoning.

5.2 Ensemble Methods for LLM-based Systems

Ensemble learning techniques [25, 26] have been widely used to improve robustness and generalization by aggregating diverse predictions. In the context of LLM-based systems, ensemble strategies have recently demonstrated strong performance gains. Studies in mathematical reasoning, such as Best-of-N [27], show that selecting among multiple LLM-generated solutions via a scoring model yields substantial improvements. In competitive programming, S* [28] integrates execution feedback from public and LLM-generated tests with clustering-based selection to identify high-quality code candidates. EnsLLM [29] combines syntactic similarity (CodeBLEU [30]) and behavioral similarity (CrossHair [31]) to vote among candidate patches.

Several works adapt ensemble-style selection to software issue resolution. Augment [32] adopts an LLM-as-a-judge strategy to evaluate semantic alignment between issue descriptions and candidate patches. DeiBase [33] prompts the LLM to generate explanations and confidence scores for each patch before ranking candidates. While effective in Python benchmarks, these ensemble techniques rely on the underlying agent’s ability to retrieve accurate code context—a capability that degrades severely in C++ environments.

5.3 Summary and Positioning

Existing approaches demonstrate substantial progress on Python issue resolution but do not address the structural complexity inherent in C++. As evidenced by the substantial performance decline observed when Python-optimized agents are applied to C++ repositories, the dominant research direction lacks mechanisms for resolving identifier ambiguity, disambiguating overloaded declarations, or navigating deep syntactic structures. Consequently, these systems are unable to perform reliable localization or context retrieval in large-scale C++ codebases.

In contrast, INFCode-C++ is designed specifically to address these deficiencies. By integrating semantic code-intent retrieval with deterministic AST-based querying, INFCode-C++ provides a C++-aware retrieval and localization pipeline that overcomes the limitations of Python-focused agents. This enables reliable context construction and significantly improves issue resolution in complex C++ repositories.

6 THREATS TO VALIDITY

Several factors may affect the validity of our findings. Internal validity may be influenced by the nondeterminism of LLM generation, though we mitigate this by using fixed seeds where possible

and executing all agents under identical compute budgets. Construct validity is limited by the evaluation protocol of MultiSWE-bench [3], where issue resolution rate does not capture partial progress, and a minority of issue reports include stack traces that may slightly simplify reproduction. External validity is constrained by the use of C++ projects from a single benchmark; results may differ on other domains such as embedded systems or mixed-language repositories. Nevertheless, the consistent improvements observed across all settings suggest that the benefits of INFCode-C++'s C++-aware retrieval and structured reasoning generalize beyond the evaluated projects.

7 CONCLUSION

We presented INFCode-C++, a C++-aware autonomous system for repository-level issue resolution. Unlike prior agents designed for Python, INFCode-C++ integrates semantic code-intent retrieval and AST-structured querying to address the identifier ambiguity and deep structural complexity of C++ codebases. Evaluated on MultiSWE-bench-CPP, INFCode-C++ achieves state-of-the-art performance, resolving 25.58% of issues and substantially outperforming leading baselines. Ablation and behavioral analyses highlight the importance of both semantic and structural retrieval, and reveal current bottlenecks in reproduction and localization. Overall, INFCode-C++ demonstrates that reliable C++ repair requires language-aware reasoning beyond existing agent designs and provides a foundation for future multi-language issue resolution systems.

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?" *arXiv preprint arXiv:2310.06770*, 2023.
- [2] P. Gao, Z. Tian, X. Meng, X. Wang, R. Hu, Y. Xiao, Y. Liu, Z. Zhang, J. Chen, C. Gao *et al.*, "Trae agent: An llm-based agent for software engineering with test-time scaling," *arXiv preprint arXiv:2507.23370*, 2025.
- [3] D. Zan, Z. Huang, W. Liu, H. Chen, L. Zhang, S. Xin, L. Chen, Q. Liu, X. Zhong, A. Li *et al.*, "Multi-swe-bench: A multilingual benchmark for issue resolving," *arXiv preprint arXiv:2504.02605*, 2025.
- [4] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh *et al.*, "Openhands: An open platform for ai software developers as generalist agents," *arXiv preprint arXiv:2407.16741*, 2024.
- [5] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Agentless: Demystifying llm-based software engineering agents," *arXiv preprint arXiv:2407.01489*, 2024.
- [6] 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.
- [7] X. Chen, Z. Tao, K. Zhang, C. Zhou, W. Gu, Y. He, M. Zhang, X. Cai, H. Zhao, and Z. Jin, "Revisit self-debugging with self-generated tests for code generation," *arXiv preprint arXiv:2501.12793*, 2025.
- [8] A. Sonwane, I. White, H. Lee, M. Pereira, L. Caccia, M. Kim, Z. Shi, C. Singh, A. Sordoni, M.-A. Côté *et al.*, "Bugpilot: Complex bug generation for efficient learning of swe skills," *arXiv preprint arXiv:2510.19898*, 2025.
- [9] OpenAI, "Gpt-5," Aug. 2025, accessed: 2025-11-13. [Online]. Available: <https://openai.com/index/introducing-gpt-5/>
- [10] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.
- [11] V. Aggarwal, O. Kamal, A. Japesh, Z. Jin, and B. Schölkopf, "Dars: Dynamic action re-sampling to enhance coding agent performance by adaptive tree traversal," *arXiv preprint arXiv:2503.14269*, 2025.
- [12] 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.
- [13] F. Mu, J. Wang, L. Shi, S. Wang, S. Li, and Q. Wang, "Experepair: Dual-memory enhanced llm-based repository-level program repair," *arXiv preprint arXiv:2506.10484*, 2025.
- [14] H. Li, Y. Shi, S. Lin, X. Gu, H. Lian, X. Wang, Y. Jia, T. Huang, and Q. Wang, "Swe-debate: Competitive multi-agent debate for software issue resolution," *arXiv preprint arXiv:2507.23348*, 2025.
- [15] S. Chen, S. Lin, X. Gu, Y. Shi, H. Lian, L. Yun, D. Chen, W. Sun, L. Cao, and Q. Wang, "Swe-exp: Experience-driven software issue resolution," *arXiv preprint arXiv:2507.23361*, 2025.
- [16] Y. Ma, Q. Yang, R. Cao, B. Li, F. Huang, and Y. Li, "Alibaba lingmaagent: Improving automated issue resolution via comprehensive repository exploration," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, 2025, pp. 238–249.

- [17] H. Ruan, Y. Zhang, and A. Roychoudhury, "Specrover: Code intent extraction via llms," *arXiv preprint arXiv:2408.02232*, 2024.
- [18] S. Ouyang, W. Yu, K. Ma, Z. Xiao, Z. Zhang, M. Jia, J. Han, H. Zhang, and D. Yu, "Repograph: Enhancing ai software engineering with repository-level code graph," *arXiv preprint arXiv:2410.14684*, 2024.
- [19] Y. Ma, Y. Li, Y. Dong, X. Jiang, R. Cao, J. Chen, F. Huang, and B. Li, "Thinking longer, not larger: Enhancing software engineering agents via scaling test-time compute," *arXiv preprint arXiv:2503.23803*, 2025.
- [20] Y. Ma, R. Cao, Y. Cao, Y. Zhang, J. Chen, Y. Liu, Y. Liu, B. Li, F. Huang, and Y. Li, "Lingma swe-gpt: An open development-process-centric language model for automated software improvement," *arXiv preprint arXiv:2411.00622*, 2024.
- [21] A. Antoniadou, A. Örwall, K. Zhang, Y. Xie, A. Goyal, and W. Wang, "Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement," *arXiv preprint arXiv:2410.20285*, 2024.
- [22] Y. Liu, P. Gao, X. Wang, J. Liu, Y. Shi, Z. Zhang, and C. Peng, "Marscode agent: Ai-native automated bug fixing," *arXiv preprint arXiv:2409.00899*, 2024.
- [23] OpenAI, "Introducing swe-bench verified," <https://openai.com/index/introducing-swe-bench-verified/>, 2024, accessed: YYYY-MM-DD.
- [24] Anthropic, "Claude sonnet 3.7," Model released by Anthropic; available at <https://www.anthropic.com/claude/sonnet>, Feb. 2025, accessed: 2025-11-13.
- [25] R. Polikar, "Ensemble learning," in *Ensemble machine learning*. Springer, 2012, pp. 1–34.
- [26] O. Sagi and L. Rokach, "Ensemble learning: A survey," *Wiley interdisciplinary reviews: data mining and knowledge discovery*, vol. 8, no. 4, p. e1249, 2018.
- [27] C. V. Snell, J. Lee, K. Xu, and A. Kumar, "Scaling llm test-time compute optimally can be more effective than scaling parameters for reasoning," in *The Thirteenth International Conference on Learning Representations*, 2025.
- [28] D. Li, S. Cao, C. Cao, X. Li, S. Tan, K. Keutzer, J. Xing, J. E. Gonzalez, and I. Stoica, "S*: Test time scaling for code generation," *arXiv preprint arXiv:2502.14382*, 2025.
- [29] T. Mahmud, B. Duan, C. Pasareanu, and G. Yang, "Enhancing llm code generation with ensembles: A similarity-based selection approach," *arXiv preprint arXiv:2503.15838*, 2025.
- [30] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.
- [31] P. Schanely, "CrossHair: An analysis tool for Python that blurs the line between testing and type systems," <https://github.com/pschanely/CrossHair>, 2025, accessed: 2025-xx-xx.
- [32] Augment Code, "Augment swe-bench verified agent," <https://github.com/augmentcode/augment-swebench-agent>, 2025.
- [33] K. Zhang, W. Yao, Z. Liu, Y. Feng, Z. Liu, R. Murthy, T. Lan, L. Li, R. Lou, J. Xu *et al.*, "Diversity empowers intelligence: Integrating expertise of software engineering agents," *arXiv preprint arXiv:2408.07060*, 2024.