

Improved Bug Localization with AI Agents Leveraging Hypothesis and Dynamic Cognition

Asif Mohammed Samir

Dalhousie University
Halifax, NS, Canada
asifsamir@dal.ca

Mohammad Masudur Rahman

Dalhousie University
Halifax, NS, Canada
masud.rahman@dal.ca

Abstract

Software bugs cost technology providers (e.g., AT & T) billions annually and cause developers to spend roughly 50% of their time on bug resolution. Traditional methods for bug localization often analyze the suspiciousness of code components (e.g., methods, documents) in isolation, overlooking their connections with other components in the codebase. Recent advances in Large Language Models (LLMs) and agentic AI techniques have shown strong potential for code understanding, but still lack causal reasoning during code exploration and struggle to manage growing context effectively, limiting their capability. In this paper, we present a novel agentic technique for bug localization –*CogniGent*– that overcomes the limitations above by leveraging multiple AI agents capable of causal reasoning, call-graph-based root cause analysis and context engineering. It emulates developers-inspired debugging practices (a.k.a., dynamic cognitive debugging) and conducts hypothesis testing to support bug localization. We evaluate CogniGent on a curated dataset of 591 bug reports using three widely adopted performance metrics and compare it against six established baselines from the literature. Experimental results show that our technique consistently outperformed existing traditional and LLM-based techniques, achieving MAP improvements of 23.33–38.57% at the document and method levels. Similar gains were observed in MRR, with increases of 25.14–53.74% at both granularity levels. Statistical significance tests also confirm the superiority of our technique. By addressing the reasoning, dependency, and context limitations, CogniGent advances the state of bug localization, bridging human-like cognition with agentic automation for improved performance.

CCS Concepts

- Software and its engineering → Software maintenance tools; Maintaining software; Software testing and debugging.

Keywords

Bug Localization, LLM, Agentic AI, Cognition, Debugging, Software Engineering, Information Retrieval

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.

ICPC '26, Rio de Janeiro, Brazil

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2026/04
<https://doi.org/XXXXXXX.XXXXXXX>

ACM Reference Format:

Asif Mohammed Samir and Mohammad Masudur Rahman. 2026. Improved Bug Localization with AI Agents Leveraging Hypothesis and Dynamic Cognition. In *Proceedings of 34th IEEE/ACM International Conference on Program Comprehension (ICPC 2026), Research Track (ICPC '26)*. ACM, New York, NY, USA, 13 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 Introduction

Software bugs pose major challenges in security and operational reliability, costing technology providers (e.g., AT&T [19, 91], Microsoft [81]) billions of dollars every year. Experts from major technology companies (e.g., Google, Meta, Microsoft) identify bug resolution as one of the most pressing challenges in software development and maintenance [94]. This concern is reinforced by industry studies showing that nearly 42% of organizations lose ≈\$1 million annually due to poor software quality [76]. This leads to developers spending about 50% of their programming time on bug resolution [10, 17, 55]. Such debugging efforts are cognitively demanding, requiring reasoning and deep code understanding [30, 45, 87, 89].

Software bugs are submitted to issue-tracking systems (e.g., Jira, Bugzilla, GitHub) as bug reports. These reports are the primary source of information for developers to understand the problem, inspect the project repository, and locate the buggy code. However, the quality of bug reports often varies depending on the reporter's expertise in explaining the problem and writing style [62]. As a result, even experienced developers relying on the bug reports may struggle to identify the faulty code [61]. A recent study with developers found that 79% of debugging time was spent in long episodes (15–33 minutes) dedicated to locating and diagnosing bugs, but many of them ended without successfully identifying the faulty code [4]. Therefore, bug localization has been a long-standing challenge for developers. Unfortunately, automated approaches have not yet matured into reliable, practical tools to date [14].

Traditional methods for bug localization can be grouped into two main categories: spectrum-based and Information Retrieval (IR)-based approaches. Spectrum-based techniques rely on execution traces, which are not always available in practice [52, 69, 77]. In contrast, IR-based approaches attempt to localize bugs by matching tokens in bug reports with those in the source code [37, 42, 54, 78]. While IR-based methods are lightweight and fast, they rely on surface-level token matching, which makes them prone to vocabulary mismatch between natural language and source code [22], especially due to the variability in bug reports. To address the limitations, several avenues have been explored, such as query reformulations [12, 61, 63, 71], incorporating statistical insights from code change history, and leveraging past bug fixes [79, 90]. However, these techniques have not delivered significant improvements

over earlier methods [44] and do not capture the deeper semantic context (e.g., program semantics [18]) needed for accurate bug localization [59, 69].

Recent advancements in deep learning have shown great promise in understanding natural language and source code [1, 74], which could support bug localization [14, 59]. However, training deep learning models requires large amounts of relevant, high-quality data [69]. In contrast, prompt-based and agentic AI techniques [34, 92] present a promising alternative since they are powered by Large Language Models (LLMs), pre-trained on giant corpora. However, these approaches have been primarily optimized for complete issue resolution rather than detecting the location of bugs, which could lead to sub-optimal performance. Besides the above, contemporary approaches suffer from three major challenges as follows.

(a) Determining fault-proneness of code components overlooking their dependencies: Traditional approaches for bug localization often determine the fault-proneness of a code component (e.g., method or source document) in isolation, overlooking its interactions with or dependencies on other components [36, 62, 69, 78, 79]. However, many bugs could trigger during runtime due to error propagation across components and could span across multiple components. Recent graph-based methods attempt to incorporate structural information to represent program flow and syntactic relationships [49, 75]; however, their analysis often remains confined to the methods within the same source document, which might not be sufficient. Detection of such propagated bugs warrants an analysis of not only individual code components but also their dependencies within the code base.

(b) Lack of reasoning about root causes during code exploration: Bug reports often describe the failure symptoms of a software application rather than its faulty code since they are written by software users or testers [33, 69]. Existing IR-based techniques rely on textual and semantic similarity between bug reports and source code documents to retrieve and detect faulty code [11, 78]. Given their simplicity, these similarity measures might fail to capture the causal links between the reported symptoms and the faulty code [69]. Even recent agentic techniques (e.g., LocAgent [13]), powered by LLMs, assume the presence of code entities within the issue reports to choose the starting points for their code exploration. However, presence of such cues in bug reports cannot always be guaranteed [69]. Thus, such a lack of reasoning in existing techniques [13] when finding candidates to explore could result in overlooking important dependencies or exploring irrelevant paths during bug localization [33].

(c) Lack of effective context management during code exploration: During code navigation, agentic techniques perform iterative reasoning and extend the context length at each step with new information [13, 33, 92]. Although recent LLMs support much larger contexts (e.g., 128k tokens [3]), studies show that performance declines as context length increases [29, 46]. Irrelevant information can further worsen this effect [85]. Thus, contemporary techniques might suffer from low performance and high inference costs during bug localization.

In this paper, we propose a novel technique, *CogniGent*, that employs multiple AI agents and conducts hypothesis testing to localize software bugs. First, *CogniGent* emulates developers' debugging

practices (a.k.a., dynamic cognitive debugging) and formulates multiple hypotheses on the *root cause* of a bug based on the symptoms found in the bug report. Second, its AI agents apply causal reasoning to dynamically select starting points within the code and explore *dependent code* (e.g., invoked methods) to assess their suspiciousness. To facilitate this process, our technique employs a novel algorithm –*Click2Cause*– that traverses the call graph via depth-first exploration. It also applies *scratchpad-based context management* [39] to prevent context overload during code exploration. Finally, an independent observer agent evaluates the explored code chains, the generated artifacts or evidence, and determines their fitness to the hypotheses, delivering the final ranked list of faulty components at different granularity levels (e.g., methods, documents).

We evaluate our technique using 591 bug reports curated from an existing dataset of Samir et al. [70] and determine its performance at both the method and document levels using three widely adopted metrics: Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), and HIT@K. We compared our approach with six baseline techniques [13, 21, 62, 68, 69, 86] from two major areas—IR and LLM. *CogniGent* consistently outperformed existing traditional and LLM-based techniques, achieving MAP improvements of 23.33–38.57% at the document and method levels. Similar gains were observed in MRR, with increases of 25.14–53.74% across the same levels. These results underscore the effectiveness and superiority of *CogniGent* in software bug localization.

Thus, this research makes the following contributions—

- A novel agentic emulation of *dynamic cognitive debugging*, that represents human cognition during debugging by formulating and testing hypotheses and systematically reasoning about causal relationships between bugs and code.
- A novel agentic workflow, *CogniGent*, that adopts dynamic cognitive debugging and employs Click2Cause, a repository-level code navigation algorithm, and scratchpad-based context management to effectively localize the faulty code (e.g., methods, documents).
- An extensive evaluation of *CogniGent* using three widely used metrics on a dataset of 591 recent bug reports annotated at both method and document levels.
- A replication package [6] containing the prototype, curated dataset, and configuration details for third-party use.

2 Motivating Example

In this section, we present a motivating example to demonstrate the effectiveness of our technique in bug localization. Let us consider a real bug report from the *Apache HBase* project (check Table 1) that provides only a generic explanation of the bug. Here, the bug caused the failsafe snapshot of a table to survive from a rollback operation, leaving redundant artifacts in the cluster.

If we examine the source code in Fig. 1, at first glance, it appears correct—it calls `deleteSnapshot()` method after restoration, suggesting proper cleanup operation. However, given the reported bug, a developer might assume that the deletion may not be taking place after rollback. To verify this, they might navigate (e.g., *Ctrl+Click*) to the `deleteSnapshot()` method (line 20) for a close inspection, only to find it syntactically and logically correct. Given

Table 1: Bug Report (Apache HBase 15801258)

Title: The failsafe snapshot should be deleted after rollback successfully.		
Description: When a table exists and is in a disabled state, HBase supports restoring from a snapshot. In this scenario, HBase creates a failsafe snapshot for the disabled table and rolls back using it when restore fails. However, the failsafe snapshot remains even after successful rollback, and it should be deleted.		
Technique	Approach	Rank
Baseline IR (Lucene [21])	Textual matching (Bug Report)	87
Agentless [86]	Project structure cue + Semantic similarity	35
LocAgent [13]	Agentic	8
CogniGent	Dynamic Cognitive Debugging	1

```
@@ -2663,6 +2663,13 @@ @@ HBaseAdmin.java:restoreSnapshot
1  public void restoreSnapshot(final String snapshotName, ...)
2  {
3      ...
4      try {
5          get(internalRestoreSnapshotAsync(snapshotName,
6              <-- tableName, restoreAcl, null),
7              syncWaitTimeout, TimeUnit.MILLISECONDS);
8      } catch (IOException e) {
9          if (takeFailSafeSnapshot) {
10              try {
11                  get(internalRestoreSnapshotAsync(
12                      failSafeSnapshotName, tableName, restoreAcl, null),
13                      syncWaitTimeout, TimeUnit.MILLISECONDS);
14                  throw new RestoreSnapshotException("Restore failed
15                      <-- but rollback succeeded", e);
16              } catch (IOException ex) {
17                  throw new RestoreSnapshotException("Rollback
18                      <-- failed", ex);
19              }
20          } else {
21              throw new RestoreSnapshotException("Restore failed",
22                  e);
23      }
24      if (takeFailSafeSnapshot)
25          deleteSnapshot(failSafeSnapshotName);
26  }
```

Figure 1: Buggy Code for Bug #15801258 from HBase

the role of the rollback operation, the developer might then inspect `internalRestoreSnapshotAsync()` method (line 3) to check whether it handles cleanup internally. As shown in Fig. 2, the method only initiates the restore or rollback operation but contains no deletion logic. The code contains two rollback operations within two-level nested try-catch blocks, complicating the control flows. When the first restore operation fails, an exception is caught, and a second attempt at restoration is made. When the second attempt to restoration is successful, a new `RestoreSnapshotException` is rethrown unexpectedly (line 13). This rethrow causes control flow to exit the method before reaching the `deleteSnapshot()` (line 20) placed after the try-catch block. Even if `takeFailSafeSnapshot` is set to true, the deletion will not take place, and the failsafe snapshot will remain intact.

This example highlights the importance of human cognitive processes in debugging, which involves formulating a hypothesis, collecting evidence, and testing the hypothesis. Emulating these cognitive abilities through AI agents, our technique *CogniGent* ranks the buggy `restoreSnapshot()` method first. Since the bug report lacks direct references to code elements, approaches relying on textual, semantic, or structural cues struggle to establish relevance. As a result, traditional IR-based baseline [21] that relies solely on textual matching ranks it 87th. Agentless [86], which integrates semantic similarity with project-structure cues, performs better but still ranks it 35th. LocAgent [13], an agentic approach that explores code using syntactic cues, ranks the method eighth.

3 Methodology

Fig. 2 shows the schematic diagram of our proposed technique, *CogniGent*, for software bug localization. We discuss the major design steps of our technique as follows.

3.1 Design of AI Agents

In our agentic workflow, six role-specific agents form a pipeline to support bug localization: a restructuring agent (reorganize a bug report), a retrieval agent (collect candidate code segments), a filtering agent (filter out irrelevant candidates), a hypothesis agent (generate hypotheses), an investigation agent (test hypotheses), and an observer agent (validate hypotheses). We implement this workflow using the LangGraph library [40], combining varying sizes of LLMs (e.g., Devstral [2]) to balance efficiency, cost, and performance. The agents communicate through a shared LangGraph state [41], where each agent uses only the parts relevant to its role and updates only the portion it contributes (e.g., adding hypotheses).

Since prompt design is central to the effectiveness of the workflow, we construct each agent's system prompt following established prompt-engineering practices [84] and employ few-shot chain-of-thought prompting [47, 82]. We also apply a meta-prompting approach [93] and iteratively refine the prompts with the help of LLMs (e.g., ChatGPT).

3.2 Graph-Based Indexing of Source Code

We collect candidate code segments (e.g., methods, constructors) of a software project using Information Retrieval (IR), which warrants the indexing of source code (Step 1-2, Fig. 2). Rather than treating the entire source code document as the retrieval unit [62, 69], we follow Liu et al. [48] and index code segments with their functionality and relationships. In particular, we use Neo4j [53] for its graph construction capability and textual search, powered by the Lucene engine [21]. We construct the index as a directed heterogeneous graph $G = (V, E)$, where nodes V represent code segments (methods, constructors), each uniquely identified by an ID, and edges E capture the connections among them. To extract these connections (e.g., invoke, inherit), we use JavaParser [20], which automatically identifies relationships through runtime type resolution. We indexed the source code of 132 versions from 15 software systems in the dataset to ensure that each bug was checked against the system and version for which it was originally reported [69].

3.3 Retrieving and Filtering Documents

3.3.1 Restructuring Bug Reports. Several parts of bug reports (e.g., title, description) are often used as queries to retrieve potential buggy candidates. However, they might contain noisy or irrelevant information (e.g., emotional expressions, unrelated context), which can cause textual retrieval methods (e.g., Apache Lucene [21]) to return irrelevant code. Standard preprocessing techniques (e.g., stop-word removal, stemming) are insufficient for reducing such noise, as they risk losing the continuous textual understanding required by LLMs. Therefore, we employ a lightweight LLM (e.g., Qwen-Coder, 1.5B parameters) to restructure bug reports and remove their noise before the retrieval operation (Step 4, Fig. 2).

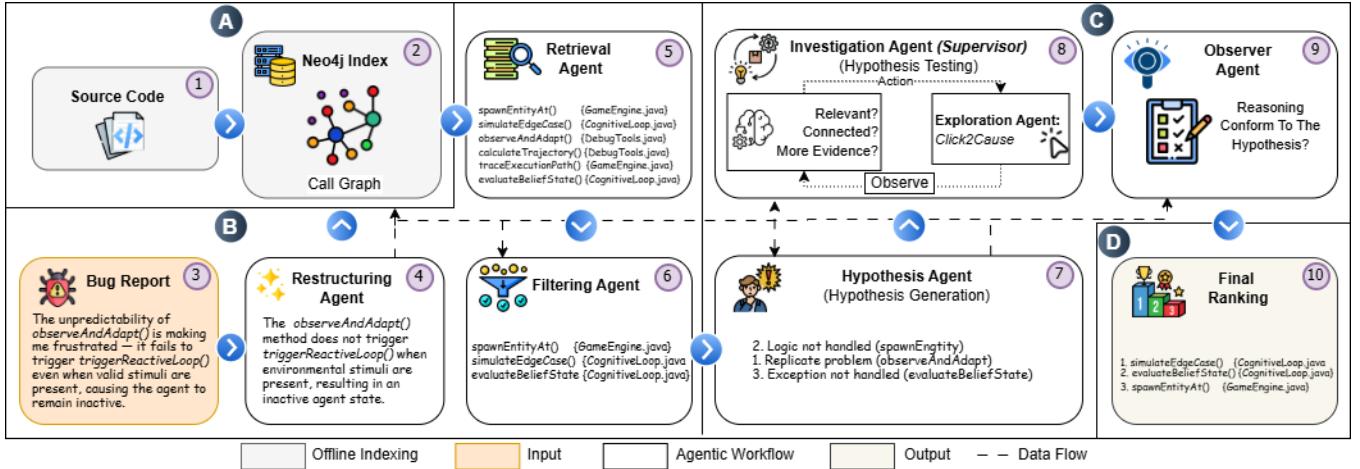


Figure 2: Schematic Diagram of CogniGent: (A) Graph-Based Indexing of Source Code, (B) Retrieving and Filtering Documents, (C) Dynamic Cognitive Debugging, (D) Final Ranking.

Table 2: Tools Used in CogniGent

Tool Name	Used By	Function
<code>retrieveCandidates</code>	Retrieval Agent	Retrieve the potential buggy source code segments (e.g., methods) from the bug reports.
<code>exploreCallChain</code>	Supervisor Agent	Determines and retrun the most relevant chain of code segment(s).
<code>getCodeSegment</code>	Exploration Agent	Retrieves the source code of a target code segment located in the neighborhood of the current segment under exploration.

3.3.2 Retrieving Potentially Buggy Segments. We use the restructured bug reports to retrieve the top (e.g., 100) potentially buggy code segments from our Neo4j index (Step 5, Fig. 2). We restrict the search to the system and version of the code associated with each bug report to avoid irrelevant results. We then pass the retrieved results to the subsequent filtering stage of our agentic workflow.

3.3.3 Filtering Documents. We filter candidate segments using Intelligent Relevance Feedback (IRF) [69], where a lightweight LLM reasons about the likelihood a code segment might be connected to the reported bug (Step 5, Fig. 2). Because bug reports often describe error symptoms [33] and keyword matching alone may miss related code [24, 25], we leverage the contextual reasoning ability of LLMs to identify faulty candidate segments [69]. This step produces a small set of top-ranked candidates (e.g., 10).

3.4 Dynamic Cognitive Debugging

In our proposed technique, we apply Dynamic Cognitive Debugging (DCD) to bug localization. Prior studies identify debugging as a cognitively demanding task that requires programmers to construct and refine mental models to locate and fix errors [30, 45, 87, 89]. This process involves forming hypotheses, navigating the codebase to test them, and confirming or discarding those hypotheses based on relevant evidence. To emulate this important cognitive process, we leverage the reasoning capability of LLMs as AI agents, and employ them for code navigation and bug detection as follows.

3.4.1 Hypothesis Generation. During bug resolution, human experts typically form multiple hypotheses about the possible causes of a bug based on the bug report, available code and then investigate further [87]. Similarly, we provide the bug report and the filtered code segments above (Step 6) to a powerful LLM (e.g., Devstral 20B parameters), and leverage the model to generate multiple *competitive hypotheses* (Step 7, Fig. 2). For each code segment, we guide the LLM to generate a hypothesis on the root cause of the reported bug, along with a confidence category (e.g., high, medium, or low) and a confidence score (e.g., 0-1). These categories represent the relative likelihood of each segment being buggy compared to the other candidates. We then retain the segments with high or medium confidence for deeper investigation in the subsequent steps.

3.4.2 Investigation: Hypothesis Testing using AI Agents. In this step, we simulate how developers investigate and navigate code during debugging (Step 8, Fig. 2). For example, when a bug report mentions a *race condition*, a developer may reason about it, inspect code related to threading functionality and then follow the relevant call chain for closer examination. Similarly, our technique applies contextual reasoning and autonomously determine whether the current segment provides sufficient evidence or whether parts of it (e.g., invoked methods) should be explored further.

We model this cognitive process with two specialized agents: a *supervisor* and an *explorer*. The supervisor is built on the ReAct framework (think, act, observe) [88] to guide the exploration agent. For each retained code segment from the previous step (Step 7, Fig. 2), a dedicated *supervisor* agent is instantiated with its associated hypothesis and the bug report as part of its reasoning context. The agent reviews the segment, collects evidence, and determines whether further exploration is required to test the hypothesis. When an additional analysis is necessary, the agent first identifies the calls to follow from the current code segment, reasoning about their relevance to the reported symptoms. It then dynamically instantiates an independent *exploration agent* and delegates the exploration task. The agent receives the bug report, the current code segment, the

Algorithm 1: Click2Cause: Call Chain Analysis

```

Input: bugReport, startSeg, callsToExplore, maxDepth,
       confidence threshold  $\tau$ , initial confidence  $C_{parent}$ 
Output: Most probable call chain  $C^*$  with confidence  $C_{LLM}(C^*)$ 
1 Global: visited  $\leftarrow \emptyset$ , scratchPad  $\leftarrow \emptyset$ ,  $C^* \leftarrow (\emptyset, 0)$ 
2 foreach call  $\in$  callsToExplore do
3   DFS(startSeg, [call], depth=1, Cparent, bugReport)
4 return  $C^*$  // Return best call chain
5 Recursive DFS(seg, path, depth, Cparent, bugReport):
6   if seg  $\in$  visited or depth  $>$  maxDepth then return;
7   Add seg to scratchPad, visited, and path
8    $\mathcal{A}(\text{seg}) \leftarrow \text{LLMReason}(\text{bugReport}, \text{path})$ 
9    $C_{LLM}(\text{seg}) \leftarrow \mathcal{A}(\text{seg}).\text{conf}$ 
10  if  $C_{LLM}(\text{seg}) < C_{parent}$  then
11    Backtrack() // Prune weak branch
12  else
13    if  $C_{LLM}(\text{seg}) > C_{LLM}(C^*)$  then
14       $C^* \leftarrow (\text{path}, C_{LLM}(\text{seg}))$ 
15    if  $C_{LLM}(\text{seg}) \geq \tau$  then
16      return // early stop
17    foreach next  $\in \mathcal{A}(\text{seg}).\text{callsToExplore} do
18      DFS(next, path, depth+1,  $C_{LLM}(\text{seg})$ , bugReport)
19 return$ 
```

selected calls to explore prioritized by suspiciousness, and a dynamically determined maximum depth. We implement the exploration agent as a tool (Table 2) that maintains a separate scratchpad-based [39, 88] reasoning context focused exclusively on relevant code segments, thereby preventing context confusion. [32]. The scratchpad is separate from CogniGent’s pipeline state (see 3.1) and exists only for the lifetime of the exploration agent, maintaining its reasoning traces during exploration.

To gather more information targeting a hypothesis, the exploration agent uses our proposed Click2Cause algorithm (Alg. 1) to traverse the call graph recursively. Unlike LocAgent [13], which employs a breadth-first search (BFS) strategy [16], Click2Cause applies a depth-first search (DFS) [16], mimicking the *Ctrl+Click* navigation familiar to developers [57]. The algorithm navigates method-call relationships stored in Neo4j, following a recursive *think-act cycle*. It hops through the call graph by making tool calls (Table 2) that retrieve the next callee method and its source code for inspection, and controls the maximum exploration depth along any single branch based on LLM’s reasoning. During traversal, the exploration LLM assigns confidence scores to each segment-chain based on its alignment with the reported bug symptoms. When a branch becomes less promising (i.e., chain confidence drops), Click2Cause backtracks to the previous decision point, prunes the current path from the scratchpad, and continues exploring alternative sibling branches. This backtracking and pruning process ensures that only promising paths remain in the agent’s working memory. Conversely, when a confidence score exceeds a threshold specified in the prompt (e.g., 90%), the exploration agent terminates early and returns the identified call chain to the supervisor agent. Otherwise, it returns the most promising chain observed.

Upon receiving a call chain, the supervisor agent reasons over it with respect to the hypothesis and the reported symptoms, and assigns a score. If the evidence supports the hypothesis, the chain is accepted as the likely root cause; otherwise, the supervisor either initiates further exploration or concludes based on the best evidence currently available.

This coordinated delegation strategy enables autonomous investigation to capture the buggy call chain through systematic context engineering [50]. In contrast to contemporary techniques [13], our technique uses causal reasoning from the reported symptoms to determine candidates to explore.

3.4.3 Observer. Once all investigations are complete, an independent observer agent evaluates each candidate explanation—either a single code segment or a call chain—along with its associated hypothesis and reasoning traces (Step 8, Fig. 2). The observer assesses how well each hypothesis is supported by the corresponding code and reasoning, and assigns a confidence score. The final suspiciousness score combines the supervisor’s reasoning confidence with the observer’s validation score, mitigating bias from either stage to find the most plausible root cause.

3.5 Final Ranking

Based on the confidence scores assigned by the observer agent, we rank the code segments and their associated call chains while preserving the order of related segments within each chain. Since investigation is performed on a limited set of segments filtered through hypothesis generation, and because our evaluation focuses on the top-K methods (e.g., top-10), any remaining positions are filled with other candidates in descending order of confidence, first from the hypothesis generation stage (Step 7, Fig. 2) and then from the filtering stage (Step 6, Fig. 2), while avoiding redundancy. Finally, to obtain document-level rankings, we map these methods to their corresponding documents in the same confidence order to determine the top-K documents. This process produces the final ranked list of locations most likely to be buggy at both the method and document levels.

4 Experiments

We curated a dataset of 591 bug reports from the work of Samir et al.[70], who extended the Bench4BL dataset [44] with recent reports. For evaluation, we employ three widely used metrics from the literature: Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), and HIT@K ($K=1, 5, 10$). We experiment with three different LLMs and compare our solution, CogniGent, against five relevant baseline techniques to demonstrate its effectiveness. Through these experiments, we address three research questions:

- **RQ₁:** (a) How does CogniGent perform in localizing software bugs? (b) How does it perform on different types of bugs? (c) How effective is it in localizing bugs that span multiple documents?
- **RQ₂**: How do the individual agentic modules of CogniGent contribute to its overall performance in bug localization?
- **RQ₃**: Can CogniGent outperform the relevant baseline techniques in bug localization?

4.1 Dataset Construction

Recent advances in agent-based solutions have motivated the construction of new datasets (e.g., [34]). However, these datasets were primarily created for issue-fixing tasks and contain not only bug reports but also feature requests, questions, and other tasks [28]. Such a mixture of issues can negatively affect the performance of bug localization techniques [28]. Therefore, we curated a dataset composed exclusively of bug reports for our investigation.

Table 3: Dataset Summary

Project	Systems	Versions	Count	NL	PE	ST
Apache	4	17	104	60	23	21
Spring	7	92	422	206	163	53
Wildfly	4	23	65	33	14	18
Total	15	132	591	299	200	92

NL = Natural Language, PE = Program Elements, ST = Stack Traces

We curated our dataset based on the extended version of Bench4BL constructed by Samir et al. [70], which includes $\approx 1.7k$ bug reports from 2018–2024. These reports were collected from Jira and GitHub issue tracking systems. We first confirmed the existence of ground-truth documents (e.g., faulty code) corresponding to the bug reports. While the original dataset contained only ground-truth documents, we augmented it with the methods modified during bug resolution and separated test components (e.g., methods, documents) as distinct entries to enhance applicability. Since agentic systems are computationally demanding and indexing multiple software versions is challenging, we selected specific versions of the subject systems to enable version-based retrieval, while ensuring the dataset contains diverse bug report types (e.g., natural language descriptions and program elements) [62]. We spent about 22 hours validating and refining the resulting dataset comprising 591 bug reports across 132 versions of 15 Java-based systems. Table 3 provides the summary of our curated dataset.

4.2 Evaluation Metrics

Mean Average Precision (MAP). Precision@K measures the accuracy of retrieved results up to rank-K for each occurrence of a buggy source code component (e.g., method or document) in the ranked list. Average Precision is the mean of these Precision@K values across all relevant (buggy) items for a given query. Consequently, the Mean Average Precision (MAP) is obtained by averaging the Average Precision scores across all queries Q in the dataset.

$$AP@K = \frac{1}{|D|} \sum_{k=1}^K P_k \times B_k \quad | \quad MAP = \frac{1}{|Q|} \sum_{q=1}^Q AP@K_q$$

Here, $AP@K$ is the average precision within the top- K results, where P_k is precision at rank k and B_k indicates if the k^{th} item is buggy (1) or not (0). MAP averages this across all queries q in Q , with D as the ground-truth set.

Mean Reciprocal Rank (MRR). Reciprocal Rank (RR) represents the position of the first relevant code component returned by a given method. It is calculated as the inverse of that component's rank within the ordered list of retrieved items for each query.

$$RR_q = \frac{1}{\text{Rank of First Relevant Item}} \quad | \quad MRR = \frac{1}{|Q|} \sum_{q=1}^{|Q|} RR_q$$

Here, MRR is the mean of all Reciprocal Ranks (RR_q) computed across the set of queries Q .

HIT@K. HIT@K [68] quantifies the percentage of queries for which at least one relevant component appears within the top- K retrieved results. Higher HIT@K values indicate better performance in bug localization techniques.

$$HIT@K = \frac{1}{|Q|} \sum_{q=1}^{|Q|} \begin{cases} 1, & r_q \in \mathcal{G} \\ 0, & \text{otherwise} \end{cases}$$

Here, r_q equals 1 if a query q retrieves a ground truth item in the top- K results (0 otherwise), in the set of all queries Q .

4.3 Experimental Setup

For our experiments, we selected three open-weight, instruction-tuned LLMs of varying scales with agent tool-calling support: LLaMA 3.3 70B [3], a general-purpose model; Devstral 20B [2], a Mistral variant fine-tuned for software engineering tasks; and Qwen3-Coder 30B [15], optimized for coding-related reasoning. These models have different types of specialization to support the interpretation of bug reports and the understanding of source code to aid bug localization, while also considering computational efficiency.

To balance performance and resource utilization, we adopted a hybrid execution setup. We used a lightweight 1.5B-parameter Qwen2.5-coder model on a local GPU-enabled system (NVIDIA GeForce RTX 2060) to perform restructuring and filtering tasks, while the larger models mentioned above were executed via a cloud-based multi-model gateway to handle reasoning and localization. The generation temperature was fixed at 0.5 to balance factual accuracy and creative reasoning [65] throughout all model interactions.

4.4 Evaluating CogniGent

Answering RQ₁ - Performance of CogniGent. We evaluate the performance of our proposed technique, CogniGent, with three key metrics- Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), and top-K results (K = 1, 5, 10). Table 4 summarizes the performance of our technique using three different LLMs.

From Table 4, we observe that CogniGent performs effectively in localizing bugs across different levels of granularity. Among all variants, CogniGent (Devstral) demonstrates the strongest performance, achieving a Mean Average Precision (MAP) of 0.407 at the document level. This reflects CogniGent's ability to rank buggy source documents higher than irrelevant ones. The Mean Reciprocal Rank (MRR) of 0.418 further indicates that the first relevant document typically appears within the top two results. Moreover, its HIT@1 score of 0.374 shows that, for approximately 37.4% of the bugs, the buggy document is ranked at the very top. The performance remains consistent when more results are considered, with HIT@5 = 0.475 and HIT@10 = 0.526, demonstrating CogniGent's strong localization ability. The Qwen-Coder variant exhibits comparable results, trailing slightly behind Devstral variant by 0.19%–1.72% across all metrics. On the other hand, CogniGent (LLaMA) performs

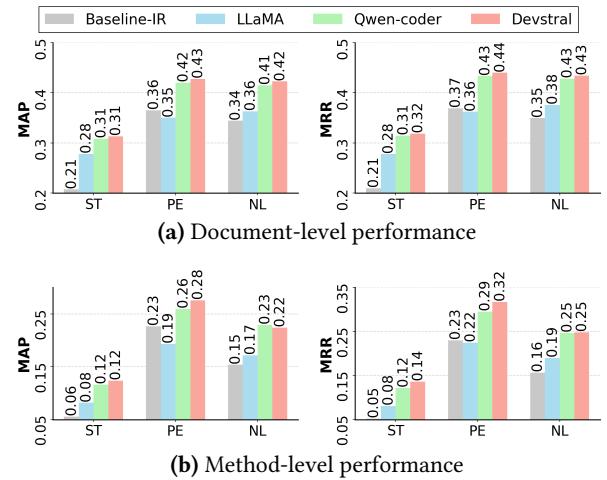
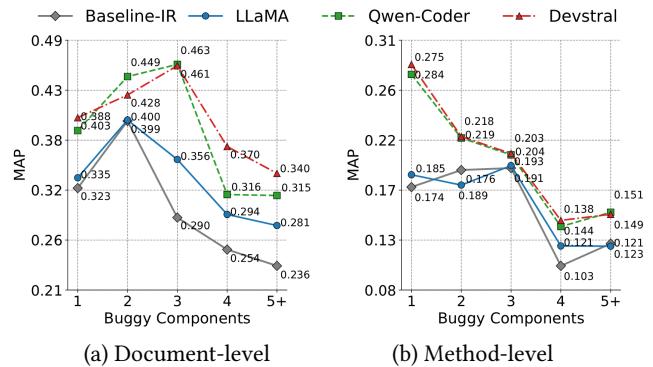
Table 4: Performance and Efficiency of CogniGent at Different Levels of Granularity

Techniques	Level	MAP	MRR	HIT@1	HIT@5	HIT@10	Cost/Instance (\$)	Time/Instance (s)
Baseline-IR (Lucene) Method	Document	0.330	0.334	0.254	0.443	0.487	–	–
	Method	0.163	0.165	0.122	0.217	0.246		
CogniGent (LLaMA) Method	Document	0.345	0.356	0.283	0.455	0.509	0.0011	2m 43s
	Method	0.165	0.185	0.135	0.232	0.268		
CogniGent (Qwen-Coder) Method	Document	0.400	0.412	0.360	0.470	0.515	0.0058	3m 50s
	Method	0.222	0.244	0.212	0.271	0.285		
CogniGent (Devstral) Method	Document	0.407	0.418	0.374	0.475	0.526	0.0026	2m 58s
	Method	0.226	0.254	0.227	0.272	0.289		

moderately well while outperforming the traditional IR-based baseline, achieving around 15% improvement in MAP and MRR, and up to 11.4% improvement in HIT@1. However, it still lags behind the Devstral and Qwen-Coder variants. At the method level, CogniGent (Devstral) also demonstrates substantial gains, improving over the IR baseline by 1.23%–53.94% across all metrics. Similar trends are observed for Qwen-Coder, with improvements ranging from 12.12% to 53.93%, indicating strong robustness even at finer code granularity.

In terms of efficiency, CogniGent (LLaMA) achieves the lowest cost per instance (i.e., bug report) at \$0.0011 (0.11 cents) with an average runtime of 2 minutes 43 seconds (Table 4). In comparison, Devstral costs 0.26 cents per instance with a comparable runtime, while Qwen-Coder, despite its similar performance, incurs more than double the cost and requires approximately 29% more time for execution. During our experiments, we observed that the LLaMA variant makes fewer tool calls, which possibly explains their lower cost, while their general-purpose design may contribute to their lower performance.

Existing work has categorized bug reports into three categories based on the presence of structured entities in them – Stack Trace (ST), Program Elements (PE), and Natural Language (NL) types [62]. We categorize the reports using regular expressions (regex) [62]. To demonstrate the robustness of our technique, we also evaluated its performance across these bug types. Fig. 3 shows CogniGent’s bug localization performance in terms of MAP and MRR. Our technique exhibits consistent trends across all variants, performing best on PE and NL reports while struggling with ST reports at the document level. For ST-type reports, CogniGent (Devstral) achieves a MAP of 0.31—about 34% higher than baseline IR and up to 14.28% higher than the other two variants. A similar pattern is observed in MRR, confirming that bug reports with stack traces remain the most difficult to localize. In contrast, the Devstral and Qwen-Coder variants report higher gains for PE and NL, performing comparably with MAP and MRR improvements of 17.15% and 19.09%, respectively. A similar trend appears at the method level, though overall scores decline due to finer granularity. CogniGent (Devstral) remains dominant across all types of bug reports, achieving MAP values of 0.28 for PE and 0.22 for NL, while the Qwen-Coder variant trails by 4–6%. CogniGent (LLaMA) shows a 25–40% drop compared to the other two variants but still improves over baseline IR by up to 0.34% for ST and 0.32% for NL, with a slight decrease for PE. Thus, despite the increased difficulty of method-level localization, CogniGent presents a better alternative than the baseline for bug localization.

**Figure 3: CogniGent’s Localization Performance Across Three Bug Types****Figure 4: Performance Across Single vs. Multi-Component Bugs**

Given the prevalence of propagated bugs, we examined CogniGent’s performance on bugs spanning multiple source components (e.g., methods, documents). We first grouped bug reports by their ground-truth counts—1, 2, 3, 4, or 5+ buggy documents or segments (e.g., methods)—and evaluated our techniques’ performance using Mean Average Precision (MAP) at both document and method levels. Fig. 4 shows CogniGent’s performance across these groups of bug reports. CogniGent (Devstral) achieves a MAP of 0.40 for

single-document bugs, improving to 0.43 and 0.46 for two and three documents (6.45% and 9.42% gains), but drops to 0.37 and 0.34 for four and five or more. Across all cases, it outperforms the IR baseline by 7.36–58.99%. Qwen-Coder follows a similar trend—starting at 0.39, surpassing Devstral by 4.6% for two-document bugs—and reaches up to 59.51% improvement over IR. LLaMA variant reports lower MAP scores overall but still achieves 0.28–19% gains over IR. At the method level, scores decline as bugs span more segments. CogniGent (Devstral) remains best, starting at 0.28 for single-segment bugs and dropping to 0.14–0.15 for broader scopes, with up to 39.70% improvement over IR. Qwen-Coder trails by 2–4%, while LLaMA drops to 0.12 for large bugs yet maintains 1.87–6.38% gains in other cases. Overall, while bug spread increases the difficulty of method-level reasoning, our technique effectively handles such cases through inter-component exploration and reasoning, achieving consistently strong localization performance.

RQ1 Summary: CogniGent significantly improves bug localization at various granularity levels, particularly with Devstral, reaching a high MAP score of 0.407. This performance can be attributed to our technique’s ability to reason across multiple source code segments (e.g., methods) while dynamically exploring inter-component dependencies to discover root causes of failures for bug localization.

Answering RQ₂ - Contribution of Different Modules. CogniGent consists of five major agentic modules that collectively contribute to bug localization. In this section, we evaluate the contribution of each module through an ablation study. Considering the trade-off among cost, time, and performance, we report the results only for CogniGent (Devstral). Table 5 reports the contribution of each module towards our technique’s performance.

From Table 5, we observe that all modules contribute positively to the localization precision (a.k.a., MAP) of CogniGent, though to varying degrees. At the document level, removing the investigation module (Step 8, Fig. 2) causes the largest drop in MAP to 0.360, which is an 11.57% decrease. Removing the filtering or hypothesis generation modules (Step 6 and 7, Fig. 2) also leads to moderate drops, reducing MAP to 0.364 and 0.379, respectively. This shows that both modules are important for selecting the right candidates and ranking them effectively. In comparison, the absence of the restructuring and observer modules (Step 4 and 9, Fig. 2) reduces CogniGent’s MAP scores to 0.382 and 0.392 (6.14% and 3.69% drops), indicating that although these modules contribute to the overall performance, their effects are comparatively smaller.

We see a similar pattern at the method level. Removing the investigation module causes the largest decrease in MAP to 0.183, a 19.02% drop. The filtering and hypothesis modules also remain important, reducing MAP by 16.37% and 13.71% to 0.189 and 0.195, respectively. The restructuring and observer modules make smaller but consistent contributions, with MAP values of 0.212 and 0.216.

Overall, these findings show that the hypothesis generation and investigation modules (Steps 7–8, Fig. 2), core parts of dynamic cognitive debugging, contribute substantially to bug localization by reasoning about hypotheses and gathering evidence across connected code segments. However, the ablation results also highlight

Table 5: Effect of Individual Components in CogniGent

Restructuring	Filtrering	Hypothesis	Investigation	Observation	MAP Document	MAP Method
✓	✓	✓	✓	✓	0.407	0.226
–	✓	✓	✓	✓	0.382	0.212
✓	–	✓	✓	✓	0.364	0.189
✓	✓	–	✓	✓	0.379	0.195
✓	✓	✓	–	✓	0.360	0.183
✓	✓	✓	✓	–	0.392	0.216

that the overall effectiveness of CogniGent relies on the collective contribution of all modules working together.

RQ2 Summary: CogniGent’s ablation study shows that hypothesis generation and hypothesis testing drive the largest gains (up to 19%). Smaller modules, such as restructuring and observation, still offer steady improvements, indicating that CogniGent’s effectiveness arises from the interplay among its modules.

Answering RQ₃ - Comparison with Baseline Technique. To position our work within the broader research landscape, we compare our technique with relevant baseline methods from prior studies. Table 6 compares the performance in terms of MAP, MRR, and HIT@K (K=1, 5) of our technique against five baselines from two categories: traditional IR-based techniques [62, 68] and LLM-based approaches [13, 69, 86]. For comparison, we use the CogniGent (Devstral) variant, which offers a balanced trade-off between performance, computational cost, and execution time (Table 4).

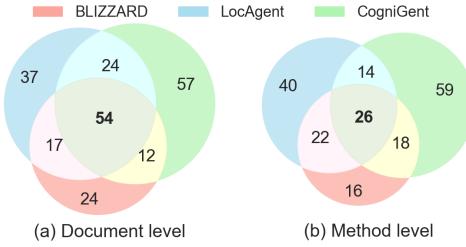
We compare our technique with traditional IR-based techniques: Baseline IR [36] and two techniques from the literature, BLIZZARD [62] and BLUiR [68]. To replicate the Baseline IR, we extract all methods and constructors from the source repository and index them using Apache Lucene [21], while keeping the default BM25 parameters (k and b) [66] and the preprocessing settings. During localization, we use bug reports (title + description) as queries to retrieve buggy code segments from the target system and version, as reported. To replicate BLIZZARD [62] and BLUiR [68], we follow a similar indexing policy and index the source code in Apache Lucene [21]. BLIZZARD classifies bug reports into three types and uses PageRank-based [9] text graph to generate tailored queries to retrieve the buggy code segments. On the other hand, BLUiR constructs structured queries from bug reports and source elements, performing eight separate searches over class names, method names, variable names, comments, and bug report fields (title and description). The results of these searches are then combined to produce an overall suspiciousness score for localizing buggy segments. We collected the BLUiR replication package from Bench4BL [44] and the BLIZZARD implementation from its replication package [44, 60], and adapted both for version-based retrieval.

We also compare CogniGent against three LLM-based techniques—BRaIn [69], Agentless [86] and LocAgent [13]. BRaIn is a state-of-the-art technique that improves bug localization through Intelligent Relevance Feedback. It captures contextual feedback from LLMs on initially retrieved documents, leverages them to enhance the query,

Table 6: Comparison of Baselines against CogniGent

Metrics	IR-based Techniques						LLM-based Techniques							
	Baseline-IR		BLUiR		BLIZZARD		BRaIn		Agentless		LocAgent		CogniGent	
	Doc.	Met.	Doc.	Met.	Doc.	Met.	Doc.	Met.	Doc.	Met.	Doc.	Met.	Doc.	Met.
MAP	0.330	0.163	0.344	0.191	0.380	0.177	0.375	0.206	0.332	0.177	0.384	0.207	0.407	0.226
MRR	0.334	0.165	0.350	0.196	0.387	0.183	0.395	0.223	0.352	0.176	0.392	0.224	0.418	0.254
HIT@1	0.254	0.122	0.257	0.139	0.321	0.127	0.343	0.187	0.324	0.137	0.353	0.197	0.374	0.227
HIT@5	0.443	0.217	0.453	0.259	0.446	0.234	0.464	0.267	0.448	0.217	0.465	0.265	0.475	0.272

Doc.: Document-level localization, Met.: Method-level localization.

**Figure 5: Coverage of Buggy Components Localized within Top-5 Across All Multi-Component Bugs**

and then employs Information Retrieval to localize bugs. For experiment, we collected the authors' replication package [7]. AgentLess is a recent technique that combines LLM-based reasoning with embedding-based retrieval to identify buggy source documents, which are then used for fixing corresponding issues. It builds a structured repository representation, retrieves semantically related documents using embeddings, and then uses the LLM to reason over their skeleton forms functions to refine and rank the most likely buggy locations. We adapt the replication package from the authors [56] and use the localization portion of the technique for the experiment. On the other hand, LocAgent is a graph-guided agentic framework for issue localization. It builds a heterogeneous code graph of a project and creates text indexes (e.g., BM25) over code entities to quickly retrieve those relevant to a bug report. The LLM agent then searches and traverses the graph, reasoning over the retrieved entities to produce a ranked list of suspicious documents and functions. We collected the replication packages from the authors [23] and adapt it for our dataset and version based localization and comparison.

Table 6 summarizes the comparison results between CogniGent and six baseline methods. Among traditional techniques, BLUiR achieves a MAP of 0.344, while BLIZZARD and the baseline-IR reach 0.379 and 0.330, respectively. CogniGent outperforms all of them with a MAP of 0.407, marking a up to 23.33% improvement, and achieves a maximum 25.14% gain in MRR (0.418). It also improves HIT@1 and HIT@5 to 0.374 and 0.475, yielding 16.31–47.33% and 5.00–7.25% gains against IR baselines in these metrics, respectively. At the method level, CogniGent achieves a MAP of 0.226 compared to 0.163, 0.191, and 0.177 for Baseline-IR, BLUiR, and BLIZZARD, showing 18.32–38.57% improvements. In MRR, HIT@1, and HIT@5, it also performs well, with gains of up to 53.74%, 86.11%, and 25.78%. These results demonstrate our technique's benefits over the traditional baseline techniques.

Table 7: Statistical Test: CogniGent vs. BLIZZARD and LocAgent

Evaluation Point	vs. BLIZZARD	vs. LocAgent
Top-1	0.0027 **, (Medium [†])	0.0180 *, (Small [†])
Top-5	0.0011 **, (Large [†])	0.0082 **, (Medium [†])

* = statistical significance, [†] = Effect size (Cliff's δ)

CogniGent also performs better than LLM-based techniques (Table 6), which capture relationships between bug reports and faulty code through reasoning. At the document level, CogniGent achieves a MAP score of 0.407, representing improvements of 8.53%, 22.59%, and 5.98% over BRaIn, AgentLess, and LocAgent, respectively. Similar gains are observed in MRR, HIT@1, and HIT@5, with improvements of up to 18.75%, 15.46%, and 6.20%. At the method level, CogniGent further surpasses these techniques, achieving MAP and MRR scores of 0.226 and 0.254—maximum gains of 31.39% and 43.50%, respectively. For HIT@1 and HIT@5, it records improvements of 65.60% and 25.58%. Overall, these results demonstrate CogniGent's effectiveness in ranking buggy documents closer at higher positions.

We further examine CogniGent's performance on multi-component bugs, where a single bug affects multiple documents or methods. Fig. 5 compares CogniGent with BLIZZARD and LocAgent, the closest competitors. In our dataset, we have 185 bug reports each of which triggers changes across multiple documents, totaling 523 buggy documents. Among these CogniGent localized 147 (28.10%) in the top-5 positions, outperforming LocAgent (132) and BLIZZARD (107). All three techniques commonly localized 54 documents, while CogniGent uniquely localized 57, compared to 37 by LocAgent and 24 by BLIZZARD. At the method level, 328 bug reports target 1,766 buggy methods in our dataset, where CogniGent localized 117 methods in the top-5, again outperforming LocAgent (102) and BLIZZARD (82). These results confirm that even at finer granularity, CogniGent provides consistently superior coverage of multi-component bugs.

To further validate these performance differences, we conducted non-parametric statistical tests - Wilcoxon signed-rank test [8] and measured effect size with Cliff's δ [8] using the ranks of first correct results from each three techniques above. As shown in Table 7, CogniGent demonstrates statistically significant improvements over BLIZZARD at both Top-1 and Top-5 positions ($p < 0.05$), with medium to large effect sizes. When compared to LocAgent, the results also show statistically significant improvements, with small to medium effect sizes. These findings confirm that CogniGent consistently localizes buggy components more effectively than both BLIZZARD and LocAgent.

RQ3 Summary: CogniGent outperforms both traditional and LLM-based baselines, achieving 23.33% and 38.57% higher MAP scores at the document and method levels, respectively. This suggests that dynamic cognitive debugging may have contributed to CogniGent localizing buggy documents closer to the top, with observed improvements of 2.36–86.11% across metrics compared to techniques that do not consider inter-component relationships. Statistical significance tests further confirm its superiority.

5 Related Work

5.1 IR and Bug Localization

Traditional bug localization methods fall into two categories: spectra-based and Information Retrieval (IR)-based [78]. Spectra-based approaches rely on execution traces and test cases, making them resource-intensive and often impractical [52, 77]. On the other hand, IR-based methods treat bug localization as a retrieval task and rely on the textual similarity between bug reports and source code to identify buggy documents.

Early IR-based techniques used the Vector Space Model (VSM) [43]. Later work refined these approaches with contextual cues such as bug resolution history, code changes, and version metadata [67, 71, 83]. Saha et al. [68] enhanced retrieval using structured features via the Indri engine [73], while BugLocator [36] combined rVSM scores with bug-fix history. AmaLgam and AmaLgam+ [79, 80] integrated multiple IR techniques (e.g., BLUiR, BugLocator) with stack traces and version or reporter history. Advanced models like LSI and LDA [36, 58] capture latent topics to address vocabulary mismatch; however, their performance remains comparable to simpler VSM-based approaches [44].

Since IR performance largely depends on query quality, some researchers focused on query reformulation—refining bug report queries by adding or removing terms [51, 61]. Refoqus [26] employed machine learning to suggest expansion or reduction strategies, while graph-based and genetic methods [61, 62] optimized queries by analyzing term relationships. Relevance feedback techniques, such as Rocchio’s algorithm [26] and Spatial Code Proximity (SCP) [72], further reformulated queries based on statistical and co-occurrence patterns from initial search results.

These traditional approaches rely on textual matching and statistical cues. While we also apply BM25-based retrieval (e.g., Lucene in Neo4j) to gather initial candidates, we perform multi-step LLM reasoning to capture symptom-level cues and align them with code behavior, moving beyond surface-level similarity.

5.2 Deep Learning and Bug Localization

Advances in deep learning and Large Language Models (LLMs) have improved the understanding of source code and natural language, motivating their application to bug localization. Existing approaches can be broadly categorized into training-based models and prompt-based instruction-following methods.

Among training-based techniques, DNNLOC [38], a seminal work, trains on positive and negative (bug report, source file) pairs

and incorporates multiple signals (e.g., rVSM similarity [36], class-name similarity, bug-fix recency), with a second network combining them for ranking. However, its dependence on bug-fixing recency can limit practical use. FBL-BERT [14] trains on bug-report and changeset pairs and follows ColBERT [35]-style late interaction for ranking, matching bug-report tokens to their most similar tokens in each changeset to identify buggy code. However, its reliance on changesets limits applicability in rapidly evolving projects. BLAZE [11] trains a GPT-based model using hard-example contrastive learning and dynamic chunking to improve alignment between bug reports and source code, enabling effective retrieval of buggy documents. Other approaches have employed convolutional neural networks to learn representations for bug localization [5, 31]. However, these techniques can lack generalizability and require retraining, limiting scalability in large, evolving codebases.

Prompt-based approaches leverage LLM reasoning to detect software bugs. BRaIn applies Intelligent Relevance Feedback, determining document relevance leveraging an LLM, making better search queries, and refining document retrieval through re-ranking. LocAgent [13] uses an agentic workflow where an LLM performs reasoning over a heterogeneous code graph. Guided by BM25-based retrieval and graph traversal, the agent iteratively hops across nodes (e.g., files, functions) through multiple edge types to identify likely buggy components. CoSIL [33] similarly performs graph-guided localization through a two-stage process: expanding the search space via module-level call graphs and refining to function-level candidates via iterative pruning.

In contrast, our technique leverages the pre-trained knowledge of LLMs while retrieving only a limited set of code segments via Lucene [21], improving scalability. Through code navigation guided by human-inspired causal reasoning and maintaining a focused context window, CogniGent gains deeper understanding of root causes and improves bug localization.

6 Threats to Validity

Threats to internal validity, which involve potential experimental errors or biases, primarily arise from replicating existing baselines. To address this, we relied on replication packages released by the original authors (e.g., BLIZZARD, BRaIn, Agentless, LocAgent [13, 60, 69, 86]) and from Bench4BL [44] (BLUiR [68]). Since Indri is now deprecated, we replaced it with Lucene in our BLUiR replication. To reduce bias, we tested on two datasets and found minimal differences from the baselines.

Threats to external validity pertain to the generalizability of our findings. Although CogniGent was evaluated solely on Java projects, the underlying language models (e.g., LLaMA [3]) are inherently adaptable to multiple programming languages, which helps reduce this limitation.

Threats to construct validity concern the suitability of the evaluation metrics. We adopted widely accepted measures—Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), and HIT@K—which are standard in both bug localization [62, 64, 68] and Information Retrieval research [27]. Using these established metrics helps ensure the reliability of our evaluation.

7 Conclusion

Software bugs undermine system reliability and security, leading to significant financial losses [81, 91]. To address these long-standing challenges, we introduce *CogniGent*, a technique designed to overcome the limitations of IR and AI-based bug localization by enabling causal reasoning, dependency-aware navigation, and context-managed code exploration. *CogniGent* applies human-inspired Dynamic Cognitive Debugging within an LLM-driven agentic workflow to formulate hypotheses, traverse interconnected components, and trace fault propagation to identify underlying causes. To evaluate *CogniGent*'s performance, we use three widely adopted metrics in bug localization—Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), and HIT@K—and observed substantial improvements: 23.33% and 38.57% gains in MAP at the document and method levels, respectively, and 25.14% and 53.74% gains in MRR across the same levels of granularity.

Building on this, we plan to capture finer-grained structural relationships and extend our agentic workflow to better assist developers in locating and resolving bugs.

References

- [1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653* (2020).
- [2] All Hands AI and Mistral AI. [n. d.]. DevStral: A New State-of-the-Art Open Model for Coding Agents. <https://www.all-hands.dev/blog/devstral-a-new-state-of-the-art-open-model-for-coding-agents>.
- [3] Meta AI. [n. d.]. The Llama 3 Herd of Models. ([n. d.]). <https://arxiv.org/abs/2407.21783>
- [4] Abdulaziz Alaboudi and Thomas D LaToza. 2023. What constitutes debugging? An exploratory study of debugging episodes. *Empirical Software Engineering* 28, 5 (2023), 117.
- [5] Waqas Ali, Lili Bo, Xiaobing Sun, Xiaoxue Wu, Saifullah Memon, Saima Siraj, and Ann Suwaree Ashton. 2023. Automated software bug localization enabled by meta-heuristic-based convolutional neural network and improved deep neural network. *Expert Systems with Applications* 232 (2023), 120562. doi:10.1016/j.eswa.2023.120562
- [6] Anonymous. [n. d.]. CogniGent Replication Package. <https://shorturl.at/Kv6r5>.
- [7] Mohammad Masudur Rahman Asif Samir. 2024. BRAIn: Replication Package. <https://github.com/asifsamir/BRaIn>.
- [8] Tyler Barnes, Scott C. Moore, and Katherine Osatuke. 2018. *Testing Significance Tests: A Simulation with Cliff's Delta, t-tests, and Mann-Whitney U*. National Center for Organizational Development, Department of Veteran Affairs.
- [9] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems* 30, 1-7 (1998), 107–117.
- [10] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. 2013. Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep* 229 (2013).
- [11] 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* 51, 8 (Aug. 2025), 2254–2267. doi:10.1109/tse.2025.3579574
- [12] Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus. 2017. Using observed behavior to reformulate queries during text retrieval-based bug localization. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSM)*. IEEE, 376–387.
- [13] Zhaoling Chen, Xiangru Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025. Locagent: Graph-guided ILM agents for code localization. *arXiv preprint arXiv:2503.09089* (2025).
- [14] 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.
- [15] Alibaba Cloud. [n. d.]. Qwen3-Coder: Large Mixture-of-Experts Model for Agentic Coding. <https://github.com/QwenLM/Qwen3-Coder>.
- [16] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press, Cambridge, MA.
- [17] DevOps. 2024. Survey: Fixing Bugs Stealing Time from Development. <https://shorturl.at/Fj8sB>
- [18] Yangruibo Ding, Jinjun Peng, Marcus J. Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. 2024. SemCoder: Training Code Language Models with Comprehensive Semantics Reasoning. *arXiv:2406.01006 [cs.CL]* <https://arxiv.org/abs/2406.01006>
- [19] Federal Communications Commission. 2024. February 22, 2024 AT&T Mobility Network Outage Report and Findings. <https://www.benton.org/headlines-february-22-2024-att-mobility-network-outage-report-and-findings> Published by the Benton Institute for Broadband & Society. Details the cause and impact of a nationwide AT&T wireless service outage affecting over 125 million devices and blocking more than 92 million calls..
- [20] Christoph Fischer. 2019. Javaparser. <https://github.com/javaparser/javaparser>
- [21] The Apache Software Foundation. 2021. Apache Lucene. (2021). <https://lucene.apache.org/>
- [22] George W. Furnas, Thomas K. Landauer, Louis M. Gomez, and Susan T. Dumais. 1987. The vocabulary problem in human-system communication. *Commun. ACM* 30, 11 (1987), 964–971.
- [23] Gerstein Lab. [n. d.]. LocAgent Replication Package. <https://github.com/gersteinlab/LocAgent>.
- [24] GitHub. 2024. Towards Natural Language Semantic Code Search. <https://github.blog/ai-and-ml/machine-learning/towards-natural-language-semantic-code-search/> Accessed: 2025-09-25.
- [25] Greptile. 2023. Semantic Codebase Search: Why It's Hard and What Works. <https://www.greptile.com/blog/semantic-codebase-search> Accessed: 2025-09-25.
- [26] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. 2013. Automatic query reformulations for text retrieval in software engineering. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 842–851.
- [27] Donna Harman. 2011. *Information retrieval evaluation*. Morgan & Claypool Publishers.
- [28] Kim Herzig, Sascha Just, and Andreas Zeller. 2013. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *2013 35th International Conference on Software Engineering (ICSE)*, 392–401. doi:10.1109/ICSE.2013.6606585
- [29] Kelly Hong, Anton Troynikov, and Jeff Huber. 2025. *Context Rot: How Increasing Input Tokens Impacts LLM Performance*. Technical Report. Chroma. <https://research.trychroma.com/context-rot>
- [30] Danniell Hu, Priscila Santiesteban, Madeline Endres, and Westley Weimer. 2024. Towards a Cognitive Model of Dynamic Debugging: Does Identifier Construction Matter? *IEEE Transactions on Software Engineering* (2024).
- [31] Xuan Huo and Ming Li. 2017. Enhancing the Unified Features to Locate Buggy Files by Exploiting the Sequential Nature of Source Code.. In *IJCAI* 1909–1915.
- [32] Yerin Hwang, Yongil Kim, Jahyun Koo, Taegwan Kang, Hyunkyung Bae, and Kyomin Jung. 2025. Llms can be easily confused by instructional distractions. *arXiv preprint arXiv:2502.04362* (2025).
- [33] Zhonghao Jiang, Xiaoxun Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. 2025. CoSIL: Software Issue Localization via LLM-Driven Code Repository Graph Searching. *arXiv preprint arXiv:2503.22424* (2025).
- [34] Carlos E Jimenez, John Yang, Alexander Wetlig, 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).
- [35] Omar Khattab and Matei Zaharia. 2020. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*, 39–48.
- [36] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. 2013. Where should we fix this bug? a two-phase recommendation model. *IEEE transactions on software Engineering* 39, 11 (2013), 1597–1610.
- [37] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2017. Bug localization with combination of deep learning and information retrieval. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 218–229.
- [38] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2017. Bug Localization with Combination of Deep Learning and Information Retrieval. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 218–229. doi:10.1109/ICPC.2017.24
- [39] LangChain. [n. d.]. Agent Scratchpad. https://langchain-ai.github.io/langgraph/concepts/multi_agent/
- [40] LangChain. [n. d.]. LangGraph: A Library for Building Stateful Multi-Agent Workflows. <https://github.com/langchain-ai/langgraph>.
- [41] LangChain, Inc. [n. d.]. Low-level Concepts – LangGraph. https://langchain-ai.github.io/langgraph/concepts/low_level/.
- [42] Tien-Duy B Le, Richard J Oentaryo, and David Lo. 2015. Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 579–590.
- [43] DL Lee, Huei Chuang, and K. Seamons. 1997. Document ranking and the vector-space model. *IEEE Software* 14, 2 (1997), 67–75. doi:10.1109/52.582976
- [44] Jaekwon Lee, Dongsun Kim, Tegawendé F Bissyandé, Woosung Jung, and Yves Le Traon. 2018. Bench4bl: reproducibility study on the performance of ir-based bug localization. In *Proceedings of the 27th ACM SIGSOFT international symposium*

- on software testing and analysis.* 61–72.
- [45] J. C. S. do Prado Leite and G. H. Travassos. 2023. *Cognitive Debugging*. Springer. <https://link.springer.com/book/10.1007/978-3-031-42064-1> Accessed: 2025-09-25.
- [46] Mosh Levy, Alon Jacoby, and Yoav Goldberg. 2024. Same Task, More Tokens: the Impact of Input Length on the Reasoning Performance of Large Language Models. arXiv:2402.14848 [cs.CL] <https://arxiv.org/abs/2402.14848>
- [47] Yuanyuan Liang, Jianing Wang, Hanlun Zhu, Lei Wang, Weinig Qian, and Yunshi Lan. 2023. Prompting large language models with chain-of-thought for few-shot knowledge base question generation. *arXiv preprint arXiv:2310.08395* (2023).
- [48] Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Shieh, and Wenmeng Zhou. 2024. CodexGraph: Bridging Large Language Models and Code Repositories via Code Graph Databases. (8 2024). <http://arxiv.org/abs/2408.03910>
- [49] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting Coverage-Based Fault Localization via Graph-Based Representation Learning. In *ESEC/FSE*. doi:10.1145/3468264.3468580
- [50] Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, Baolong Bi, Yujun Cai, Jiazhi Liu, Mingyu Li, Zhong-Zhi Li, Duzhen Zhang, et al. 2025. A survey of context engineering for large language models. *arXiv preprint arXiv:2507.13334* (2025).
- [51] Chris Mills, Esteban Parra, Jevgenija Pantliuchina, Gabriele Bavoata, and Sonia Haiduc. 2020. On the relationship between bug reports and queries for text retrieval-based bug localization. *Empirical Software Engineering* 25 (2020), 3086–3127.
- [52] Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. 2014. On the Use of Stack Traces to Improve Text Retrieval-Based Bug Localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 151–160. doi:10.1109/ICSME.2014.37
- [53] Inc. Neo4j. 2025. Neo4j Graph Database. <https://neo4j.com>.
- [54] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N Nguyen. 2011. A topic-based approach for narrowing the search space of buggy files from a bug report. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 263–272.
- [55] Devon H O'Dell. 2017. The Debugging Mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills. *Queue* 15, 1 (2017), 71–90.
- [56] OpenAutoCoder. 2024. Agentless Replication Package. <https://github.com/OpenAutoCoder/Agentless>.
- [57] Augustin Popa. 2017. *Productivity Improvements: Ctrl + Click Go to Definition*. <https://devblogs.microsoft.com/cppblog/productivity-structure-visualizer-crtl-click-to-go-to-definition/> Microsoft Dev Blogs.
- [58] Denys Poshyvanyk, Yann-Gal Guneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. 2007. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering* 33, 6 (2007), 420–432.
- [59] Binhang Qi, Hailong Sun, Wei Yuan, Hongyu Zhang, and Xiangxin Meng. 2022. DreamLoc: A Deep Relevance Matching-Based Framework for bug Localization. *IEEE Transactions on Reliability* 71, 1 (2022), 235–249. doi:10.1109/TR.2021.3104728
- [60] Masud Rahman. [n.d.] BLIZZARD. <https://github.com/masud-technope/BLIZZARD>
- [61] Mohammad Masudur Rahman, Foutse Khomh, Shamima Yesamin, and Chanchal K Roy. 2021. The forgotten role of search queries in ir-based bug localization: an empirical study. *Empirical Software Engineering* 26, 6 (2021), 116.
- [62] 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.
- [63] Mohammad Masudur Rahman and Chanchal K Roy. 2021. A Systematic Review of Automated Query Reformulations in Source Code Search. *ACM Transactions on Software Engineering and Methodology* (2021).
- [64] Mohammad Masudur Rahman, Chanchal K. Roy, and David Lo. 2016. RACK: Automatic API Recommendation Using Crowdsourced Knowledge. Institute of Electrical and Electronics Engineers (IEEE), 349–359. doi:10.1109/saner.2016.80
- [65] Matthew Renze. 2024. The effect of sampling temperature on problem solving in large language models. In *Findings of the association for computational linguistics: EMNLP 2024*. 7346–7356.
- [66] Stephen E Robertson, Steve Walker, Susan Jones, Micheline M Hancock-Beaulieu, Mike Gatford, et al. 1995. Okapi at TREC-3. *Nist Special Publication Sp 109* (1995), 109.
- [67] Riton K Saha, Julia Lawall, Sarfraz Khurshid, and Dewayne E Perry. 2014. On the effectiveness of information retrieval based bug localization for c programs. In *2014 IEEE international conference on software maintenance and evolution*. IEEE, 161–170.
- [68] Riton K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. 2013. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 345–355.
- [69] Asif Mohammed Samir and Mohammad Masudur Rahman. 2025. Improved IR-based Bug Localization with Intelligent Relevance Feedback. *arXiv preprint arXiv:2501.10542* (2025).
- [70] Asif Mohammed Samir and Mohammad Masudur Rahman. 2025. Improving IR-based Bug Localization with Semantics-Driven Query Reduction. *arXiv:2510.04468* [cs.SE] <https://arxiv.org/abs/2510.04468>
- [71] Bunyamin Sisman and Avinash C Kak. 2012. Incorporating version histories in information retrieval based bug localization. In *2012 9th IEEE working conference on mining software repositories (MSR)*. IEEE, 50–59.
- [72] Bunyamin Sisman and Avinash C. Kak. 2013. Assisting code search with automatic Query Reformulation for bug localization. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. 309–318. doi:10.1109/MSR.2013.6624044
- [73] Trevor Strohman, Donald Metzler, Howard R. Turtle, and W. Bruce Croft. 2005. Indri : A language-model based search engine for complex queries (extended version). <https://api.semanticscholar.org/CorpusID:18471028>
- [74] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 1433–1443.
- [75] Yijun Tian, Lijun Mei, Yue Yu, Shaowei Wang, Haoyu Wang, and Yang Liu. 2023. Structural-Guided Attention for Bug Localization. In *IJCAI*. doi:10.24963/ijcai.2023/461
- [76] Tricentis. 2025. *2025 Quality Transformation Report*. Technical Report. Tricentis. <https://www.tricentis.com/resources/quality-transformation-report> Accessed: 2025-09-25.
- [77] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the usefulness of IR-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) (ISSTA 2015). Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/2771783.2771797
- [78] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the usefulness of IR-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 1–11.
- [79] Shaowei Wang and David Lo. 2014. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*. 53–63.
- [80] 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.
- [81] Lian Kit Wee. 2024. Here comes the wave of insurance claims for the CrowdStrike outage. <https://shorturl.at/5Y1Q>
- [82] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.
- [83] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating bugs from software changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 262–273.
- [84] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. In *Proceedings of the 30th Conference on Pattern Languages of Programs* (Monticello, IL, USA) (PLoP '23). The Hillside Group, USA, Article 5, 31 pages.
- [85] Siyu Wu, Jian Xie, Jiangjin Chen, Tinghui Zhu, Kai Zhang, and Yanghua Xiao. 2024. How easily do irrelevant inputs skew the responses of large language models? *arXiv preprint arXiv:2404.03302* (2024).
- [86] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489* (2024).
- [87] Shaochun Xu and Vaclav Rajlich. 2004. Cognitive process during program debugging. In *Proceedings of the Third IEEE International Conference on Cognitive Informatics*, 2004. IEEE, 176–182.
- [88] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izahk Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.
- [89] Byung-doo Yoon and Oscar N Garcia. 1998. Cognitive activities and support in debugging. In *Proceedings fourth annual symposium on human interaction with complex systems*. IEEE, 160–169.
- [90] Klaus Changsun Youm, June Ahn, Jeongho Kim, and Eunseok Lee. 2015. Bug Localization Based on Code Change Histories and Bug Reports. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*. 190–197. doi:10.1109/APSEC.2015.23
- [91] Morgan Zahn, Jon Haworth, Josh Margolin, Jack Date, and Luke Barr. 2024. AT&T outage caused by software update, company says. <https://abcnews.go.com/US/att-outage-impacting-us-customers-company/story?id=107440297> Accessed: 2025-09-22.
- [92] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM*

- SIGSOFT International Symposium on Software Testing and Analysis.* 1592–1604.
- [93] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. In *The eleventh international conference on learning representations*.
- [94] Weiqin Zou, David Lo, Zhenyu Chen, Xin Xia, Yang Feng, and Baowen Xu. 2018. How practitioners perceive automated bug report management techniques. *IEEE Transactions on Software Engineering* 46, 8 (2018), 836–862.