IssueExec: A Test-Driven Approach for Localizing Software Engineering Issues

JIAWEI LIU, Shanghai Jiao Tong University, China and Shanghai Innovation Institute, China
YUN LIN*, Shanghai Jiao Tong University, China
CHENYAN LIU, National University of Singapore, Singapore
YU QIAN, Shanghai Jiao Tong University, China
YIMING LIU, Shanghai Jiao Tong University, China and Shanghai Innovation Institute, China
JIAXIN CHANG, Shanghai Jiao Tong University, China
WEINAN ZHANG, Shanghai Jiao Tong University, China and Shanghai Innovation Institute, China
LINPENG HUANG, Shanghai Jiao Tong University, China

Issue localization, which identifies code locations requiring modification from issue descriptions, is a critical step in automated software maintenance. Existing approaches predominantly attempt to directly align issue descriptions with code elements, yet often struggle due to the inherent abstraction gap between the issue description and code implementation. Seeking alternative signals, our theoretical analysis suggests that test suites can serve as executable proxies for requirements, reducing localization uncertainty by 2.01 bits on average. A large-scale empirical study on 18 repositories validates this premise: existing tests cover 96.98% of ground-truth files, and the two-hop pathway yields stronger semantic connectivity than direct matching in 82.4% of cases. Despite their potential, leveraging tests for localization faces two key challenges: the semantic gap separating issue descriptions from test identifiers, and the substantial noise in execution traces from infrastructure code. To address these, we propose IssueExec, which bridges the semantic gap through domain-knowledge-enhanced test representations and filters noise via hierarchical trace analysis. Experiments on SWE-bench Lite show that IssueExec achieves state-of-the-art performance, improving function-level Recall@1 by 41.57% over the strongest baseline. When integrated into the Agentless pipeline, IssueExec resolves 17.72% more issues, demonstrating practical downstream benefits.

CCS Concepts: • **Software and its engineering → Test-driven software engineering**; • **Computing methodologies → Information extraction**.

Additional Key Words and Phrases: Issue Localization, Test-driven Analysis

*Corresponding author.

Authors' Contact Information: Jiawei Liu, amberwabi2003@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China and Shanghai Innovation Institute, Shanghai, China; Yun Lin, lin_yun@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China; Chenyan Liu, chenyan@u.nus.edu, National University of Singapore, Singapore, Singapore; Yu Qian, qysaltyfish@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China; Yiming Liu, liu_yiming@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China and Shanghai Innovation Institute, Shanghai, China; Jiaxin Chang, cjx001234@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China; Weinan Zhang, wnzhang@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China and Shanghai Innovation Institute, Shanghai, China; Linpeng Huang, lphuang@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China.

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Jiawei Liu, Yun Lin, Chenyan Liu, Yu Qian, Yiming Liu, Jiaxin Chang, Weinan Zhang, and Linpeng Huang. 2018. IssueExec: A Test-Driven Approach for Localizing Software Engineering Issues. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 19 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Automating issue resolution is a longstanding goal in software maintenance, and recent LLM-based tools have renewed interest in practical automation [21, 23]. A central step is **issue localization**, which occupies approximately two-thirds of the total debugging time according to empirical studies [9]. Given an issue description and a repository (with its accompanying test suite), this task aims to identify the code locations that should be modified to implement the intended behavior.

Recent solutions typically follow a *direct issue–code alignment* paradigm [12, 32, 36]. In practice, they either (i) treat localization as a retrieval task and rank code elements by similarity to the issue description [10, 19], or (ii) use a structured search procedure, often powered by an LLM agent, to traverse the repository hierarchy, invoke tools, and iteratively narrow down candidate files and functions [31, 51, 55]. Despite their progress, these approaches still hinge on matching requirement-level language in issues to implementation-oriented identifiers in code, making them brittle when the issue describes behavior while the relevant code is organized by technical concerns [16, 18].

Concretely, issues express *behavioral expectations* while code identifiers reflect *technical organization*. As illustrated in Figure 1(a), the issue requests support for "ServerlessV2" (a functional capability), yet the target function `_get_db_cluster_kwargs` is named after its implementation role in parameter extraction, with no indication of the feature it serves. A natural alternative would be requirement-aware localization, but explicit requirement documentation drifts from implementation and rarely maps to code precisely [30]. Tests, however, must remain synchronized with implementation to pass [58]. As Figure 1(b) shows, the test `test_create_serverless_db_cluster` shares behavioral terminology with the issue (both reference "serverless" and "db_cluster"), providing a natural intermediate target for retrieval. Executing the retrieved tests further yields execution traces that connect the test intent to the exercised implementation, including the target function. This two-hop pathway (issue → tests → code) can provide a more reliable evidence chain than direct issue–code matching alone. This is not an isolated case: our theoretical analysis suggests that test-driven localization reduces uncertainty by 2.01 bits on average compared to direct retrieval (Section 2). To validate this premise, our empirical study on 18 high-quality open-source Python repositories (Section 3) shows that existing tests cover 96.98% of ground-truth files and 66.70% of functions, while the two-hop pathway yields stronger semantic connectivity than direct matching in 82.4% of cases.

This effectiveness stems from tests' unique position in software development: they are both human-readable specifications and machine-executable validators [2, 8, 24]. Each test verifies specific functionality, with identifiers that encode requirement-level semantics [34, 38], while execution traces establish deterministic links to implementation [18, 25, 49]. This duality motivates our key insight: **test suites can serve as executable requirements**, as test names and assertions provide a requirement-level semantic channel, while execution traces provide a concrete dynamic link from tests to the exercised code.

Leveraging test coverage for issue localization introduces two technical challenges:

- **C1. Domain-Specific Semantic Gap:** Retrieving relevant tests from issues requires domain knowledge absent from generic embeddings, e.g., "connecting wrong time displayed for users in different countries" to `test_tz_aware_datetime` demands understanding that "tz" denotes timezone [13, 35, 53];

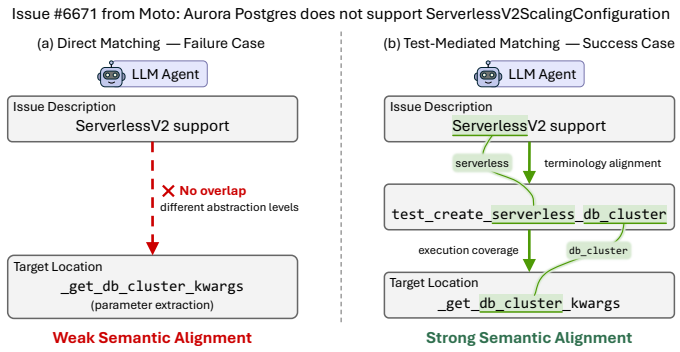

Fig. 1. Motivation for test-mediated localization. (a) A failure case of direct issue-to-code matching due to weak semantic alignment. (b) A success case using tests as functional bridges to align requirements with target locations.

- **C2. Infrastructure Noise and Trace Dilution:** Localizing target code from test coverage is non-trivial as a single test may execute thousands of functions including infrastructure irrelevant to the requirement, and only passing tests exist before resolution, precluding spectrum-based techniques [1, 28].

To address these challenges, we propose IssueExec (Issue Executable), a test-mediated localization framework that treats test suites as executable realizations of issue requirements. In particular, IssueExec tackles **C1** by retrieving candidate tests that reflect the issue intent using test representations enhanced with project-specific domain knowledge (e.g., abbreviations and API aliases) mined from historical commits. It then tackles **C2** by leveraging the runtime execution traces of the retrieved tests and modeling their execution hierarchy as a trace graph, which helps filter incidental infrastructure and highlight requirement-central code locations. IssueExec combines (1) domain knowledge enhancement via historical commit mining for robust issue–test alignment; and (2) dynamic trace graph modeling for hierarchy-aware trace denoising and localization.

Together, these components construct a focused, requirement-centric search space that maintains high recall while substantially reducing context size. On SWE-bench Lite [27], IssueExec boosts the Recall@1 localization performance at the file, module, and function levels by 17.78%, 25.98%, and 41.57%, respectively. When integrated into the Agentless pipeline [51], our approach resolves 17.72% more issues, indicating benefits for downstream patch generation.

Our contributions are as follows:

- We formalize and operationalize the notion of **tests as executable requirements** for issue localization, showing how semantic alignment at the test level and execution grounding at runtime jointly bridge the issue–code gap.
- We propose IssueExec, a procedure-based framework that enhances test representations with domain knowledge for effective issue-test alignment and leverages execution trace hierarchy to pinpoint suspicious locations.
- We demonstrate state-of-the-art localization on SWE-bench Lite, with 17.72% improvement in end-to-end resolution when integrated into Agentless.

Due to space constraints, we will make the source code, prompts, and additional materials available on an anonymous artifact page [3].

2 Theoretical Motivation

We conduct a preliminary theoretical analysis and empirical validation to establish whether the test-driven localization paradigm yields meaningful uncertainty reduction over direct issue-to-code matching. We model how uncertainty evolves along the hierarchical retrieval process (issue to tests to trace-constrained code) and validate the predicted gain under method-agnostic conditions [20]. Our goal is to quantify the uncertainty reduction achievable by the test-driven paradigm itself, independent of specific implementation choices, thereby providing principled justification for the subsequent framework design.

2.1 Formal Preliminaries

Setup and Notation. Given an issue description d and repository R, let L be the universe of candidate locations (e.g., all functions/methods), with $N = |L|$. Let $\mathcal{G} \subseteq L$ denote the ground-truth edit set for d, with $|\mathcal{G}| > 0$. For any issue localization system that returns a ranked list of locations under a predefined budget k_{ret}; the resulting set is denoted as L_{ret}, where $|L_{\text{ret}}| = k_{\text{ret}}$.

For the test-driven pipeline, we retrieve a fixed-size set of tests $\hat{\mathcal{T}}_d$, based on the given issue description d, with a predefined budget k_{test}, i.e., $|\hat{\mathcal{T}}_d| = k_{\text{test}}$. Each retrieved test $t \in \hat{\mathcal{T}}_d$ has a dynamic coverage set $\text{Cov}(t) \subseteq L$. The trace-constrained candidate subspace is the union coverage:

$$L_{\text{cov}} = \bigcup_{t \in \hat{\mathcal{T}}_d} \text{Cov}(t), \qquad N_{\text{cov}} = |L_{\text{cov}}|. \tag{1}$$

Entropy Proxy. We approximate uncertainty with Hartley entropy $H(C) \approx \log_2 |C|$ [14, 22], which captures the combinatorial ambiguity of selecting the correct edit locations from a candidate set. Our objective is not to estimate the intractable distribution $P(L \mid d)$, but to quantify the *relative* uncertainty reduction induced by hierarchical retrieval and trace constraints.

2.2 Quantifying Localization Uncertainty

We now formulate the localization uncertainty for two different paradigms to quantify the theoretical advantage of test-driven localization. Specifically, we define H_{direct} for the baseline approach that localizes code locations by directly matching the issue description to code, and H_{indirect} for our proposed two-stage hierarchical process that leverages tests as executable requirements to perform indirect localization via an issue-to-test and test-to-code mapping.

Direct Retrieval Uncertainty. A direct method ranks the entire space L and returns L_{ret}. Let $g_{\text{ret}} = |\mathcal{G} \cap L_{\text{ret}}|$ be the number of ground-truth locations already included in the returned set. We define a partition-based entropy proxy that accounts for whether ground-truth locations fall inside or outside the returned set:

$$H_{\text{direct}} = \mathbb{1}[g_{\text{ret}} > 0] \cdot \left(-\log_2 \frac{g_{\text{ret}}}{|L_{\text{ret}}|} \right) + \mathbb{1}[|\mathcal{G}| - g_{\text{ret}} > 0] \cdot \left(-\log_2 \frac{|\mathcal{G}| - g_{\text{ret}}}{N - |L_{\text{ret}}|} \right) \tag{2}$$

Intuitively, the first term measures the ambiguity among returned candidates that contain (some of) the true edits, while the second term captures the residual uncertainty when some ground-truth edits are not retrieved.

Test-Driven Retrieval Uncertainty. Test-driven localization decomposes the search into two stages: (i) retrieving requirement-relevant tests, and (ii) localizing within a trace-constrained subspace. We model uncertainty as the sum of stage-wise entropies, accounting for realistic failure modes.

Stage 1: Test selection entropy. Among the retrieved tests $\hat{\mathcal{T}}_d$, we call a test *effective* if it covers at least one ground-truth location, i.e., $\text{Cov}(t) \cap \mathcal{G} \neq \emptyset$. Let n be the number of effective tests in $\hat{\mathcal{T}}_d$.

We define:

$$H_{\text{stage1}} = \mathbb{1}[n > 0] \cdot \Big(- \log_2 \frac{n}{k_{\text{test}}} \Big). \tag{3}$$

This term captures how confidently the pipeline selects tests that are functionally linked to the true edits. When $n = 0$, the pipeline fails to retrieve any effective test, and the second stage will necessarily incur a large penalty (defined below).

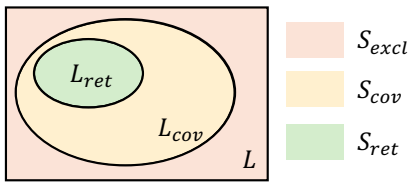

Fig. 2. Venn diagram illustrating the hierarchical search space partition

Stage 2: Trace-constrained localization entropy. We partition the global space L into three disjoint regions induced by the trace subspace and the returned top-k_{ret} locations. Critically, in the test-driven pipeline, the final retrieved set L_{ret} is a subset of the trace-covered space, i.e., $L_{\text{ret}} \subseteq L_{\text{cov}}$. We define the partitions as:

$$S_{\text{ret}} = L_{\text{ret}}, \qquad S_{\text{cov}} = L_{\text{cov}} \setminus L_{\text{ret}}, \qquad S_{\text{excl}} = L \setminus L_{\text{cov}}. \tag{4}$$

Here, S_{excl} represents the excluded regions that are not covered by any retrieved test traces, as shown in Figure 2. Let $N_i = |S_i|$ and $g_i = |\mathcal{G} \cap S_i|$ for $i \in \{\text{ret}, \text{cov}, \text{excl}\}$. We then define:

$$H_{\text{stage2}} = \sum_{i \in \{\text{ret,cov,excl}\},\, g_i > 0} \Big(- \log_2 \frac{g_i}{N_i} \Big). \tag{5}$$

This formulation makes two realistic phenomena explicit: (i) *Space compression* [44]: when L_{cov} is much smaller than L, uncertainty shrinks because $N_{\text{cov}} \ll N$; (ii) *Coverage/retrieval penalty:* if $g_{\text{excl}} > 0$ (i.e., some true edits fall outside trace coverage), the term $-\log_2(g_{\text{excl}}/N_{\text{excl}})$ can dominate, reflecting the high uncertainty caused by incomplete trace constraints.

Finally, the hierarchical uncertainty is:

$$H_{\text{indirect}} = H_{\text{stage1}} + H_{\text{stage2}}. \tag{6}$$

2.3 Entropy Gain Analysis

For each issue, we define the empirical uncertainty reduction:

$$\Delta H = H_{\text{direct}} - H_{\text{indirect}}. \tag{7}$$

A positive ΔH indicates that introducing tests and traces reduces the ambiguity of edit localization compared to direct matching, *under the same budget* of returning top-k_{ret} locations. The decomposition further reveals two complementary directions for maximizing entropy gain: (1) improving test retrieval precision to increase n/k_{test} and reduce ext-failure ($g_{\text{excl}} > 0$), thereby lowering H_{stage1}; (2) refining trace analysis to shrink N_{cov} while preserving g_{ret}, thereby lowering H_{stage2}. These insights directly inform our framework design in subsequent sections.

2.4 Theoretical Validation

We compute the above entropy proxies on 208 real-world resolved issues. To ensure a fair comparison that isolates the effect of the paradigm itself, we employ a unified embedding model for both retrieval pathways: (i) *direct retrieval*, which ranks code locations in $\mathcal{L}$ by similarity to the issue description,

and (ii) *indirect retrieval*, which first retrieves tests by the same similarity measure, then constrains the candidate space via execution coverage. Test coverage is obtained from standard execution traces without task-specific filtering.

Table 1. Pipeline-aware entropy analysis under realistic test retrieval. H_{direct} and H_{indirect} are defined in Equation 2 and Equation 6. ΔH is the uncertainty reduction (Equation 7). "Ext. failure" reports the fraction of issues with $g_{\text{excl}} > 0$ (some ground-truth edits outside trace coverage). Results use $k_{\text{test}} = 5$ and $k_{\text{ret}} = 10$.

Metric	**Value**
Total Valid Samples	208
Mean H_{direct} (bits)	12.85
Mean H_{indirect} (bits)	10.84
Mean H_{stage1} (bits)	0.22
Mean H_{stage2} (bits)	10.63
Mean Entropy Gain ΔH (bits)	**2.01**
Ext. failure rate ($g_{\text{excl}} > 0$)	24.31%

The validation results are shown in Table 1. A consistently positive ΔH confirms that the two-hop pathway (issue → tests → trace-constrained code) reduces localization uncertainty on average. Notably, even with a high ext-failure rate (24.31%), the overall ΔH remains strongly positive. For issues without external failure, trace constraints yield substantial gains ($\Delta H = 3.91$ bits) by compressing the search space. For issues with external failure, direct retrieval also typically fails to rank the ground-truth locations highly ($H_{\text{direct}} = 13.20$ bits), while indirect retrieval still achieves partial hits within the trace-constrained subspace, resulting in a positive ΔH of 1.08 bits. These findings validate the theoretical advantage of test-driven localization and motivate the design of IssueExec, which realizes this paradigm through enhanced test retrieval and hierarchical trace analysis.

3 Empirical Study

Theoretical arguments in Section 2 suggest that tests can serve as *executable requirements* for issue localization by providing (i) requirement-level semantics and (ii) a dynamic link to exercised code. In this section, we validate this premise empirically on a large-scale dataset, and quantify when and how tests provide actionable signals for localization. We test the following hypotheses:

- **H1 (Coverage feasibility).** Existing tests execute the ground-truth edit locations.
- **H2 (Retrievability).** Tests that cover ground-truth locations are more semantically aligned with the issue than non-covering tests.
- **H3 (Bridging effect).** The two-hop pathway Issue→Test→Location provides stronger semantic connectivity than direct Issue→Location matching.

3.1 Setup

The empirical investigation is conducted on a collection of 929 issues sourced from SWE-bench [27] and SWE-bench Gym [37], covering 18 widely-adopted Python repositories (each with ≥500 stars). These projects represent diverse domains including web frameworks, data processing, and scientific computing. To ensure the reliability of dynamic analysis, we employ a Docker-based execution framework. Each repository-issue pair is isolated within a container where the environment is reset to its pre-patch state. We retain only those instances where the test environment is sufficiently robust, defined as having at least 50% of test functions execute successfully, to prevent environment configuration errors from confounding the findings.

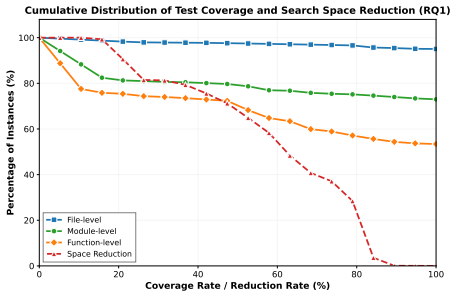

Fig. 3. Cumulative distribution of test coverage rates (H1). Existing tests cover 96.98% of ground-truth files and 66.70% of ground-truth functions on average, reducing search space to 58.39% of the repository.

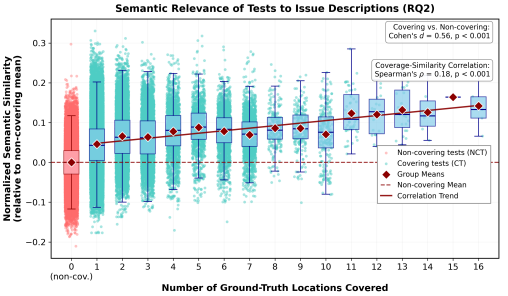

Fig. 4. Semantic similarity of tests to issue descriptions (H2). Tests covering more ground-truth locations exhibit higher similarity to the issue, enabling retrieval-based selection.

Function-level coverage traces are collected via instrumentation using the Python `sys.settrace` mechanism. During the execution of the full test suite, the tracer records call and return events for every function invocation. We apply a filtering layer to exclude standard library calls and third-party dependencies, retaining only repository-internal functions identified by their fully-qualified paths. Ground-truth edit locations are extracted by parsing the pull request diffs associated with each issue, identifying the specific functions modified by developers to resolve the reported problem.

For semantic analysis, the bge-large-en-v1.5 [52] model is utilized to generate dense vector embeddings. Each test case is transformed into a textual representation by concatenating its docstring, its fully-qualified module path, and its complete function body. These components are truncated to 512 tokens to fit the model context window. Code locations are similarly represented by their fully-qualified signatures. Semantic similarity is then quantified using the cosine similarity between the resulting embeddings.

3.2 Findings

H1. (Coverage feasibility). Figure 3 demonstrates the structural feasibility of using existing tests to reach target locations. By intersecting the dynamic traces of the entire test suite with the ground-truth edit sets, we observe that existing tests provide broad coverage, reaching 96.98% of files and 66.70% of functions requiring modification. This confirms that for the vast majority of issues, the necessary execution signals already exist within the repository infrastructure. Furthermore, the search space reduction is quantified by comparing the cardinality of the union of all test-covered functions against the total functions in the repository. This restriction narrows the candidate pool to 58.39% of the repository on average, achieving high recall while providing effective search space compression for subsequent localization stages.

H2. (Retrievability). For test-based localization to succeed, covering tests must be distinguishable from non-covering tests via semantic signals. Our experimental setup compares the similarity of tests that cover ground-truth locations against a baseline of randomly sampled tests from the same repository. After verifying the normality of the similarity distribution via the Shapiro-Wilk test, a paired t-test is conducted [43]. As shown in Figure 4, covering tests exhibit significantly higher similarity to issue descriptions in 90.9% of instances ($p < 0.001$; Cohen's $d = 0.56$), indicating a medium-to-large statistical effect. Additionally, Spearman's rank correlation analysis between the number of ground-truth locations covered and the issue similarity yields $\rho = 0.18$ ($p < 0.001$). This suggests that semantic relevance can effectively prioritize tests with broader functional coverage of the requirement.

H3. (Bridging effect). Our central hypothesis is that tests serve as semantic bridges between abstract issues and technical code. For each (issue, location) pair where coverage exists, we compare the direct similarity $s_{\text{direct}} = \text{sim}(\text{issue}, \text{location})$ against a test-mediated pathway strength. We formalize this mediated connectivity using a two-hop geometric mean formulation [20]:

$$s_{\text{mediated}} = \max_{t \in \mathcal{T}_{\text{cover}}} \sqrt{\text{sim}(\text{issue}, t) \cdot \text{sim}(t, \text{location})}$$

This formulation ensures that a strong bridge requires balanced semantic associations across both the requirement-to-test and test-to-implementation links. As illustrated in the results, the mediated pathway is stronger in 82.4% of cases (paired t-test, $p < 0.001$; Cohen's $d = 0.93$). The substantial effect size confirms that tests act as effective semantic relays, leveraging both requirement-level language and deterministic execution grounding to achieve connectivity that direct issue-to-code matching cannot.

3.3 Implications for IssueExec

These findings directly inform IssueExec's design. H1 establishes that test coverage provides a high-recall, reduced-entropy search space. H2 confirms that semantic retrieval can identify requirement-relevant tests, though the moderate effect size ($d = 0.56$) motivates our domain-knowledge enhancement (Section 4.2) to strengthen Issue→Test alignment. H3 validates the two-hop localization strategy, while the gap between covering and non-covering code within traces motivates our hierarchical trace analysis (Section 4.4) to filter infrastructure noise.

4 The IssueExec Framework

The previous two sections provide complementary support for our central premise that *tests can serve as executable requirements* for issue localization: Section 2 motivates the premise from a theoretical perspective, and Section 3 validates it empirically at scale. Building on this premise, we introduce IssueExec, which operationalizes tests as an intermediate layer that connects issue descriptions to candidate code locations.

4.1 Problem Formulation

Given a natural-language issue description d and a repository R with code entities V, issue localization aims to identify the code locations $L = \{l_1, l_2, \ldots, l_k\} \subseteq V$ that need to be modified to resolve the issue [35, 50]. Formally, we seek the optimal L^* that minimizes $|L|$ such that $\text{Patch}(R, L) \models d$.

Guided by the preliminary study in Section 2, which confirms the uncertainty reduction achievable via test-driven localization, we instantiate a two-stage formulation. First, we retrieve tests $\mathcal{T}_d$ semantically aligned with d to maximize the effective test ratio n/k_{test}. Second, we leverage the execution coverage $\text{Cov}(\mathcal{T}_d)$ to localize L^* within a trace-constrained, low-entropy search space.

Framework Overview. Figure 5 illustrates the framework. IssueExec first performs offline preprocessing to collect test execution traces and mine commit history for enriching test representations with domain knowledge (Section 4.2). At inference time, given an issue description, the localization pipeline proceeds as following stages: ❶ retrieve relevant tests $\mathcal{T}_d$ through two-phase filtering (Section 4.3), implementing the first stage of our formulation; ❷ analyze execution traces to identify suspicious locations within $\text{Cov}(\mathcal{T}_d)$ (Section 4.4); ❸ refine and rerank candidates to produce the final L^* (Section 4.5).

4.2 Domain Knowledge Enhancement

The semantic distance between the generic embeddings of the issue description and the tests remains substantial, as default test representations (e.g., function signatures and docstrings) fail to

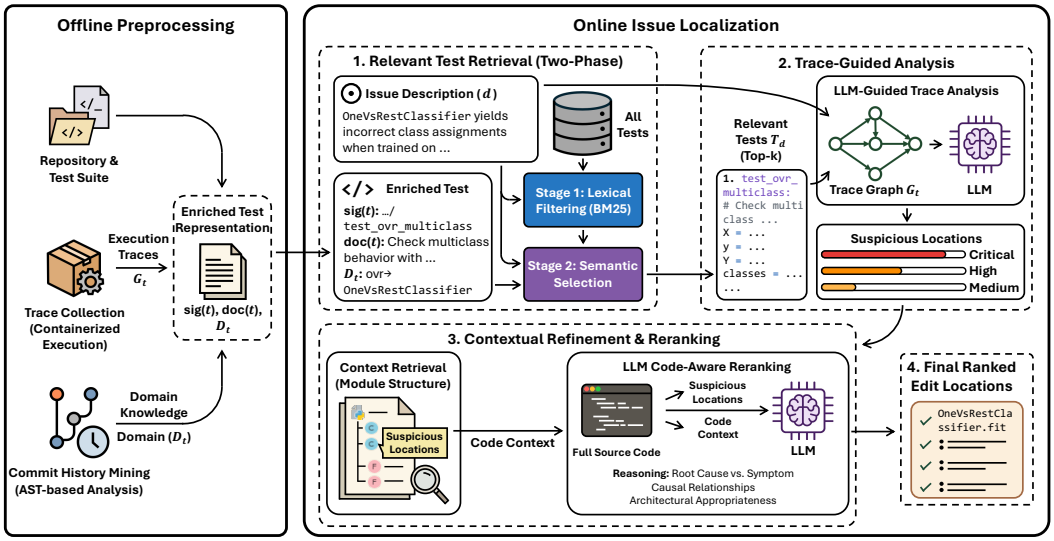

Fig. 5. Overview of the IssueExec framework. **Left:** Offline preprocessing collects test execution traces G_t via containerized execution and extracts domain knowledge D_t from commit history to construct enriched test representations. **Right:** Online issue localization proceeds through four steps: (1) relevant test retrieval combining BM25 lexical filtering and LLM-based semantic selection to obtain $\mathcal{T}_d$, (2) trace-guided analysis leveraging execution traces and LLM reasoning to identify suspicious locations with confidence levels, (3) contextual refinement and reranking using module structure and full source code, and (4) final ranked edit locations L^* for downstream patch generation.

capture domain-specific associations and project-specific terminology. By incorporating domain knowledge D_t, we bridge the terminology gap between issue descriptions and test identifiers, effectively reducing the semantic gap for requirement-relevant tests.

To this end, we extract domain knowledge from the repository's commit history, which captures the semantic context under which tests were introduced and evolved. For each test function t, we extract two types of signals.

First, we identify the commit that first introduced t via AST-based diff analysis, filtering out low-quality commits (e.g., merge commits, bulk refactoring). The associated commit message provides requirement-level semantic context:

$$m_t = \text{msg}(\arg\min_{h \in \mathcal{H}} \{\text{time}(h) \mid t \in \text{added}(h)\}) \tag{8}$$

where $\mathcal{H}$ denotes the commit history and $\text{added}(h)$ returns test functions introduced in commit h.

Second, we identify co-changed entities, i.e., code locations frequently modified together with t across commits:

$$\mathcal{A}_t = \{l \in V \mid \text{cochange}(t, l) \geq \tau\} \tag{9}$$

where V is the set of all code entities, $\text{cochange}(t, l)$ counts the number of commits in which both t and l were modified, and τ is a frequency threshold. These entities typically represent requirement-related code that t implicitly covers.

Finally, we distill domain-specific tokens from these extracted signals to form the test-specific knowledge set D_t:

$$D_t = \text{tok}(m_t) \cup \text{ident}(\mathcal{A}_t) \setminus \text{tok}(\text{sig}(t)) \tag{10}$$

where tok$(\cdot)$ extracts semantic tokens from the commit message and ident$(\mathcal{A}_t)$ extracts function and class names from co-changed entities. The set difference removes tokens already present in the test signature to avoid redundancy.

We construct the enriched representation by concatenating three components for each test:

$$\text{repr}(t) = [\text{sig}(t); \text{doc}(t); D_t] \tag{11}$$

where sig(t) denotes the fully qualified test identifier and doc(t) is the docstring if present. By incorporating D_t, domain-specific tokens provide additional semantic alignment between issues and tests beyond what generic embeddings can capture from sig(t) alone, thereby more effectively bridging the terminology gap and reducing the semantic gap for requirement-relevant tests.

4.3 Relevant Test Retrieval

This section implements the first stage of our localization pipeline. Given an issue description d and the enriched test representations constructed in Section 4.2, IssueExec employs a two-phase filtering process that retrieves a subset $\mathcal{T}_d \subseteq \mathcal{T}$ with $|\mathcal{T}_d| \ll |\mathcal{T}|$ while maximizing information extraction for downstream localization from the issue.

In the lexical filtering phase, we apply BM25 [40] to efficiently reduce the search space from potentially thousands of tests to a manageable candidate set:

$$\mathcal{T}_{\text{cand}} = \text{Filter}(d, \mathcal{T}, N) \tag{12}$$

where Filter returns the top-N tests ranked by lexical similarity to the issue d. This phase prioritizes recall, ensuring that requirement-relevant tests are retained.

In the semantic selection phase, we employ an LLM to carefully identify the most relevant subset from the candidates. Given the enriched representations repr(t) that incorporate domain-specific tokens D_t, the LLM explicitly reasons about the issue's intent and the tests' purposes:

$$\mathcal{T}_d = \text{Select}(d, \{\text{repr}(t) \mid t \in \mathcal{T}_{\text{cand}}\}, k) \tag{13}$$

where $k \ll N$ controls the size of the final retrieved set. The LLM is prompted to analyze which components are likely responsible for the reported issue, which test modules exercise those components, and which specific tests are most likely to cover the root cause. This produces a small set of relevant tests $\mathcal{T}_d$ whose execution traces will guide subsequent localization. The detailed prompt used for test selection is provided in our anonymous artifact page [3].

This two-phase design strikes a balance between efficiency and quality: BM25 filtering reduces the search space from $|\mathcal{T}|$ to N candidates efficiently, while LLM selection further refines to $k \ll N$ high-quality tests, avoiding the prohibitive cost of applying LLM reasoning to all tests.

4.4 Trace-Guided Localization

The retrieved tests $\mathcal{T}_d$ narrow the search space from the entire test suite to a small, requirement-relevant subset. However, test retrieval alone is insufficient for precise localization: the execution traces of these tests typically cover a large portion of the codebase, where the true edit locations constitute only a small fraction ($|L^*| \ll |\text{Cov}(\mathcal{T}_d)|$). While a single test may execute hundreds of functions, its execution trace preserves caller-callee relationships that encode causal structure. We leverage this hierarchical information to distinguish requirement-central code from incidental infrastructure and utility functions.

For each retrieved test $t \in \mathcal{T}_d$, we extract its dynamic execution trace and construct a directed graph $G_t = (V_t, E_t)$ rooted at t. The nodes are functions covered by the test:

$$V_t = \text{Cov}(t) \tag{14}$$

The edge set captures observed caller-callee relationships:

$$E_t = \{(u, v) \mid u \xrightarrow{t} v\} \tag{15}$$

where $u \xrightarrow{t} v$ denotes that u directly calls v during execution of t. Since G_t may exceed LLM context limits, we apply BFS-based pruning to retain only shallow layers of the call hierarchy. We then analyze these execution traces in conjunction with the issue description d and the enriched test representations $\text{repr}(t)$:

$$S_t = \text{Analyze}(d, \text{repr}(t), G_t) \tag{16}$$

where $S_t \subseteq V_t$ denotes the set of suspicious locations identified from trace G_t. The LLM generates diagnostic reports that analyze potential assertion deviations, hypothesize root causes, trace error propagation paths through the call chain, and identify systematic risks across related functions. Based on these analyses, each location $l \in S_t$ is assigned a confidence level $\text{conf}(l) \in \{\text{critical}, \text{high}, \text{medium}\}$. The aggregated suspicious set is:

$$S = \bigcup_{t \in \mathcal{T}_d} S_t \tag{17}$$

By analyzing execution traces with diagnostic reasoning, we aim to acquire the suspicious set S, which is designed to be sparse ($|S| \ll |\text{Cov}(\mathcal{T}_d)|$) while maintaining high recall of ground-truth locations, filtering out infrastructure code that contributes little information about the issue's root cause.

4.5 Refinement and Reranking

Building on the trace-guided suspicious set S from Section 4.4, we further refine and rerank candidates by enriching each location with additional contextual information.

The coverage-based candidate set may miss relevant locations that are structurally related but not directly covered. To address this, for each file f containing at least one suspicious location, we retrieve its structure as context. For modules exceeding context limits, we provide a compressed skeleton that preserves function signatures and class hierarchies while omitting implementation details. Here S' denotes the expanded candidate set after contextual refinement.

$$S' = S \cup \text{Refine}(\{f \mid f \cap S \neq \emptyset\}, d) \tag{18}$$

To prioritize candidates and filter false positives, we retrieve the full source code for each candidate location and perform code-aware reranking, reasoning about root-cause distinctions, inter-candidate causalities, and the architectural appropriateness of each contextual refinement:

$$L^* = \text{Rerank}(S', \{\text{code}(l) \mid l \in S'\}, d) \tag{19}$$

where $\text{code}(l)$ denotes the full source code of location l. The output L^* is the final ranked list of edit locations for downstream patch generation.

This refinement and reranking process ensures that L^* includes not only directly covered locations but also structurally and semantically related entities, improving recall beyond execution coverage. The reranking stage then prioritizes the most plausible edit locations, improving top-k precision.

5 Experiments

We evaluate IssueExec on SWE-bench Lite to answer the following research questions:

- **RQ1 (Localization performance, Section 5.2)** How effective is IssueExec for issue localization?
- **RQ2 (Issue resolution performance, Section 5.3)** Can improved localization benefit downstream issue resolution?

- **RQ3 (Cost Analysis, Section 5.4)** What is the cost-efficiency of IssueExec compared to existing baselines?
- **RQ4 (Ablation study, Section 5.5)** How does each component contribute to the overall performance of IssueExec?

5.1 Experimental Setup

Benchmarks. We evaluate on a widely-adopted benchmark for automated issue resolution. SWE-bench Lite [27] contains 300 issues sampled from 11 popular Python repositories, filtered for self-contained problems solvable without extensive codebase knowledge.
Baselines. We compare against three categories of methods. *Retrieval-based* approaches include BM25 [40], mGTE [60], CodeSage [59], and CodeRankEmbed [45]. *Procedure-based* methods include Agentless [51] and PatchPilot [31]. *Agent-based* approaches include LocAgent [12], SWE-Agent [55], OrcaLoca [57], OpenHands [48], and MoatlessTools [4].
Metrics. We evaluate localization at three granularities: file, module (class or top-level function), and function. We report Precision (Prec.) and Recall (Rec.) at cutoffs @1, @3 and @5, measuring the ability to identify ground-truth edit locations within the top-ranked predictions. We use resolved rate as the metric for the downstream repair task, and report F1-score for the ablation study.
Implementation Details. We use `GPT4o-2024-05-13` and `Claude3.5-sonnet-20241022` as the base models for all methods to ensure fair comparison. For IssueExec, we set the co-change frequency threshold τ to 3, the candidate test set size $|\mathcal{T}_{cand}|$ to 200, and the final selected test set size $|\mathcal{T}_d|$ to 5.

5.2 RQ1. Localization performance

Table 2 summarizes localization performance on SWE-bench Lite at file, module, and function granularities, reporting Precision and Recall at @1/@3/@5.

IssueExec achieves state-of-the-art performance across all granularities and metrics. Specifically, IssueExec with GPT-4o attains 70.07% file-level Recall@1, outperforming the best baseline Agentless by 17.78%. The improvements are more pronounced at finer granularities: IssueExec achieves 60.58% / 46.72% Precision@1 and 58.39% / 41.07% Recall@1 at module and function levels respectively, representing a notable boost of 25.98% and 41.57% in Recall@1 over the strongest baselines (Agentless at module level and MoatlessTools at function level); it further improves function-level Recall@3 by 23.38% over the strongest baseline. Similar gains hold with Claude, indicating that IssueExec is robust across backbone models, as test-mediated localization (issue→tests→code) reduces reliance on direct issue–code matching.

5.3 RQ2. Issue resolution performance

To evaluate whether improved localization translates to better end-to-end performance, we integrate IssueExec into the Agentless pipeline [51], replacing its original localization module while keeping the rest unchanged.

Table 3 presents the results on SWE-bench Lite. IssueExec-augmented Agentless achieves a resolution rate of 37.67%, compared to 32.00% for the original Agentless, representing 17 additional resolved issues. This improvement demonstrates that superior @1 localization performance, particularly the advantages of fine-grained localization at the function levels, directly benefits downstream patch generation by providing the repair model with a highly focused and relevant context. By pinpointing exact edit locations rather than providing entire files, IssueExec effectively avoids overwhelming the model with extraneous code, thereby reducing potential distractions during the reasoning process; detailed examples and cases illustrating this effect will be provided in our anonymous artifact page [3].

Table 2. RQ1. Performance comparison on SWE-bench Lite. Precision and Recall at @1, @3, and @5 are reported in %. **Bold**: best; Underline: second best.

Type	Method	File (%)				Module (%)						Function (%)					
		Precision		Recall		Precision			Recall			Precision			Recall		
		@1	@3	@1	@3	@1	@3	@5	@1	@3	@5	@1	@3	@5	@1	@3	@5
Retrieval	BM25	27.37	14.72	27.37	43.43	19.71	12.04	10.35	19.34	29.87	32.79	13.87	7.54	5.40	13.26	21.41	25.36
	mGTE	35.04	22.44	35.04	63.50	29.93	19.22	16.70	29.38	47.26	50.36	18.25	12.16	8.91	17.15	34.25	41.42
	CodeSage	32.85	19.34	32.85	51.09	26.28	15.88	14.43	25.91	35.95	38.87	14.23	7.91	6.20	12.77	21.44	27.86
	CodeRankEmbed	28.83	18.31	28.83	52.19	24.82	17.52	15.85	24.27	37.41	40.33	16.42	8.88	6.57	15.82	25.30	30.47
GPT-4o-2024-05-13																	
Procedure	Agentless	59.49	38.38	59.49	**78.27**	47.75	25.18	21.30	46.35	62.59	66.42	23.36	19.04	16.53	21.78	43.25	47.26
	PatchPilot	53.70	39.57	53.70	64.44	47.78	34.38	33.16	45.37	55.86	57.35	30.74	22.78	21.54	26.98	34.51	35.25
Agent	LocAgent	57.30	33.21	57.30	77.74	18.61	16.91	13.14	18.43	48.36	57.30	13.87	12.47	10.15	12.90	33.64	43.43
	SWE-Agent	54.74	54.01	54.74	57.66	41.24	40.82	40.77	40.15	43.61	43.61	30.29	**29.50**	**29.45**	28.25	32.00	32.00
	OrcaLoca	55.11	53.53	55.11	60.58	44.16	41.91	41.72	43.43	51.45	51.45	10.95	22.16	21.85	10.58	42.03	44.28
	OpenHands	31.39	30.41	31.39	31.75	28.83	27.86	26.41	28.28	28.83	29.75	24.09	24.21	23.13	21.90	24.15	24.64
	MoatlessTools	53.65	38.99	53.65	54.74	43.43	40.82	40.62	42.52	44.71	44.71	30.66	27.74	27.41	29.01	30.60	31.25
Procedure	IssueExec	**70.07**	**59.43**	**70.07**	78.10	**60.58**	**43.98**	**43.55**	**58.39**	**68.86**	**70.32**	**46.72**	26.95	25.94	**41.07**	**53.36**	**55.67**
Claude-3-5-Sonnet-20241022																	
Procedure	Agentless	57.71	42.57	57.71	69.70	57.66	26.88	21.83	55.47	67.67	**69.13**	24.45	17.09	13.87	21.09	40.03	43.65
	PatchPilot	55.84	37.29	55.84	67.15	50.73	26.15	22.79	48.97	56.69	59.18	24.09	15.27	13.75	20.67	28.45	**53.66**
Agent	LocAgent	53.65	27.80	53.65	68.25	34.67	15.39	11.49	34.12	39.05	40.51	10.95	5.96	5.02	10.40	11.86	12.23
	SWE-Agent	47.45	47.45	47.45	70.07	36.86	38.75	37.90	35.58	58.12	61.19	30.66	29.99	25.91	28.22	44.59	47.26
	OrcaLoca	65.69	40.08	65.69	70.80	45.62	27.74	27.12	44.83	48.60	48.60	43.07	32.97	32.41	40.45	51.89	52.25
	MoatlessTools	51.82	46.78	51.82	71.90	42.34	39.96	40.29	41.24	61.68	64.96	33.58	32.36	31.70	30.93	51.19	52.65
Procedure	IssueExec	**69.71**	**59.85**	**69.71**	**76.64**	**60.22**	**47.75**	**47.29**	**57.30**	**68.19**	68.73	**48.18**	**34.37**	**33.49**	**41.74**	**52.93**	**53.66**

Table 3. RQ2. End-to-end issue resolution on SWE-bench Lite.

Method	Resolved	Rate (%)
Agentless (original)	96	32.00
Agentless + IssueExec	113 (+17 ↑)	37.67 (+5.67 ↑)

5.4 RQ3. Cost Analysis

We evaluate the cost-efficiency of IssueExec by measuring the average monetary cost per issue on SWE-bench Lite using GPT-4o, with results compared against representative baselines in Table 4. Compared with representative agent-based methods, IssueExec achieves significant cost savings by reducing the average expense per issue by approximately 35.47%. Although IssueExec incurs a slightly higher cost than the procedural baseline Agentless due to the overhead of dynamic trace analysis, this modest investment is justified by its superior performance. IssueExec significantly outperforms Agentless in localization accuracy (Section 5.2) and enables the resolution of 17.72% more issues in downstream tasks (Section 5.3). In summary, IssueExec strikes an optimal balance between resource consumption and diagnostic effectiveness, providing state-of-the-art capabilities with much higher cost-efficiency than complex agents.

5.5 RQ4. Ablation study

We conduct ablation experiments to understand the contribution of each component, reporting F_1@3 scores across three granularities in Table 5. Removing trace analysis results in a performance

Table 4. RQ3. Efficiency and cost comparison (per issue, GPT-4o). **Bold**: best; Underline: second best.

Method	Cost ($)
Agentless	**0.70**
OpenHands	1.14
SWE-Agent	1.62
OrcaLoca	1.77
IssueExec	0.94

Table 5. RQ4. Ablation study on SWE-bench Lite.

Configuration	F1@3 (%)		
	File	Module	Function
IssueExec (full)	**65.27**	**51.07**	**34.08**
w/o Domain Knowledge	58.47	43.47	26.38
w/o Trace Analysis	56.02	42.03	25.55
w/o Refinement	62.71	45.72	28.89
Module Refinement	64.54	47.92	31.74
w/o Reranking	50.97	24.32	16.24

decrease across all metrics, with an average drop of 8.94% and specific declines of 9.25%, 9.04%, and 8.53% at the file, module, and function levels. These results emphasize that execution-based call hierarchies are essential for establishing causal links between tests and code, effectively mitigating the noise inherent in static analysis. Excluding domain knowledge leads to an average 7.37% reduction in performance. This component facilitates the alignment of issue requirements with project-specific test identifiers by leveraging historical commit data to bridge terminology gaps. The removal of the refinement stage produces a smaller average decline of 4.37%, suggesting that in the majority, the IssueExec achieves robust localization performance within the space covered by existing tests alone. Replacing the full refinement process with module-level refinement limits the search space while preserving performance near the full IssueExec configuration. This observation suggests that when operating under restricted budgets, narrowing the refinement scope can further improve system efficiency with minimal impact on localization accuracy. The substantial performance loss in the absence of reranking is attributed to the F1@3 experimental setting, as this module provides more accurate importance-based ordering of entities. Furthermore, prioritizing the most probable edit locations at the top of the recommendation list reflects a method design that aligns with the requirements of real-world software maintenance tasks.

6 Threats to Validity

Internal Validity The effectiveness of IssueExec's domain knowledge enhancement depends on the availability and quality of historical commit data. If a repository has insufficient commit history or maintains messy, non-atomic commit logs, the AST-based mining process may fail to extract meaningful domain tokens, potentially weakening the semantic alignment between issues and tests. While our framework implements filtering mechanisms to exclude low-quality commits, the reliance on historical signals remains a potential internal threat. Future work could involve incorporating external documentation or API references to supplement missing local history.

External Validity A primary external threat is the generalizability of IssueExec to projects that are not well-maintained or have limited test suites. Our approach assumes that tests can serve as executable requirements. In scenarios where coverage is significantly low, the ground-truth edit locations may not be captured within any execution traces. However, our empirical study suggests that even in large-scale repositories, existing tests often exercise core functional paths, providing a high-recall starting point.

Another external threat is language and framework support. Currently, our implementation and evaluation focus exclusively on Python repositories and the 'sys.settrace' instrumentation. The characteristics of dynamic tracing and caller-callee relationship extraction may differ in statically typed languages like Java or C++. While the core methodology of test-mediated localization is

language-agnostic, the current lack of cross-language validation remains a threat to external validity. We plan to implement support for additional programming languages in future iterations.

Construct Validity The process of collecting full execution traces and constructing trace graphs can be computationally expensive for massive software systems. This overhead might raise concerns regarding the practical utility of the tool in rapid CI/CD environments. To mitigate this threat, we emphasize that trace collection is primarily an offline preprocessing step. Furthermore, this process can be optimized through incremental trace updates where only affected tests are re-executed and traced. Such engineering optimizations will ensure that the system remains scalable as the project evolves.

7 Related Work

Information Retrieval and Bug Localization. Retrieval-based approaches compute similarity between issues and code to bridge the gap between natural language and formal syntax [5, 61]. Early works relied on sparse retrievers like BM25 [40] utilizing lexical overlap. To improve accuracy, subsequent studies introduced query reformulation with contextual cues [39] or composed richer evidence sources such as code change histories and metadata [47, 56]. While effective, these enhancements introduce sensitivity to modeling choices, where configuration substantially impacts localization performance [46]. More recently, dense retrievers such as CodeSage [59] and CodeRankEmbed [45] have emerged to capture semantic relationships beyond keyword matching. This paradigm has been further adapted to specialized scenarios: CoRet [19] incorporates call graph dependencies, BLAZE [10] addresses cross-project settings, and other models target concurrent programs [42]. However, as noted in classic studies, IR-based methods remain largely static proxies for relevance and continue to struggle with the "vocabulary mismatch" problem where informal issue descriptions and formal code syntax lack sufficient overlap [33].

Traditional Fault Localization and Traceability. Our work is fundamentally grounded in the long-standing challenges of Spectrum-based Fault Localization (SBFL) and Traceability Link Recovery (TLR). SBFL techniques, such as Tarantula [28] and Ochiai [1], utilize program spectra from passing and failing tests to rank suspicious code entities. While foundational, empirical studies show that SBFL effectiveness varies significantly across scales and often requires combination with additional signals [15, 26, 29]. Crucially, these methods rely on pre-existing failing tests, limiting their applicability in general issue resolution where such tests are often absent. Traceability Link Recovery (TLR) aims to link requirements to code but remains costly to maintain and recover automatically [5, 7]. Related research in Feature Location [16] and dynamic analysis has demonstrated that execution scenarios and traces can provide direct evidence of functional behavior [17, 41]. However, interpreting these dynamic traces has historically relied on manual effort or complex static analysis. Bridging this gap by automating diagnostic reasoning on execution traces—potentially via LLMs—remains an open challenge for recovering dynamic traceability links at scale.

Procedure-based Approaches employ hierarchical pipelines to narrow the search space. Agentless [51] pioneered a three-phase paradigm, while SWE-Fixer [54] streamlines this with BM25 retrieval and reranking. BugCerberus [11] extends to statement-level localization with program slicing, and PatchPilot [31] adds reproduction and refinement stages. While efficient, these rigid pipelines are prone to irreversible error propagation, particularly when the initial retrieval step relies on suboptimal queries or configurations [39, 46].

Agent-based Approaches frame localization as sequential decision-making. SWE-Agent [55] designs an Agent-Computer Interface for navigation, while OpenHands [48] provides Docker-sandboxed environments. Multi-agent systems like MASAI [6] and SWE-Search [4] distribute tasks across sub-agents. Graph-based methods have also emerged: LocAgent [12] constructs heterogeneous graphs with multiple edge types; RepoGraph [36] supports k-hop retrieval; CodexGraph [32]

enables Cypher-based querying. Despite their sophistication, these methods navigate *structural* rather than *functional* relationships, missing disconnected targets.

8 Conclusion

This paper revisits issue localization from a requirement-centric perspective and argues that *test suites can serve as executable requirements* that bridge the abstraction gap between natural-language issues and code. Motivated by both theoretical analysis and large-scale empirical evidence, we propose IssueExec, a test-driven localization framework that leverages domain-knowledge-enhanced test representations for effective issue–test alignment and hierarchical execution trace analysis to filter infrastructure noise and pinpoint requirement-central code.

Extensive experiments on SWE-bench Lite demonstrate that IssueExec achieves state-of-the-art localization performance across file, module, and function levels. When integrated into an end-to-end issue resolution pipeline, the improved localization quality translates into tangible gains in downstream patch generation, confirming that precise, requirement-aware localization is a critical enabler for automated software maintenance. Our in-depth analysis further reveals that the advantages of test-driven localization are most pronounced for complex, multi-file issues, while also highlighting limitations arising from incomplete test coverage and noisy execution traces.

Data Availability

The source code of IssueExec, LLM prompts, detailed analysis from the theoretical motivation and empirical study, illustrative case studies, and all other supplementary materials are available at [3].

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2006. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 39–46.

[2] Gojko Adzic. 2011. *Specification by example: how successful teams deliver the right software.* Simon and Schuster.

[3] Anonymous. 2025. IssueExec — sites.google.com. https://sites.google.com/view/issueexec. [Accessed 2026-01-30].

[4] Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. 2024. Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement. *arXiv preprint arXiv:2410.20285* (2024).

[5] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. 2002. Recovering traceability links between code and documentation. *IEEE transactions on software engineering* 28, 10 (2002), 970–983.

[6] Daman Arora, Atharv Sonwane, Nalin Wadhwa, Abhav Mehrotra, Saiteja Utpala, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. 2024. Masai: Modular architecture for software-engineering ai agents. *arXiv preprint arXiv:2406.11638* (2024).

[7] Thazin Win Win Aung, Huan Huo, and Yulei Sui. 2020. A literature review of automatic traceability links recovery for software change impact analysis. In *Proceedings of the 28th International Conference on Program Comprehension*. 14–24.

[8] Kent Beck. 2003. *Test-driven development: by example.* Addison-Wesley Professional.

[9] Marcel Böhme, Ezekiel O Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 117–128.

[10] Partha Chakraborty, Mahmoud Alfadel, and Meiyappan Nagappan. 2025. BLAZE: Cross-language and cross-project bug localization via dynamic chunking and hard example learning. *IEEE Transactions on Software Engineering* (2025).

[11] Jianming Chang, Xin Zhou, Lulu Wang, David Lo, and Bixin Li. 2025. Bridging Bug Localization and Issue Fixing: A Hierarchical Localization Framework Leveraging Large Language Models. *arXiv preprint arXiv:2502.15292* (2025).

[12] Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025. Locagent: Graph-guided llm agents for code localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 8697–8727.

[13] Agnieszka Ciborowska and Kostadin Damevski. 2022. Fast changeset-based bug localization with BERT. In *Proceedings of the 44th international conference on software engineering*. 946–957.

[14] Thomas M Cover. 1999. *Elements of information theory.* John Wiley & Sons.

[15] Higor A de Souza, Marcos L Chaim, and Fabio Kon. 2016. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *arXiv preprint arXiv:1607.04347* (2016).

[16] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *Journal of software: Evolution and Process* 25, 1 (2013), 53–95.

[17] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. 2001. Aiding program comprehension by static and dynamic feature analysis. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001.* IEEE, 602–611.

[18] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. 2003. Locating features in source code. *IEEE Transactions on software engineering* 29, 3 (2003), 210–224.

[19] Fabio James Fehr, Luca Franceschi, Giovanni Zappella, et al. 2025. CoRet: Improved Retriever for Code Editing. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 775–789.

[20] Farid Feyzi and Saeed Parsa. 2019. Inference: effective fault localization based on information-theoretic analysis and statistical causal inference. *Frontiers of Computer Science* 13, 4 (2019), 735–759.

[21] GitHub. 2025. Copilot: Faster, Smarter, and Built for How You Work Now. https://github.blog/ai-and-ml/github-copilot/copilot-faster-smarter-and-built-for-how-you-work-now/ Accessed: 2025.

[22] Ralph VL Hartley. 1928. Transmission of information 1. *Bell System technical journal* 7, 3 (1928), 535–563.

[23] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79.

[24] DM Hutton. 2009. Clean code: a handbook of agile software craftsmanship. *Kybernetes* 38, 6 (2009), 1035–1035.

[25] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code coverage at Google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 955–963.

[26] Jiajun Jiang, Ran Wang, Yingfei Xiong, Xiangping Chen, and Lu Zhang. 2019. Combining spectrum-based fault localization and statistical debugging: An empirical study. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 502–514.

[27] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770* (2023).

[28] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 273–282.

[29] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. 2017. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 114–125.

[30] Stanislav Levin and Amiram Yehudai. 2017. The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 35–46.

[31] Hongwei Li, Yuheng Tang, Shiqi Wang, and Wenbo Guo. 2025. PatchPilot: A Cost-Efficient Software Engineering Agent with Early Attempts on Formal Verification. *arXiv preprint arXiv:2502.02747* (2025).

[32] Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Qizhe Shieh, and Wenmeng Zhou. 2025. Codexgraph: Bridging large language models and code repositories via code graph databases. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. 142–160.

[33] Andrian Marcus and Jonathan I Maletic. 2003. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 125–135.

[34] Nachiappan Nagappan, E Michael Maximilien, Thirumalesh Bhat, and Laurie Williams. 2008. Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Software Engineering* 13, 3 (2008), 289–302.

[35] Feifei Niu, Chuanyi Li, Kui Liu, Xin Xia, and David Lo. 2025. When Deep Learning Meets Information Retrieval-based Bug Localization: A Survey. *Comput. Surveys* 57, 11 (2025), 1–41.

[36] Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2024. Repograph: Enhancing ai software engineering with repository-level code graph. *arXiv preprint arXiv:2410.14684* (2024).

[37] Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2024. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139* (2024).

[38] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. 2012. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering*. 1–11.

[39] Mohammad Masudur Rahman and Chanchal K Roy. 2018. Improving ir-based bug localization with context-aware query reformulation. In *Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 621–632.

[40] Stephen Robertson and Hugo Zaragoza. 2009. *The probabilistic relevance framework: BM25 and beyond*. Vol. 4. Now Publishers Inc.

[41] Maher Salah, Spiros Mancoridis, Giuliano Antoniol, and Massimiliano Di Penta. 2006. Scenario-driven dynamic analysis for comprehending large software systems. In *Conference on Software Maintenance and Reengineering (CSMR'06)*. IEEE, 10–pp.

[42] Shuai Shao and Tingting Yu. 2023. Information retrieval-based fault localization for concurrent programs. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1467–1479.

[43] Samuel Sanford Shapiro and Martin B Wilk. 1965. An analysis of variance test for normality (complete samples). *Biometrika* 52, 3-4 (1965), 591–611.

[44] Marius Smytzek, Martin Eberlein, Lars Grunske, and Andreas Zeller. 2025. How Execution Features Relate to Failures: An Empirical Study and Diagnosis Approach. *ACM Transactions on Software Engineering and Methodology* (2025).

[45] Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. 2024. CoRNStack: High-quality contrastive data for better code retrieval and reranking. *arXiv preprint arXiv:2412.01007* (2024).

[46] Chakkrit Tantithamthavorn, Surafel Lemma Abebe, Ahmed E Hassan, Akinori Ihara, and Kenichi Matsumoto. 2018. The impact of IR-based classifier configuration on the performance and the effort of method-level bug localization. *Information and Software Technology* 102 (2018), 160–174.

[47] Shaowei Wang and David Lo. 2016. Amalgam+: Composing rich information sources for accurate bug localization. *Journal of Software: Evolution and Process* 28, 10 (2016), 921–942.

[48] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741* (2024).

[49] Robert White, Jens Krinke, and Raymond Tan. 2020. Establishing multilevel test-to-code traceability links. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 861–872.

[50] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[51] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489* (2024).

[52] Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighoff. 2023. C-Pack: Packaged Resources To Advance General Chinese Embedding. arXiv:2309.07597 [cs.CL]

[53] Yan Xiao, Jacky Keung, Kwabena E Bennin, and Qing Mi. 2019. Improving bug localization with word embedding and enhanced convolutional neural networks. *Information and Software Technology* 105 (2019), 17–29.

[54] Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. 2025. Swe-fixer: Training open-source llms for effective and efficient github issue resolution. *arXiv preprint arXiv:2501.05040* (2025).

[55] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.

[56] Klaus Changsun Youm, June Ahn, and Eunseok Lee. 2017. Improved bug localization based on code change histories and bug reports. *Information and Software Technology* 82 (2017), 177–192.

[57] Zhongming Yu, Hejia Zhang, Yujie Zhao, Hanxian Huang, Matrix Yao, Ke Ding, and Jishen Zhao. 2025. Orcaloca: An llm agent framework for software issue localization. *arXiv preprint arXiv:2502.00350* (2025).

[58] Andy Zaidman, Bart Van Rompaey, Arie Van Deursen, and Serge Demeyer. 2011. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering* 16, 3 (2011), 325–364.

[59] Dejiao Zhang, Wasi Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. 2024. Code representation learning at scale. *arXiv preprint arXiv:2402.01935* (2024).

[60] Xin Zhang, Yanzhao Zhang, Dingkun Long, Wen Xie, Ziqi Dai, Jialong Tang, Huan Lin, Baosong Yang, Pengjun Xie, Fei Huang, et al. 2024. mgte: Generalized long-context text representation and reranking models for multilingual text retrieval. *arXiv preprint arXiv:2407.19669* (2024).

[61] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International conference on software engineering (ICSE)*. IEEE, 14–24.