CodeScout: Contextual Problem Statement Enhancement for Software Agents

Anonymous ACL submission

Abstract

Current AI-powered code assistance tools often struggle with ambiguous problem statements that lack sufficient task context and requirements specification. Recent analysis of software engineering agents reveals that failures on such ambiguous requests are highly correlated with longer trajectories involving either over-exploration or repeated attempts at applying the same fix without proper evolution or testing, leading to suboptimal outcomes across software development tasks. We introduce CodeScout, a breakthrough contextual query refinement approach that systematically converts ambiguous user requests into comprehensive, actionable problem statements through lightweight pre-exploration of the target codebase. Our key innovation is demonstrating that structured analysis before task execution can supplement existing agentic capabilities without requiring any modifications to their underlying scaffolds. CodeScout performs targeted context scoping, conducts multi-perspective analysis examining potential fixes and exploration opportunities, then synthesizes these insights into enhanced problem statements with reproduction steps, expected behaviors, and targeted exploration hints. This pre-exploration directly addresses the identified failure patterns by reducing non-converging agent trajectories while clarifying user intent in natural language space. We evaluate CodeScout using state-of-the-art agentic scaffolds and language models on SWEBench-Verified, demonstrating a 20% improvement in resolution rates with up to 27 additional issues resolved compared to the default baseline method. Our results suggest that systematic query refinement through contextual analysis represents a promising direction for enhancing AI code assistance capabilities.

1 Introduction

The rapid advancement of large language models (LLMs) has revolutionized software development assistance, with AI-powered coding tools becoming increasingly integral to modern workflows (Kumar et al., 2025; Jiang et al., 2023). Yet a fundamental challenge persists: these systems often fail not because the underlying models lack reasoning capability, but because they are asked to operate on poorly specified problem statements (Meng et al., 2024). Developers frequently provide concise, context-dependent descriptions—omitting reproduction steps, technical details, or clear expectations—that assume a shared understanding of the codebase. In contrast, LLM-based agents work optimally with explicit, well-scoped specifications to reason effectively (Nam et al., 2025).

Empirical studies validate this phenomenon: Meng et al. (2024) found that resolvable bug reports exhibit drastically higher description quality scores, with relative differences ranging from 110% to over 2700% compared to non-resolvable ones. Similarly, Nam et al. (2025) showed that rejected code suggestions in Google's LLM-integrated development tools strongly correlate with low-quality user inputs. Recent analysis by Bouzenia and Pradel (2025) reveals that software engineering agent failures exhibit specific behavioral patterns: over-exploration where agents fail to reach the root of the problem due to context overload, and repeated application of the same fix without proper testing or evolution, demonstrating stubborn agentic behavior and lack of understanding of code or user intent. Together, these findings highlight that input quality—not just model capacity—forms the critical bottleneck in AI-assisted software engineering.

This specification gap is exacerbated by how current agents operate, but why do current agentic workflows struggle with this challenge? We hypothesize this occurs for several fundamental reasons. First, agents plan and implement simultaneously at a step-by-step granularity through reason-execute-observe loops, lacking the long-horizon

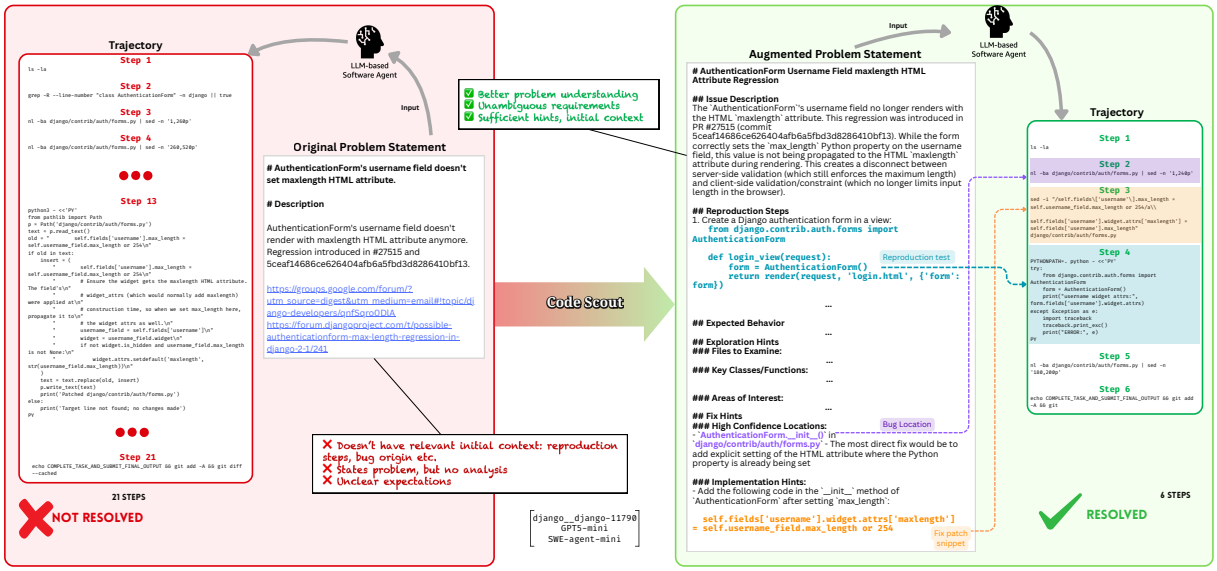


Figure 1: The original SWEBench problem statement for Instance `django__django-11790` lack relevant initial context. As a result, the downstream agent is not able to fix the issue, despite spending 21 steps exploring the repository, analyzing code, iterating the fix patch. In contrast, the enhanced problem statement generated by our approach resolves the issue in 6 agentic steps, as it includes relevant insights that can be used off-the-bat by the agent.

understanding necessary for complex debugging tasks and accumulating deviations from the true problem scope. Second, without comprehensive upfront context, agents become trapped in reactive exploration, facing information asymmetry where they must discover codebase structure incrementally while lacking the hierarchical abstraction capabilities to build strategic mental models. Figure 1 exemplifies this: a sparse Django bug report leads to 21 steps of unfocused exploration and failure, while an enhanced problem statement enables success in just 6 targeted steps.

This observation motivates a shift in perspective: instead of expecting agents to "leap" directly into problem-solving, we argue they must first look—building comprehensive understanding before attempting fixes. We introduce CodeScout, a contextual problem statement enhancement method, that transforms vague developer queries into rich, repository-aware actionable problem statements. Compared to traditional localization or context retrieval methods that merely identify relevant code, CodeScout provides detailed exploration plans, task decomposition, fix hints, and strategic guidance. Key advantages include: supplementing existing agents without requiring scaffold modifications, enabling long-horizon strategic alignment, reducing non-converging trajectories, and working universally across agent architectures. **Main contributions**:

1. CodeScout, a systematic approach for contextual problem specification enhancement that improves input quality through repository-aware analysis, demonstrating 20% improvement in resolution rates with up to 27 additional issues resolved.

2. Empirical validation demonstrating CodeScout's effectiveness across multiple software engineering tasks and agent architectures, showing consistent performance gains.

3. Detailed analysis of how CodeScout's contextual augmentation influences agent trajectories, tool usage, and problem-solving strategies, revealing cross-validation performance gains that generalize across different agent implementations. We demonstrate compelling cost-efficiency dynamics: smaller models can effectively perform problem statement enhancement to improve larger model performance, while strong models can augment problem statements to dramatically boost weaker model capabilities.

Our results suggest that investing computation in upfront problem understanding offers a powerful complement to advances in model capacity and agentic reasoning, pointing toward more reliable AI-assisted software engineering.

2 Related Work

LLM-based Software Agents

LLM-based agents (Wang and Xu, 2024; Romeo et al., 2025) have emerged as autonomous systems that use large language models as reasoning engines to perform complex software engineering tasks through planning, tool usage, and iterative problem-solving. These agents substantially extend the versatility of LLMs by enhancing them with capabilities for perceiving and utilizing external resources and tools, providing autonomy through key components including planning, memory, perception, and action (Liu et al., 2024). The sophisticated capabilities demonstrated by these agents directly motivated our work: while they excel at execution and reasoning, their performance degrades significantly when operating on ambiguous or poorly specified problem statements—a fundamental limitation that CodeScout addresses.

The evolution from traditional LLM applications to autonomous agents represents a significant shift in software engineering automation. While earlier approaches typically operated through single prompts or hard-coded feedback loops for tasks like code completion (Ziegler et al., 2022; Barke et al., 2023), automated program repair (Jiang et al., 2023), and test generation (Lemieux et al., 2023; Ryan et al., 2024; Yuan et al., 2024), modern LLM agents (Bouzenia and Pradel, 2025) can autonomously plan and execute sequences of actions while adapting based on feedback from tools and environments.

Several prominent agent frameworks have demonstrated effectiveness across software engineering tasks, but also revealed the input quality bottleneck that CodeScout targets. RepairAgent (Bouzenia et al., 2025) treats the LLM as an autonomous agent capable of planning and executing actions to fix bugs by invoking suitable tools, freely interleaving information gathering, repair ingredient collection, and fix validation. AutoCodeRover (Zhang et al., 2024) combines LLMs with sophisticated code search capabilities, working on program representations like abstract syntax trees rather than viewing projects as mere file collections. SWE-agent (Yang et al., 2024) introduces custom agent-computer interfaces that significantly enhance agents' abilities to create, edit, navigate repositories, and execute tests.

However, these agents share a critical limitation: they cannot compensate for fundamentally ambiguous or under-specified input problems. Unlike CodeScout, which proactively enhances problem understanding before agent execution, existing agents rely on reactive exploration that often leads to the over-exploration and repeated fix application patterns identified by recent analysis (Bouzenia and Pradel, 2025).

Code Query Understanding and Refinement

While LLM-based software agents demonstrate sophisticated capabilities for repository-level interactions, existing approaches to code query enhancement have primarily focused on traditional information retrieval techniques without addressing the semantic understanding of complex problem statements at the repository level. This gap in repository-aware problem understanding directly shaped the development of CodeScout's contextual enhancement approach.

Traditional query expansion approaches have relied on lexical methods to address the vocabulary mismatch problem between queries and code. Lu et al. (Lu et al., 2015) proposed extending queries with synonyms generated from WordNet to match natural language phrases extracted from source code identifiers. Nie et al. (Nie et al., 2016) introduced Query Expansion based on Crowd Knowledge (QECK), which identifies software-specific expansion words from Stack Overflow question-answer pairs to automatically generate expansion queries, improving precision by up to 64%. While these approaches improve keyword matching, they cannot handle the complex, repository-contextual problem statements that CodeScout addresses.

Recent work has explored LLM-based data augmentation for code search. Wang et al. (Wang et al., 2023) proposed ChatDANCE, which utilizes ChatGPT to generate high-quality augmented code-query pairs with a filtering mechanism, achieving 13.2% improvement in R@1. However, this approach focuses on augmenting training data for code search rather than understanding complex problem specifications in live development contexts.

In program repair contexts, some work has explored semantic augmentation of prompts. Ahmed et al. (Ahmed et al., 2024) demonstrated that adding semantic facts (parameter names, control flow, etc.) to LLM prompts improves code summarization performance, surpassing 30 BLEU1 on PHP in CodeSearchNet. Jin et al. (Jin et al., 2023) proposed InferFix, which combines retrieval of semantically similar bug-fix pairs with LLM-based

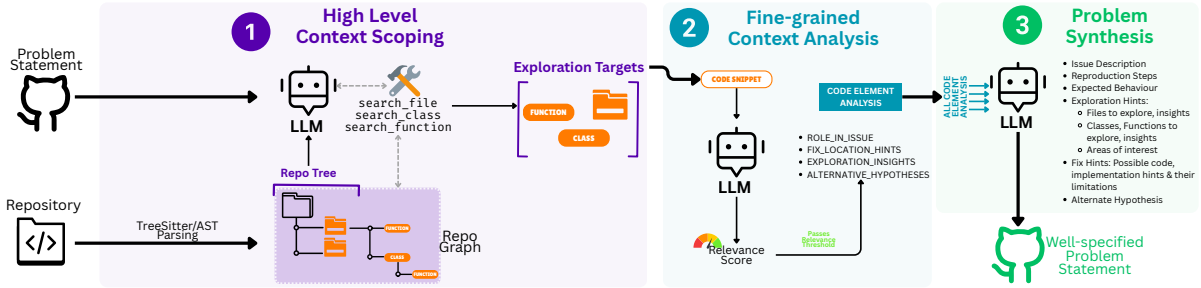


Figure 2: **CodeScout:** The pre-exploration with Repository Knowledge Graph Construction, which represents code structure and relationships. Building on this, three main stages follow: 1) *High Level Scoping*, where an LLM agent identifies relevant exploration targets, 2) *Fine-grained Context Analysis*, which extracts structured insights for each target, and 3) *Problem Synthesis*, where the original problem statement is combined with filtered insights to generate the augmented specification.

generation, achieving 76.8% accuracy in Java program repair. These approaches inform our semantic analysis component but lack the comprehensive repository-aware problem statement synthesis that CodeScout provides. The fundamental challenge that existing approaches cannot solve—and that motivated CodeScout's design—is transforming vague, context-dependent developer descriptions into comprehensive, actionable problem statements that enable strategic rather than reactive agent behavior through repository-level understanding with cross-file dependencies, architectural context, and hierarchical refinement that mirrors human developer comprehension.

3 Task Definition

We formalize the problem statement augmentation task as a preprocessing transformation that enhances task specifications for downstream software agents. Given an initial problem statement P_0, software repository $\mathcal{R}$, and task-specific agent $\mathcal{A}$, our objective is to learn a transformation function $\mathcal{T}$ such that:

$$M(\mathcal{A}(P_{aug}, \mathcal{R})) \geq M(\mathcal{A}(P_0, \mathcal{R}))$$

where $P_{aug} = P_0 \oplus \mathcal{T}(P_0, \mathcal{R})$ represents the augmented specification that builds upon the original problem statement through contextual enhancement, $\oplus$ denotes the augmentation operation, and $M(\cdot)$ is the task metric. The transformation operates as a plug-and-play preprocessing step that enriches original problem statements with contextual information extracted through systematic codebase analysis. Our approach implements a preprocessing paradigm where the augmentation process is decoupled from downstream agent architectures, maintaining interface compatibility while providing enhanced agent inputs.

4 CodeScout

CodeScout employs a three-stage pipeline (Figure 2) that systematically transforms rudimentary problem statements into comprehensive problem specifications. The methodology leverages a Large Language Model (LLM) based workflow to analyze repository structure, extract relevant code context, and synthesize actionable debugging guidance.

The pipeline operates as follows: (1) Repository analysis constructs a knowledge graph $G(\mathcal{R})$ representing code structure and relationships, (2) Context scoping uses an LLM agent to identify exploration targets $T = \{t_1, t_2, ..., t_n\}$ from $G(\mathcal{R})$ and P_0, (3) Code analysis extracts insights $I = \{i_1, i_2, ..., i_m\}$ for each target, and (4) Specification synthesis combines P_0 and I to produce P_{aug}.

4.1 Repository Knowledge Graph Construction

We construct a directed graph $G(\mathcal{R}) = (V, E)$ where vertices V represent code entities and edges E capture semantic relationships. Each vertex $v \in V$ is characterized by $v = (name, type, location, metadata)$.

We employ an Abstract Syntax Tree (AST) visitor that traverses parsed syntax trees to extract class definitions with inheritance relationships, function signatures with parameter specifications, import dependencies and module relationships, and variable declarations and scope information.

4.2 High Level Scoping

Given problem statement P_0 and repository structure representation $S(\mathcal{R})$ derived from $G(\mathcal{R})$, an LLM agent generates exploration targets using

$$T = LLM_{scope}(P_0, S(\mathcal{R})) \quad (1)$$

The scoping prompt instructs the LLM to identify code entities most likely relevant to the reported issue. The agent analyzes direct entity mentions in P_0, applies common debugging patterns from its training knowledge, examines structural relationships within $G(\mathcal{R})$, and considers entity naming conventions that might relate to the issue description, all while using the file tree as contextual guidance for understanding repository organization.

The repository structure representation $S(\mathcal{R})$ presents hierarchical organization with semantic annotations showing both file organization and code structure, including the complete file tree layout. During this scoping phase, the actual retrieval of relevant entities occurs directly from this repository-graph, enabling efficient identification of potential targets without requiring complete source code access.

Exploration targets are constrained to $|T| \leq 15$ to balance comprehensive coverage with computational feasibility. Each target $t_i = (type_i, name_i, reasoning_i)$ includes justification, explaining relevance to P_0.

4.3 Fine-grained Context Analysis

For each exploration target $t_i \in T$, we retrieve corresponding code content c_i from $\mathcal{R}$ using $G(\mathcal{R})$ for efficient lookup.

Each code element undergoes structured analysis:

$$i_i = LLM_{analyze}(P_0, c_i, t_i) \quad (2)$$

The analysis prompt $\pi_{analyze}$ extracts specific insight categories: 1) Role Assessment describing how c_i relates to the reported issue, 2) Fix Location Hints identifying potential modification points with confidence estimates, 3) Technical Insights capturing implementation patterns and architectural decisions, and Alternative Hypotheses proposing different root cause explanations.

We implement relevance-based filtering, retaining insights where $score_{rel}(i_j) \geq \tau_{rel}$, where τ_{rel} is the relevance threshold.

4.4 Problem Synthesis

The final synthesis stage combines original problem statement P_0 with filtered insights $I_{filtered}$ to generate the augmented specification:

$$P_{aug} = LLM_{synthesize}(P_0, I_{filtered}) \quad (3)$$

The synthesis prompt structures the output as P_{aug}, which contains an enhanced description, reproduction steps, expected behavior specification, exploration hints, and fix guidance, where each component serves specific debugging functions.

Enhanced Issue Description integrates technical insights from the filtered analysis to clarify problem mechanisms and scope. Reproduction Steps augment original reproduction procedures with internal technical details and error patterns discovered during code analysis. Exploration Hints map directly from analysis insights to provide structured guidance. Each hint includes specific reasoning derived from role analysis and technical insights components of relevant code elements. Fix Hints synthesize fix location suggestions with confidence estimates. High-confidence locations are derived from the fix hint analysis with associated confidence scores, while alternative hypotheses aggregate insights across analyzed elements.

The synthesis process maintains traceability by preserving relationships between generated guidance and source code analysis. The resulting augmented specification P_{aug} serves as a comprehensive debugging guide that significantly reduces initial investigation overhead compared to the original problem statement P_0.

5 Experimental Setup

We evaluate our approach on the SWEBench-Verified benchmark (Jimenez et al., 2023), a widely used dataset software engineering agents on real-world issue-solving tasks.

Our experiments consider three scaffolds: SWE-agent (Yang et al., 2024), OpenHands (Wang et al., 2025), and Mini-SWE-Agent. Each scaffold interfaces with different Large Language Models (LLMs), specifically GPT-5-mini, DeepSeek R1, and Qwen3 Coder 480B.

For **SWE-agent**, we enforce cost limits of 1.0, 1.0, and 0.75 USD per model, respectively. For **OpenHands**, we restrict execution to a maximum of 80 steps. All other configurations for both SWE-agent and OpenHands, as well as for **Mini-SWE-Agent**, follow their default settings.

6 Results

6.1 Quantitative gains and localization

We evaluate the effect of contextual problem-statement augmentation across three agentic

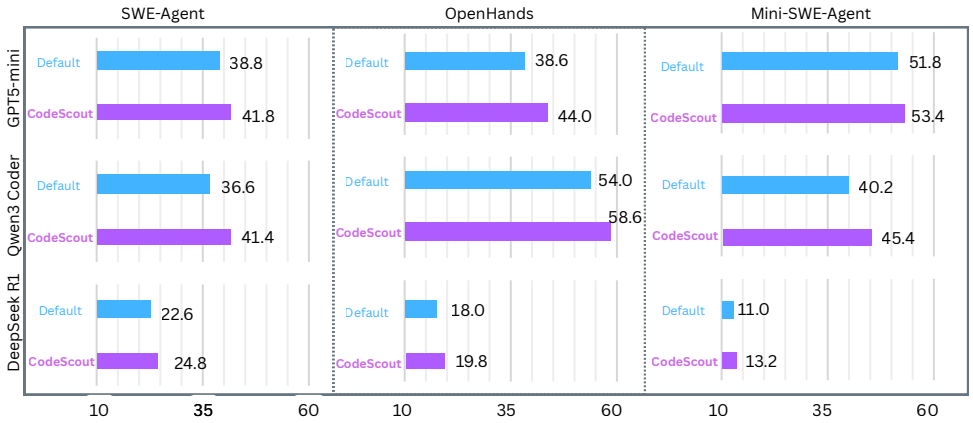


Figure 3: Main comparison between default (no augmentation) and our contextual augmentation across three LLMs (GPT-5-mini, Qwen3 Coder, DeepSeek R1) and three scaffolds (SWE-Agent, OpenHands, Mini-SWE-Agent). Augmentation yields consistent improvements in resolution rate; gains are largest when the runtime LLM has weaker agentic abilities.

scaffolds (SWE-Agent, OpenHands, Mini-SWE-Agent) and three LLM families (DeepSeek R1, Qwen3 Coder, GPT-5-mini). Across the full SWEBench-Verified evaluation the augmentation consistently improves the agent resolution rate: the gains are broadly consistent across scaffolds and models and are most pronounced when the runtime agent is relatively weak. Figure 3 summarizes the aggregate performance gap between the default (no augmentation) and CodeScout across all tested combinations.

On the SWE-Agent subset used for ablation (Table 1) our full workflow increases the number of resolved issues compared to the Default baseline by absolute counts of {+11, +15, +24} for {DeepSeek R1, GPT-5-mini, Qwen3 Coder} respectively, corresponding to relative increases of approximately {9.6%, 7.7%, 13.1%}. These numbers underscore that the augmentation pipeline consistently converts ambiguous problem statements into more actionable specifications that downstream agents can act upon.

We also evaluate *localization* (whether the generated patch touches the same file or function as the ground-truth patch). At both file- and function-level granularity the augmented setting improves localization accuracy over the default across models. The improvement is particularly large for DeepSeek R1 (which lacks strong agentic reasoning out-of-the-box), indicating that workflow-driven augmentation helps weaker agents find the correct code regions. For GPT-5-mini, however, static localization metrics underestimate actual runtime localization: GPT-5-mini often produces patches that are applied at runtime (dynamic edits or runtime instrumentation), so static file-level matches do not fully capture its localization ability.

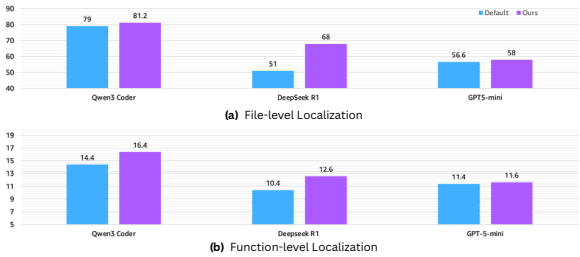


Figure 4: Localization comparison (file-level and function-level) using SWE-Agent as the scaffold.

6.2 Ablation Study

We performed ablations with SWE-Agent as the base scaffold to isolate which components of the workflow drive the gains (Table 1). Key observations:

Does the full pipeline improve performance? Yes, consistently across all models: DeepSeek R1 +11 resolved (+9.6%), GPT-5-mini +15 (+7.7%), Qwen3 Coder +24 (+13.1%).

Can agents autonomously perform augmentation? When we modify system prompts to request autonomous augmentation during execution, performance drops significantly below Default for all three models (DeepSeek R1: -5, GPT-5-mini: -17, Qwen3 Coder: -25). This validates that CodeScout's separate, structured augmentation stage addresses a real limitation—agents cannot effectively self-augment during their trajectory.

Is relevance filtering necessary? Yes. Removing the relevance filter (*CodeScout - without filtering*) reduces or eliminates much of the benefit (small or

Ablations/Baselines	DeepSeek R1	GPT-5-mini	Qwen3 Coder
Default	114	194	183
CodeScout (Ours)	**125**	**209**	**207**
Agentic Intra-trajectory Augmentation	109	177	158
CodeScout (No Filtering)	116	190	190
CodeScout (BM25 Entity Selection)	119	195	198

Table 1: Ablation results (SWE-Agent). Numbers are counts of resolved issues on SWEBench-Verified.

negative changes versus Default for some models), showing that careful filtering is crucial to avoid adding noisy context.

Does LLM scoping outperform retrieval? Yes. Replacing LLM-driven scoping with BM25 retrieval (Robertson et al., 2009)—a classical lexical matching algorithm that ranks documents by term frequency—still improves over Default but yields smaller gains than LLM scoping, particularly for Qwen3 Coder. This suggests LLM scoping captures semantic relationships beyond lexical overlap.

6.3 Cost and token analysis

Figure 5 and Figure 6 summarize token consumption and dollar cost trade-offs. **Tokens-per-resolved-issue improves for most models:** When comparing cumulative issues resolved at a given token budget (Fig. 5a), augmented runs resolve more issues for the same token budget for Qwen3 and DeepSeek. When we explicitly offset agent token consumption by the augmentation overhead (Fig. 5b), the gains remain comparable for Qwen3 and DeepSeek, showing that augmentation does not simply shift cost to a separate preprocessing step for these models. **Large-trajectory LLMs are an exception:** GPT-5-mini produces extremely long trajectories in our setup (order-of-magnitude more tokens per trajectory), so augmentation overhead is comparatively small and does not meaningfully change the tokens-per-resolved-issue metric. The net effect is that augmentation improves performance but the large absolute token budgets required by GPT-5-mini remain a deployment consideration.

6.4 Impact on problem statements

Augmentation increases the mean length of problem statements (Figure 7) but reduces noise in the long tail: default problem statements frequently include irrelevant large logs or environment dumps, while augmented statements are longer but more focused and actionable. The length-distribution plot shows a tighter, more concentrated augmented distribution and a reduced long tail relative to Default, indicating the augmentation mostly adds concentrated, high-value technical context rather than indiscriminate verbosity.

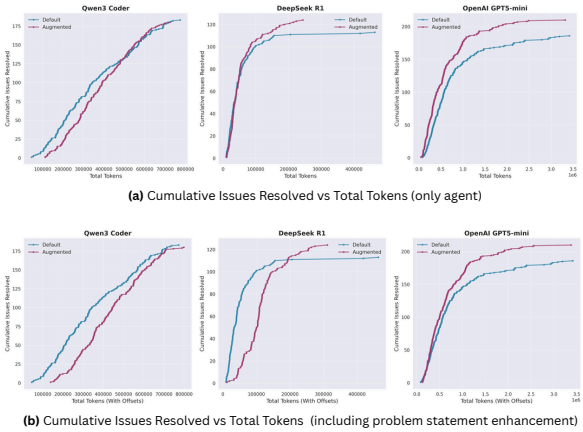


Figure 5: Cumulative issues resolved as a function of total tokens consumed (input+output). (a) Agent-only token accounting; (b) token accounting including augmentation overhead.

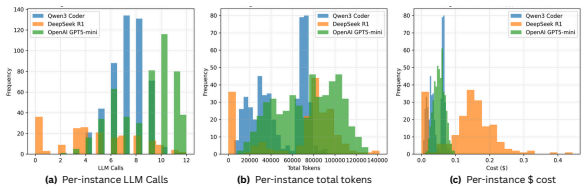


Figure 6: Augmentation metrics: number of LLM calls, tokens and dollar cost per instance.

6.5 Agent behavior and tool usage

Trajectory analysis (Figure 8) shows a systematic change in early-agent behavior under augmentation: the initial fraction of calls to `view` and `grep` rises and `find` falls, consistent with more targeted exploration. Later phases show expected increases in active repository actions (patching, creates, runs). These changes indicate that augmentation enables agents to begin trajectories with higher-quality search terms and file targets, reducing wasted broad exploration.

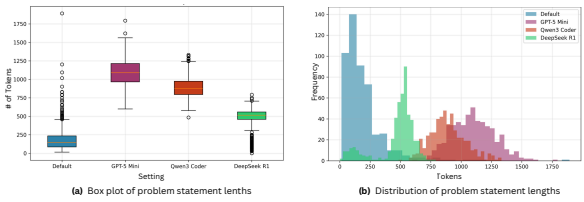


Figure 7: Distribution of problem-statement lengths (token count) for Default vs Augmented.

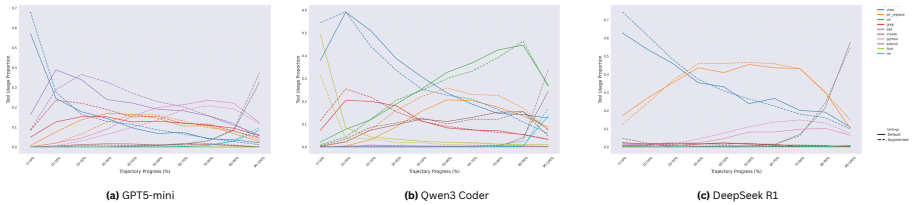


Figure 8: Evolution of tool-call proportions across a trajectory (Default vs CodeScout) for SWE-Agent.

Agent Base LLM	Default	Augmented by		
		DeepSeek R1	Qwen3 Coder	GPT5-mini
DeepSeek R1	108	125	164	132
Qwen3 Coder	183	194	209	190
GPT5-mini	194	196	207	209

Table 2: Cross-synthesis results: rows are the base (runtime) model used by the agent; columns show resolved counts when problem-statements are augmented by the indicated model.

6.6 Cross-synthesis

Table 2 (cross-synthesis) evaluates combinations where the model used to augment problem statements differs from the model used to run the agent. The key pattern is asymmetry in gains:

Weak runtime LLMs benefit most from stronger augmenters: DeepSeek R1 as the runtime model sees its largest improvement when *augmented* by Qwen3 Coder: from 108 to 164 resolved issues (+56, +51.9%). This demonstrates that a stronger off-line augmenter can compensate for limited runtime agent capabilities.

Strong runtime LLMs gain modestly from weaker augmenters: When GPT-5-mini runs the agent, augmenting with weaker models (DeepSeek R1) still yields small improvements (194 to 196, +2, +1.0%), showing robustness but diminishing returns.

Practical implication: one can use a cheaper, capable model to pre-compute augmentations for many instances (amortizing cost) and still obtain material improvements for weaker runtime agents; conversely, using a stronger augmenter can substantially raise the performance floor for weak agents.

6.7 Takeaways

Our evaluation demonstrates that CodeScout consistently enhances agent performance across diverse models and scaffolds, with structured augmentation outperforming self-augmentation approaches. These findings validate our hypothesis that systematic pre-exploration effectively supplements existing agent architectures. **Effectiveness:** Contextual problem-statement augmentation consistently improves resolution rates and localization across scaffolds and LLMs, with the largest relative gains for weaker runtime agents.

Pipeline vs autonomous augmentation: A separate, structured augmentation pipeline outperforms asking the agent to self-augment during its trajectory.

Design trade-offs: LLM-driven scoping plus relevance filtering is important—simple lexical retrieval (BM25) helps, but LLM scoping yields stronger, more semantically-relevant targets. Augmentation overhead (LLM calls, tokens) is modest relative to downstream gains for most models, though very large-trajectory LLMs maintain higher absolute token budgets.

Deployment guidance: For production settings consider (a) using a moderately capable, cheaper model for augmentation to boost weaker runtime agents, or (b) using a stronger augmenter to raise the floor of weaker agents when improving the runtime model is not feasible.

7 Conclusion

We introduced a contextual query refinement framework that transforms vague and underspecified user requests into detailed, actionable problem statements through structured codebase analysis. By combining repository graph construction, high-level context scoping, and multi-perspective code element analysis, our approach systematically grounds problem understanding before attempting solutions. Evaluation on the SWEBench-Verified benchmark demonstrates that this refinement process enhances the effectiveness of AI code assistance, particularly when operating in ambiguous or poorly defined scenarios. Beyond performance gains, our findings highlight the importance of structured problem formulation as a prerequisite for reliable software engineering support. Future work will explore extending this framework to multi-repository settings, integrating deeper program analysis techniques, and applying the approach to collaborative development environments.

8 Limitations

Our evaluation is necessarily constrained by the current state of software engineering benchmarks and tooling. We evaluate CodeScout using three language models (DeepSeek R1, GPT-5-mini, Qwen3 Coder) and three agentic scaffolds (SWE-Agent, OpenHands, Mini-SWE-Agent), which, while representing state-of-the-art systems, constitute only a subset of available approaches. Expanding to additional models and scaffolds would strengthen claims about generalizability but requires substantial computational resources and careful experimental design to ensure fair comparison. Similarly, our analysis focuses exclusively on Python repositories, not by choice but due to the availability of established benchmarks like SWEBench-Verified with reliable evaluation harnesses. Other programming languages lack comparable evaluation infrastructure that provides ground-truth issue resolution verification at scale, preventing systematic assessment of CodeScout's effectiveness across language ecosystems with different structural conventions, type systems, and semantic properties. Finally, our evaluation relies on open-source repositories from SWEBench-Verified, which may differ systematically from enterprise codebases in terms of architectural patterns, documentation practices, code quality standards, and proprietary constraints. While open-source repositories enable reproducible research, validating CodeScout's effectiveness on closed-source or domain-specific industrial code remains important future work that requires industry partnerships and appropriate data sharing agreements.

9 Ethics Statement

Our work focuses on improving AI-assisted software development through enhanced problem statement formulation. We acknowledge the inherent risks of LLM-based code assistance systems, including potential generation of insecure code, introduction of subtle bugs, and perpetuation of biases present in training data. While CodeScout aims to improve agent reliability, users should apply appropriate code review and security practices to any AI-generated solutions. All datasets used in this work—including SWEBench-Verified and associated repositories—are used in accordance with their respective licenses and usage terms. We do not introduce new datasets or collect proprietary code. Our evaluation respects the open-source nature of the benchmark and does not compromise repository security or developer privacy. The computational resources required for our experiments contribute to environmental impact through energy consumption. We encourage practitioners to consider this tradeoff when deploying augmentation systems at scale.

References

Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T Barr. 2024. Automatic semantic augmentation of language model prompts (for code summarization). pages 1–13.

Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):85–111.

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. Repairagent: An autonomous, llm-based agent for program repair. In *Proceedings of the 47th International Conference on Software Engineering*, pages 2188–2200. IEEE Computer Society.

Islem Bouzenia and Michael Pradel. 2025. Understanding software engineering agents: A study of thought-action-result trajectories. *arXiv preprint arXiv:2506.18824*.

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, pages 1430–1442. IEEE Press.

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? *Preprint*, arXiv:2310.06770.

Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. *Preprint*, arXiv:2303.07263.

Anand Kumar, Vishal Khare, Deepak Sharma, Satyam Kumar, Vijay Saini, Anshul Yadav, Sachendra Jain, Ankit Rana, Pratham Verma, Vaibhav Meena, and Avinash Edubilli. 2025. Intuition to evidence: Measuring ai's true impact on developer productivity. *Preprint*, arXiv:2509.19708.

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *Proceedings of the 45th International Conference on Software Engineering*, pages 919–931, Melbourne, Australia. IEEE/ACM.

Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large language model-based agents for software engineering: A survey. *Preprint*, arXiv:2409.02977.

Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan. 2015. Query expansion via wordnet for effective code search. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 545–549.

Xiangxin Meng, Zexiong Ma, Pengfei Gao, and Chao Peng. 2024. An empirical study on llm-based agents for automated bug fixing. *arXiv preprint arXiv:2411.10213*.

Daye Nam, Ahmed Omran, Ambar Murillo, Saksham Thakur, Abner Araujo, Marcel Blistein, Alexander Frömmgen, Vincent Hellendoorn, and Satish Chandra. 2025. Prompting llms for code editing: Struggles and remedies. *arXiv preprint arXiv:2504.20196*.

Liming Nie, He Jiang, Zhilei Ren, Zeyi Sun, and Xiaochen Li. 2016. Query expansion based on crowd knowledge for code search. *IEEE Transactions on Services Computing*, 9:771–783.

Stephen Robertson, Hugo Zaragoza, and 1 others. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

Francesco Romeo, Luigi Arena, Francesco Blefari, Francesco Aurelio Pironti, Matteo Lupinacci, and Angelo Furfaro. 2025. Arpaccino: An agentic-rag for policy as code compliance. *Preprint*, arXiv:2507.10584.

Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering*, 1(FSE):951–971.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2025. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*.

Yanlin Wang, Lianghong Guo, Ensheng Shi, Wenqing Chen, Jiachi Chen, Wanjun Zhong, Menghan Wang, Hui Li, Hongyu Zhang, Ziyu Lyu, and Zibin Zheng. 2023. You augment me: Exploring chatgpt-based data augmentation for semantic code search. *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 14–25.

Yaqi Wang and Haipei Xu. 2024. Srsa: A cost-efficient strategy-router search agent for real-world human-machine interactions. *Preprint*, arXiv:2411.14574.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37.

Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving ChatGPT for unit test generation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1703–1726.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, pages 1592–1604. Association for Computing Machinery.

Albert Ziegler, Eirini Kalliamvakou, Shawn Simister, Ganesh Sittampalam, X. Alice Li, Andrew Rice, Devon Rifkin, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, New York, NY, USA. Association for Computing Machinery.

A Prompt Design

We employ a three-stage prompting strategy to systematically augment problem statements with codebase context. Each prompt is designed to decompose the augmentation task into focused subtasks that elicit specific types of knowledge from the language model. Below we present the complete prompt templates.

A.1 Exploration Target Identification

The first prompt guides the model to identify relevant code locations given a problem statement and repository structure. This stage establishes which parts of the codebase warrant deeper investigation.

Input: Problem statement PS, repository file tree $\mathcal{F}$

Output: List of exploration targets $\mathcal{T} = \{(\text{type}, \text{name}, \text{reasoning})\}$

Stage 1: Exploration Target Identification

You are analyzing a software bug report to identify which files and code components to investigate.

PROBLEM STATEMENT:
`{problem_statement}`

REPOSITORY STRUCTURE:
`{file_tree}`

Task: Identify the most promising files, classes, and functions to explore for understanding this issue.

Focus on:

- Files/classes mentioned in the problem statement
- Related components that might be affected
- Common entry points and core functionality
- Test files that might reveal expected behavior

Return exactly 5–10 targets in this format:

```
1. target_type:file, target_name:exact_name, reasoning:why_relevant
2. target_type:class, target_name:exact_name, reasoning:why_relevant
3. target_type:function, target_name:exact_name, reasoning:why_relevant
```

Be specific with names—use exact file paths and class/function names from the tree. Each entry must be on a separate line and follow the exact format shown.

Design rationale: We ask the model to produce 5–10 targets to balance coverage with computational cost. The structured output format enables reliable parsing. By requesting explicit reasoning, we encourage the model to articulate its hypotheses about bug locations, which improves target quality.

A.2 Content Relevance Analysis

For each exploration target, we retrieve its source code and prompt the model to analyze how it relates to the reported issue. This stage extracts structured insights that will inform the final augmented problem statement.

Input: Problem statement PS, code content c, target metadata θ

Output: Structured analysis with relevance score and insights

Stage 2: Content Relevance Analysis

You are analyzing code content to understand how it relates to a specific bug report. Your analysis will directly feed into an augmented problem statement.

ORIGINAL PROBLEM:
`{problem_statement}`

CODE CONTENT:
`{content}`

TARGET INFO: `{target.target_type}` - `{target.target_name}`

Provide analysis in the following format:

RELEVANCE: X/10

ROLE_IN_ISSUE:
Describe how this code relates to the reported issue. What is its role in causing or manifesting the bug?

FIX_LOCATION_HINTS:
Identify specific locations where fixes might be needed (line numbers, methods, conditions). What type of changes might be required (validation, error handling, logic fixes)?

EXPLORATION_INSIGHTS:
Surface key technical details that help understand the issue—algorithms, data flow, dependencies, code patterns, or architectural decisions relevant to the bug. Note test scenarios or edge cases this code suggests should be examined.

ALTERNATIVE_HYPOTHESES:
Consider alternative explanations for the bug based on this code. What related components might also be involved? What different root causes does this analysis suggest?

Note: Focus on insights that will help create actionable exploration hints and fix suggestions. If relevance < 4, keep all sections brief but still provide analysis.

Design rationale: The structured output schema aligns with the final augmented problem statement format. By requesting a numerical relevance score, we enable downstream filtering of low-relevance content. The four semantic categories (role, fix hints, insights, hypotheses) encourage comprehensive analysis while maintaining organization.

A.3 Problem Statement Synthesis

Given the original problem statement and all relevant code analyses, we prompt the model to synthesize a comprehensive augmented problem statement. This stage transforms raw insights into a coherent narrative that guides developers.

Input: Original problem statement PS, filtered analyses $\mathcal{A}_{\text{rel}}$

Output: Augmented problem statement PS$'$

Stage 3: Problem Statement Synthesis

You are creating an enhanced problem statement that will help developers quickly understand and fix a software issue.
ORIGINAL PROBLEM STATEMENT: `{original_ps}`
CODE ANALYSIS RESULTS: For each analyzed target:

```
{i}. TARGET: {target_type} - {target_name}
   RELEVANCE: {score}/10
   ROLE IN ISSUE: {role_description}
   FIX LOCATION HINTS: {hints}
   EXPLORATION INSIGHTS: {insights}
   ALTERNATIVE HYPOTHESES: {hypotheses}
```

Instructions: Synthesize these analyses into a comprehensive problem statement. Include all relevant details from the original statement. Use the code analysis as context to inform your narrative—don't simply copy it verbatim.
Structure your output with these sections:
Issue Description: Rewrite the core problem clearly, incorporating technical insights.
Reproduction Steps: Include original steps if available. If the original issue includes code, preserve it here—this is essential. Add technical details about what happens internally. Include relevant stack traces or error patterns. Suggest additional reproduction scenarios if helpful.
Expected Behavior: Describe what should happen instead, informed by code analysis.
Exploration Hints:

- *Files to Examine:* `file1.py` - why relevant (role in issue, key findings)
- *Key Classes/Functions:* `ClassName.method()` - what to look for
- *Areas of Interest:* Specific code patterns or logic to investigate

Fix Hints:

- *High Confidence Locations:* Where fixes are likely needed and why
- *Implementation Suggestions:* Concrete fix ideas with limitations
- *Alternative Hypotheses:* Other potential root causes

Keep technical language precise but accessible. Focus on actionable insights.

Design rationale: The detailed template structure ensures consistency across augmented problem statements. By explicitly requesting preservation of reproduction code and technical details, we maintain the diagnostic value of the original statement while enriching it with codebase context. The hierarchical organization mirrors how developers naturally think about debugging: understanding the issue, knowing where to look, and hypothesizing about fixes.

Algorithm 1 Problem Statement Augmentation

Input: Problem statement PS, codebase $\mathcal{C}$, language model $\mathcal{M}$, threshold τ
Output: Augmented problem statement PS$'$

// Stage 1: Exploration Target Identification
$\mathcal{F} \leftarrow \textsc{GetFileTree}(\mathcal{C})$
$p_1 \leftarrow \textsc{FormatPrompt}_{\text{explore}}(\text{PS}, \mathcal{F})$
$r_1 \leftarrow \mathcal{M}(p_1)$
$\mathcal{T} \leftarrow \textsc{ParseTargets}(r_1)$

// Stage 2: Content Retrieval and Analysis
$\mathcal{A} \leftarrow [\,]$
for each target $\theta \in \mathcal{T}$ **do**
  content $\leftarrow \textsc{RetrieveCode}(\theta, \mathcal{C})$
  if content successfully retrieved **then**
    $p_2 \leftarrow \textsc{FormatPrompt}_{\text{analyze}}(\text{PS}, \text{content}, \theta)$
    $r_2 \leftarrow \mathcal{M}(p_2)$
    $a \leftarrow \textsc{ParseAnalysis}(r_2)$
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{a\}$
  end if
end for

// Stage 3: Relevance Filtering
$\mathcal{A}_{\text{rel}} \leftarrow \{a \in \mathcal{A} \mid a.\text{relevance} \geq \tau\}$

// Stage 4: Synthesis
insights $\leftarrow \textsc{FormatInsights}(\mathcal{A}_{\text{rel}})$
$p_3 \leftarrow \textsc{FormatPrompt}_{\text{synthesize}}(\text{PS}, \text{insights})$
PS$' \leftarrow \mathcal{M}(p_3)$

return PS$'$

B Augmentation Algorithm

Our approach processes each problem statement through a structured pipeline that leverages language models to systematically gather and synthesize codebase context. Algorithm 1 presents the core augmentation procedure.

B.1 Main Augmentation Procedure

Key design choices: We employ a **relevance threshold** τ to filter analyses before synthesis, ensuring only high-quality insights inform the final augmented statement. The pipeline gracefully handles retrieval failures by skipping unavailable targets rather than halting, making it robust to incom-

plete code references.

B.2 Computational Complexity

LLM API Calls: Each problem statement requires $2 + |\mathcal{T}_{\text{valid}}|$ language model calls:

- 1 call for exploration target identification
- $|\mathcal{T}_{\text{valid}}|$ calls for content analysis (typically 5–10 targets)
- 1 call for synthesis

For a dataset of size n, the total number of LLM calls is $O(n \cdot |\mathcal{T}|)$. In practice, with approximately 7 valid targets per instance, we observe roughly 9 API calls per problem statement.

C CodeScout: Further Analysis

C.1 Qualitative Examples

We present detailed visualizations of the CodeScout context retrieval pipeline across five diverse instances from the SWE-bench Verified dataset. Each instance demonstrates the three-stage pipeline: (1) high-level context scoping to identify relevant files, classes, and functions, (2) fine-grained content analysis with relevance scoring, and (3) problem statement augmentation. We show results from three different retrieval methods using DeepSeek-R1, Qwen3-Coder, and GPT-5-mini.

C.2 Quantitaive Comparison

We present comprehensive comparative analysis of the CodeScout pipeline for the chosen LLMs: DeepSeek-R1, Qwen3-Coder, and GPT-5-mini.

Score Distributions. Figure 24 shows the distribution of relevance scores assigned by each method across all targets. Qwen3-Coder and GPT-5-mini show higher concentrations of scores in the 7-9 range, while DeepSeek-R1 exhibits a broader distribution with more scores in the lower ranges. Figure 25 displays the distribution of maximum scores per instance, showing that Qwen3-Coder and GPT-5-mini more frequently assign maximum scores of 8-9, whereas DeepSeek-R1 shows greater variability.

Target Coverage. Figure 26 presents the distribution of the number of targets identified per instance for each method. DeepSeek-R1 tends to identify more targets per instance with a broader distribution, while Qwen3-Coder shows a tighter distribution around 6 targets. Figure 27 illustrates the overlap of unique targets identified across methods, showing that a substantial portion of targets are method-specific, with GPT-5-mini identifying the largest unique set.

Agreement Analysis. Figure 28 presents Bland-Altman plots comparing the average scores between method pairs. These plots reveal systematic differences: DeepSeek-R1 consistently scores lower than both Qwen3-Coder and GPT-5-mini, as evidenced by the negative mean differences. The agreement limits show the range of score differences across instances.

Score Correlations. Figure 29 shows heatmaps of score agreement for targets that were analyzed by multiple methods. Each cell shows the count of targets where the row method assigned one score and the column method assigned another. The diagonal entries represent exact agreement, while off-diagonal entries indicate disagreement. The conditional probabilities in the colorbar show how likely one method is to assign a particular score given the other method's score.

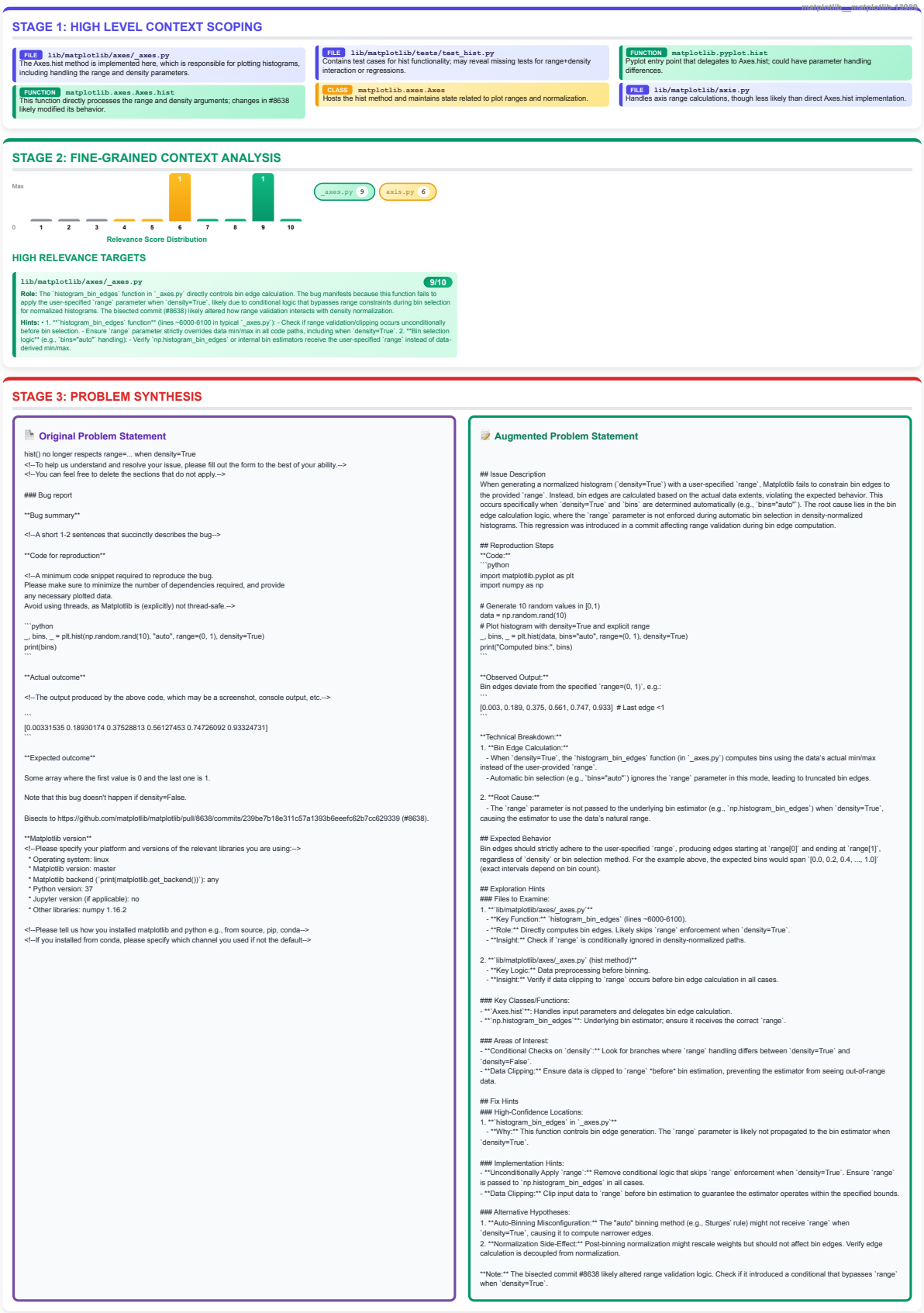


Figure 9: CodeScout visualization for `matplotlib__matplotlib-13989` with DeepSeek-R1.

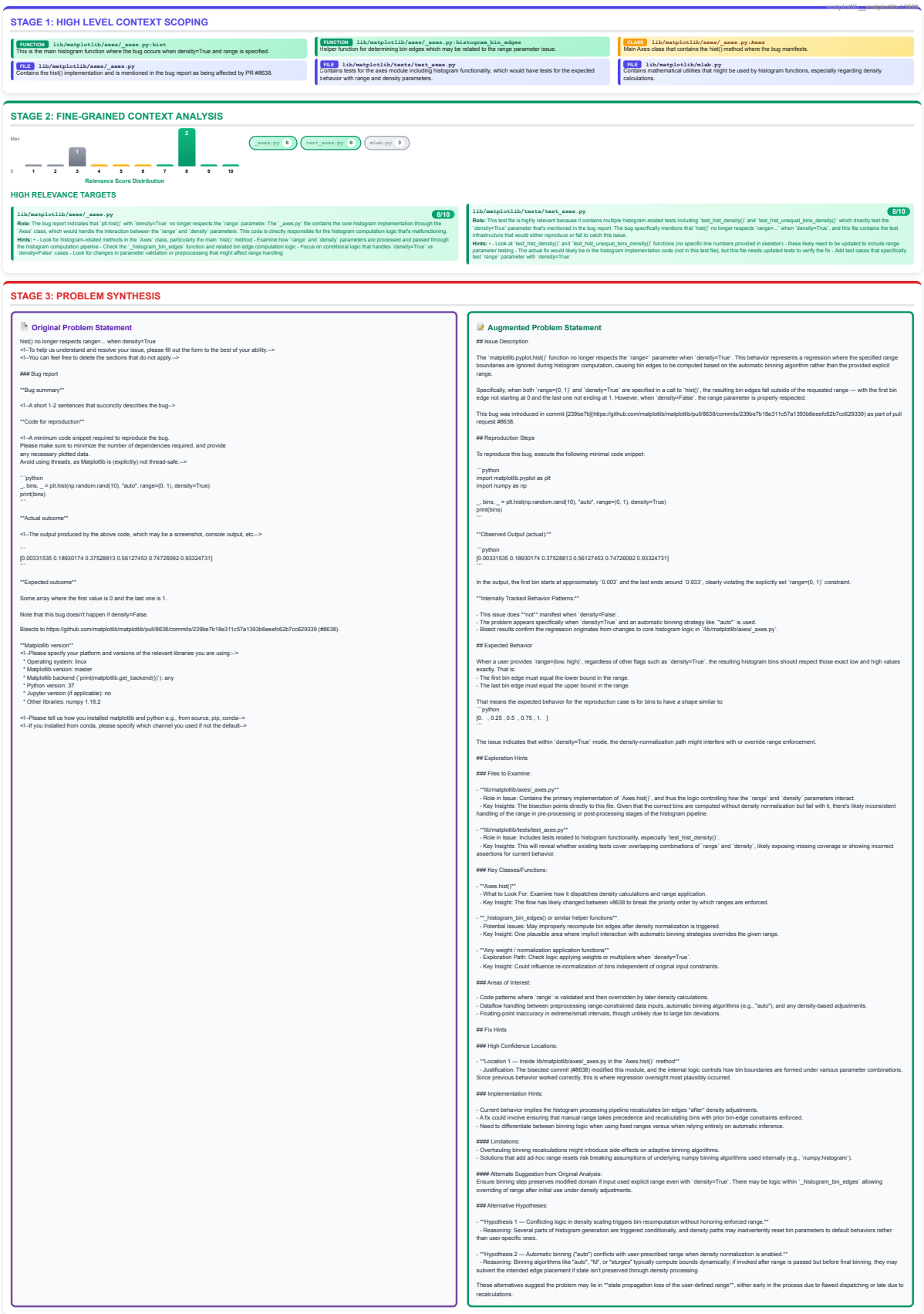


Figure 10: CodeScout visualization for `matplotlib__matplotlib-13989` with Qwen3-Coder.

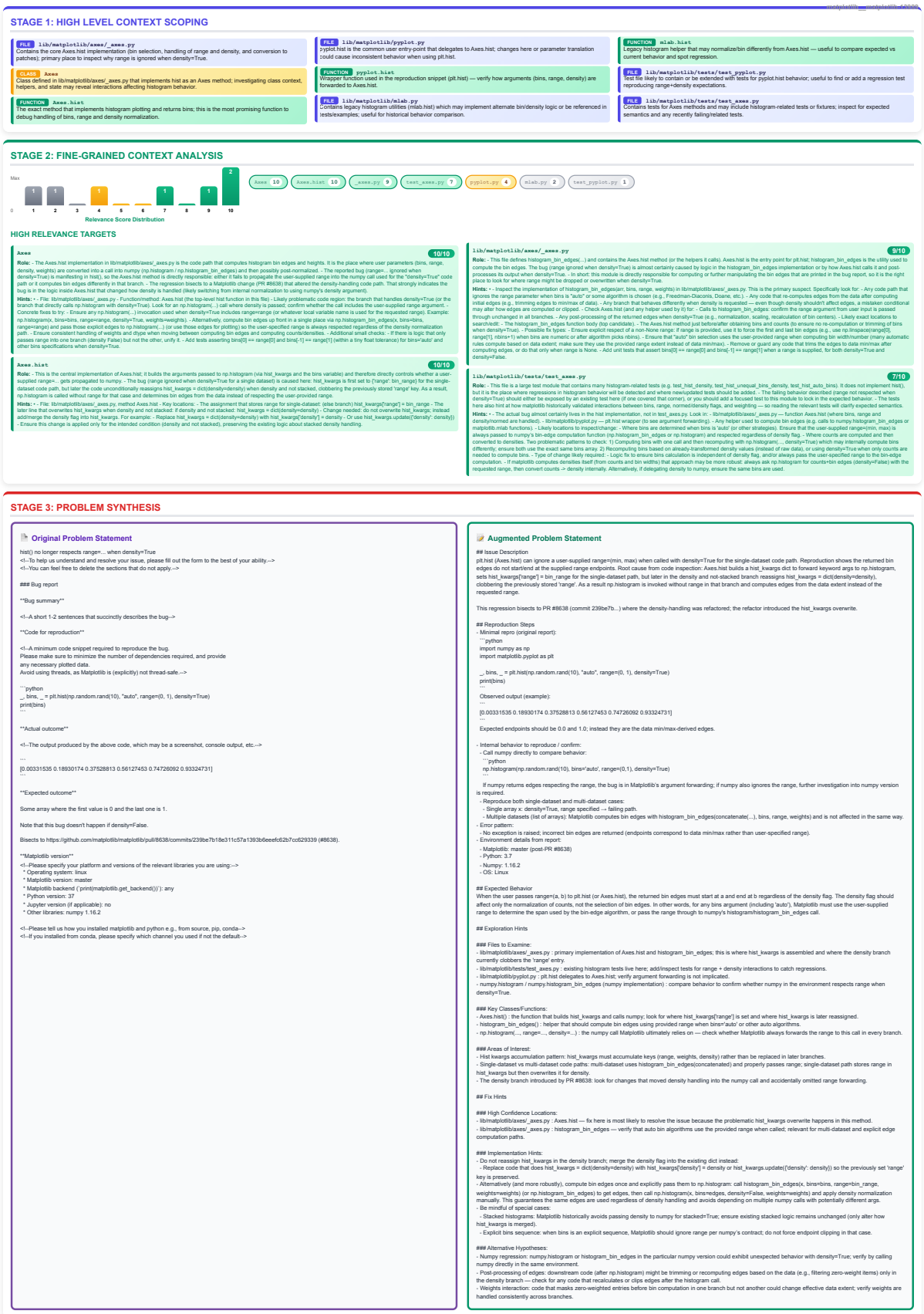


Figure 11: CodeScout visualization for `matplotlib__matplotlib-13989` with GPT-5-mini.

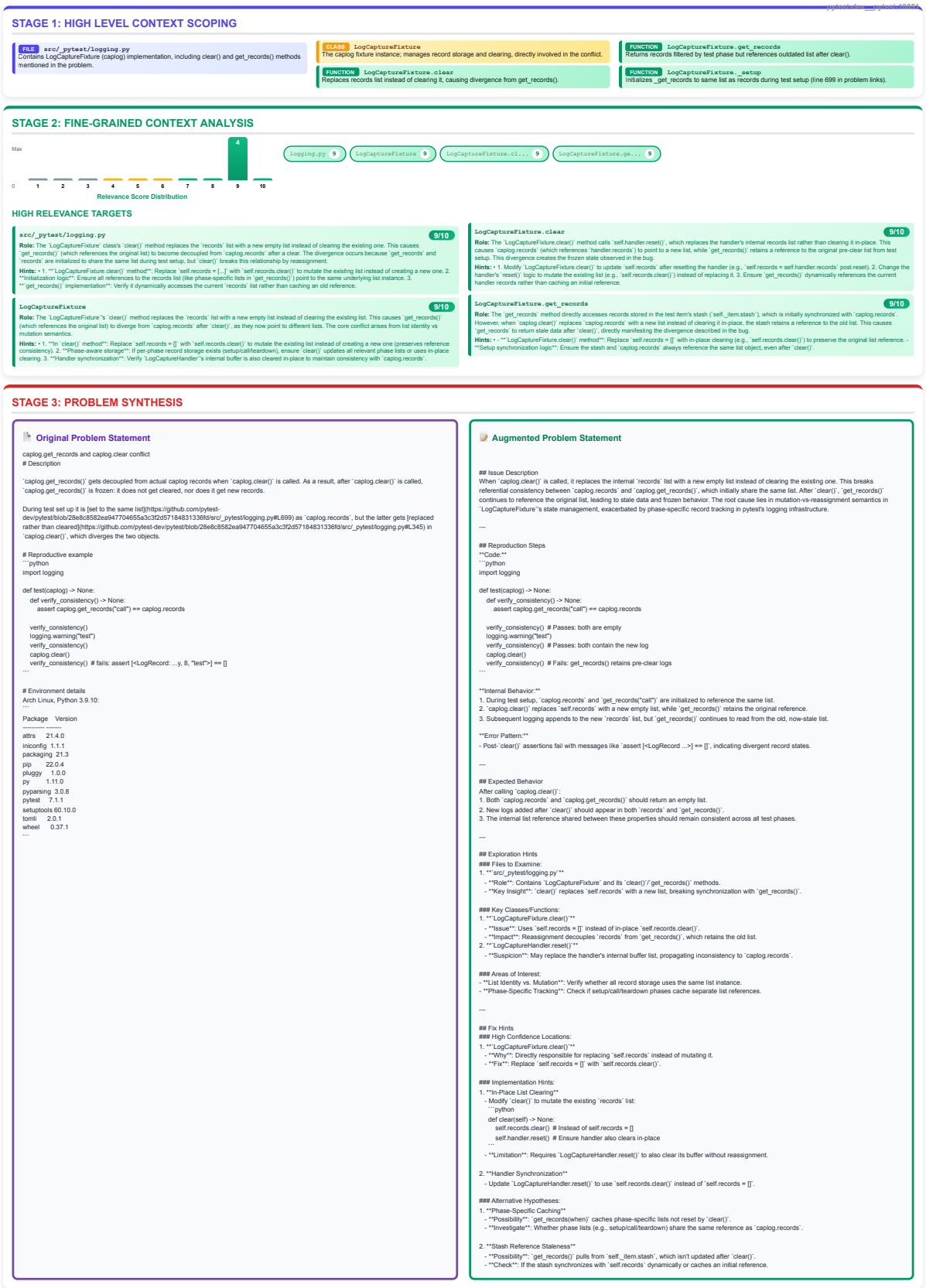


Figure 12: CodeScout visualization for `pytest-dev__pytest-10051` with DeepSeek-R1.

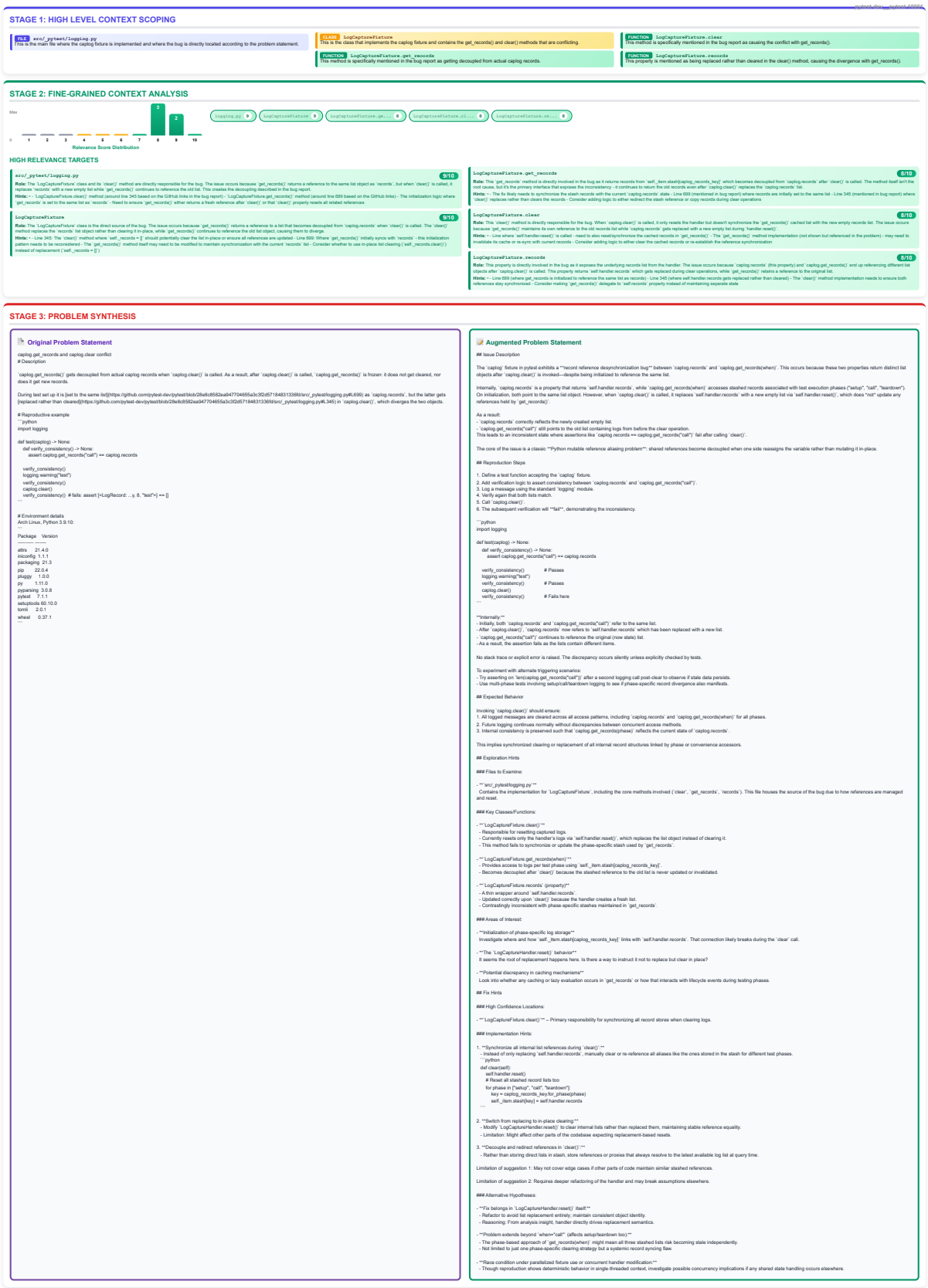


Figure 13: CodeScout visualization for `pytest-dev__pytest-10051` with Qwen3-Coder.

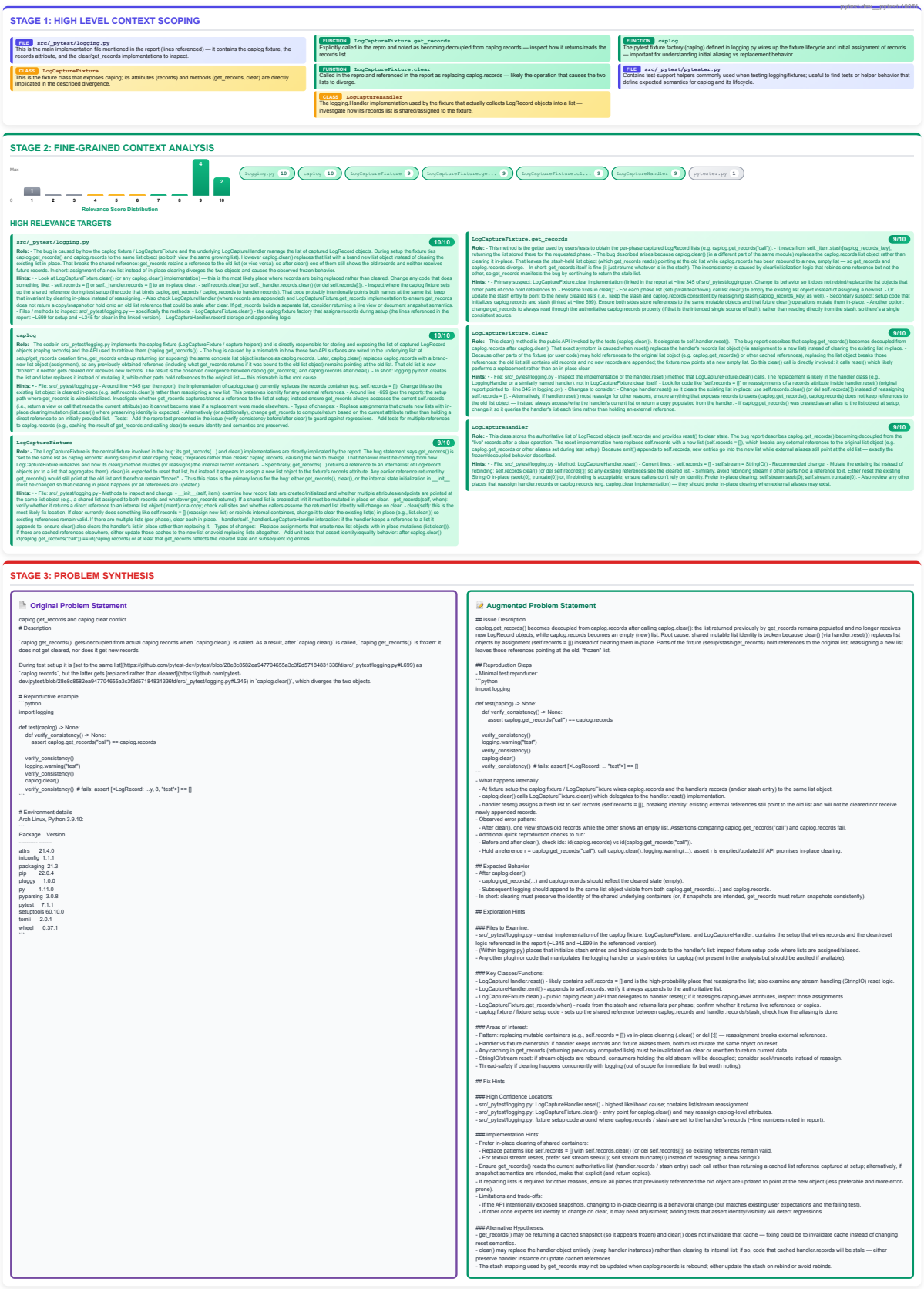

STAGE 1: HIGH LEVEL CONTEXT SCOPING
FILE src/_pytest/logging.py
This is the main implementation file mentioned in the report (lines referenced) — it contains the caplog fixture, the records attribute, and the clear/get_records implementations to inspect.
CLASS LogCaptureFixture
This is the fixture class that exposes caplog; its attributes (records) and methods (get_records, clear) are directly implicated in the described divergence.
FUNCTION LogCaptureFixture.get_records
Explicitly called in the repro and noted as becoming decoupled from caplog.records — inspect how it returns/reads the records list.
FUNCTION LogCaptureFixture.clear
Called in the repro and referenced in the report as replacing caplog.records — likely the operation that causes the two lists to diverge.
CLASS LogCaptureHandler
The logging.Handler implementation used by the fixture that actually collects LogRecord objects into a list — investigate how its records list is shared/assigned to the fixture.
FUNCTION caplog
The pytest fixture factory (caplog) defined in logging.py wires up the fixture lifecycle and initial assignment of records — important for understanding initial aliasing vs replacement behavior.
FILE src/_pytest/pytester.py
Contains test-support helpers commonly used when testing logging/fixtures; useful to find tests or helper behavior that define expected semantics for caplog and its lifecycle.
STAGE 2: FINE-GRAINED CONTEXT ANALYSIS
Max
0
1 2 3 4 5 6 7 8 9 10
Relevance Score Distribution
logging.py 10
caplog 10
LogCaptureFixture 9
LogCaptureFixture.ge... 9
LogCaptureFixture.cl... 9
LogCaptureHandler 9
pytester.py 1
HIGH RELEVANCE TARGETS
src/_pytest/logging.py 10/10
Role: - The bug is caused by how the caplog fixture / LogCaptureFixture and the underlying LogCaptureHandler manage the list of captured LogRecord objects. During setup the fixture ties caplog.get_records() and caplog.records to the same list object (so both view the same growing list). However caplog.clear() replaces that list with a brand new list object instead of clearing the existing list in-place. That breaks the shared reference: get_records retains a reference to the old list (or vice versa), so after clear() one of them still shows the old records and neither receives future records. In short: assignment of a new list instead of in-place clearing diverges the two objects and causes the observed frozen behavior.
Hints: - - Look at LogCaptureFixture.clear() (or any caplog.clear() implementation) — this is the most likely place where records are being replaced rather than cleared. Change any code that does something like: - self.records = [] or self._handler.records = [] to an in-place clear: - self.records.clear() or self._handler.records.clear() (or del self.records[:]). - Inspect where the caplog fixture sets up the shared reference during test setup (the code that binds caplog.get_records / caplog.records to handler.records). That code probably intentionally points both names at the same list; keep that invariant by clearing in-place instead of reassigning. - Also check LogCaptureHandler (where records are appended) and LogCaptureFixture.get_records implementation to ensure get_records does not return a copy/snapshot or hold onto an old list reference that could be stale after clear. If get_records builds a separate list, consider returning a live view or document snapshot semantics. - Files / methods to inspect: src/_pytest/logging.py — specifically the methods: - LogCaptureFixture.clear() - the caplog fixture factory that assigns records during setup (the lines referenced in the report: ~L699 for setup and ~L345 for clear in the linked version). - LogCaptureHandler.record storage and appending logic.
caplog 10/10
Role: - The code in src/_pytest/logging.py implements the caplog fixture (LogCaptureFixture / capture helpers) and is directly responsible for storing and exposing the list of captured LogRecord objects (caplog.records) and the API used to retrieve them (caplog.get_records()). - The bug is caused by a mismatch in how those two API surfaces are wired to the underlying list: at setup/get_records creation time, get_records ends up returning (or exposing) the same concrete list object instance as caplog.records. Later, caplog.clear() replaces caplog.records with a brand-new list object (assignment), so any previously obtained reference (including what get_records returns if it was bound to the old list object) remains pointing at the old list. That old list is now "frozen": it neither gets cleared nor receives new records. The result is the observed divergence between caplog.get_records() and caplog.records after clear(). - In short: logging.py both creates the list and later replaces it instead of mutating it, while other parts hold references to the original list — this mismatch is the root cause.
Hints: - - File: src/_pytest/logging.py - Around line ~345 (per the report): the implementation of caplog.clear() currently replaces the records container (e.g. self.records = []). Change this so the existing list object is cleared in-place (e.g. self.records.clear()) rather than reassigning a new list. This preserves identity for any external references. - Around line ~699 (per the report): the setup path where get_records is wired/initialized. Investigate whether get_records captures/stores a reference to the list at setup; instead ensure get_records always accesses the current self.records (i.e., return a view or call that reads the current attribute) so it cannot become stale if a replacement were made elsewhere. - Types of changes: - Replace assignments that create new lists with in-place clearing/mutation (list.clear()) where preserving identity is expected. - Alternatively (or additionally), change get_records to compute/return based on the current attribute rather than holding a direct reference to an initially provided list. - Tests: - Add the repro test presented in the issue (verify consistency before/after clear) to guard against regressions. - Add tests for multiple references to caplog.records (e.g., caching the result of get_records and calling clear) to ensure identity and semantics are preserved.
LogCaptureFixture 9/10
Role: - The LogCaptureFixture is the central fixture involved in the bug: its get_records(...) and clear() implementations are directly implicated by the report. The bug statement says get_records() is "set to the same list as caplog.records" during setup but later caplog.clear() "replaces rather than clears" caplog.records, causing the two to diverge. That behavior must be coming from how LogCaptureFixture initializes and how its clear() method mutates (or reassigns) the internal record containers. - Specifically, get_records(...) returns a reference to an internal list of LogRecord objects (or to a list that aggregates them). clear() is expected to reset that list, but instead it appears to assign a new list object to the fixture's records attribute. Any earlier reference returned by get_records() would still point at the old list and therefore remain "frozen". - Thus this class is the primary locus for the bug: either get_records(), clear(), or the internal state initialization in __init__ must be changed so that clearing in place happens (or all references are updated).
Hints: - - File: src/_pytest/logging.py - Methods to inspect and change: - __init__(self, item): examine how record lists are created/initialized and whether multiple attributes/endpoints are pointed at the same list object (e.g., a shared list assigned to both records and whatever get_records returns). If a shared list is created at init it must be mutated in place on clear. - get_records(self, when): verify whether it returns a direct reference to an internal list object (intent) or a copy; check call sites and whether callers assume the returned list identity will change on clear. - clear(self): this is the most likely fix location. If clear currently does something like self.records = [] (reassign new list) or rebinds internal containers, change it to clear the existing list(s) in-place (e.g., list.clear()) so existing references remain valid. If there are multiple lists (per-phase), clear each in-place. - handler/self._handler/LogCaptureHandler interaction: if the handler keeps a reference to a list it appends to, ensure clear() also clears the handler's list in-place rather than replacing it. - Types of changes: - Replace assignments that create new list objects with in-place mutations (list.clear()). - If there are cached references elsewhere, either update those caches to the new list or avoid replacing lists altogether. - Add unit tests that assert identity/equality behavior: after caplog.clear() id(caplog.get_records("call")) == id(caplog.records) or at least that get_records reflects the cleared state and subsequent log entries.
LogCaptureFixture.get_records 9/10
Role: - This method is the getter used by users/tests to obtain the per-phase captured LogRecord lists (e.g. caplog.get_records("call")). - It reads from self._item.stash[caplog_records_key], returning the list stored there for the requested phase. - The bug described arises because caplog.clear() (in a different part of the same module) replaces the caplog.records list object rather than clearing it in-place. That leaves the stash-held list object (which get_records reads) pointing at the old list while caplog.records has been rebound to a new, empty list — so get_records and caplog.records diverge. - In short: get_records itself is fine (it just returns whatever is in the stash). The inconsistency is caused by clear/initialization logic that rebinds one reference but not the other, so get_records manifests the bug by continuing to return the stale list.
Hints: - - Primary suspect: LogCaptureFixture.clear implementation (linked in the report at ~line 345 of src/_pytest/logging.py). Change its behavior so it does not rebind/replace the list objects that other parts of code hold references to. - Possible fixes in clear(): - For each phase list (setup/call/teardown), call list.clear() to empty the existing list object instead of assigning a new list. - Or update the stash entry to point to the newly created lists (i.e., keep the stash and caplog.records consistent by reassigning stash[caplog_records_key] as well). - Secondary suspect: setup code that initializes caplog.records and stash (linked at ~line 699). Ensure both sides store references to the same mutable objects and that future clear() operations mutate them in-place. - Another option: change get_records to always read through the authoritative caplog.records property (if that is the intended single source of truth), rather than reading directly from the stash, so there's a single consistent source.
LogCaptureFixture.clear 9/10
Role: - This clear() method is the public API invoked by the tests (caplog.clear()). It delegates to self.handler.reset(). - The bug report describes that caplog.get_records() becomes decoupled from caplog.records after caplog.clear(). That exact symptom is caused when reset() replaces the handler's records list object (via assignment to a new list) instead of clearing the existing list in-place. - Because other parts of the fixture (or user code) may hold references to the original list object (e.g. caplog.get_records() or other cached references), replacing the list object breaks those references: the old list still contains old records and no new records are appended; the fixture now points at a new empty list. So this clear() call is directly involved: it calls reset() which likely performs a replacement rather than an in-place clear.
Hints: - - File: src/_pytest/logging.py - Inspect the implementation of the handler.reset() method that LogCaptureFixture.clear() calls. The replacement is likely in the handler class (e.g., LoggingHandler or a similarly named handler), not in LogCaptureFixture.clear itself. - Look for code like "self.records = []" or reassignments of a records attribute inside handler.reset() (original report pointed to ~line 345 in logging.py). - Changes to consider: - Change handler.reset() so it clears the existing list in-place: use self.records.clear() (or del self.records[:]) instead of reassigning self.records = []. - Alternatively, if handler.reset() must reassign for other reasons, ensure anything that exposes records to users (caplog.get_records(), caplog.records) does not keep references to the old list object — instead always access/write the handler's current list or return a copy populated from the handler. - If caplog.get_records() was created as an alias to the list object at setup, change it so it queries the handler's list each time rather than holding an external reference.
LogCaptureHandler 9/10
Role: - This class stores the authoritative list of LogRecord objects (self.records) and provides reset() to clear state. The bug report describes caplog.get_records() becoming decoupled from the "live" records after a clear operation. The reset implementation here replaces self.records with a new list (self.records = []), which breaks any external references to the original list object (e.g. caplog.get_records or other aliases set during test setup). Because emit() appends to self.records, new entries go into the new list while external aliases still point at the old list — exactly the frozen/decoupled behavior described.
Hints: - - File: src/_pytest/logging.py - Method: LogCaptureHandler.reset() - Current lines: - self.records = [] - self.stream = StringIO() - Recommended change: - Mutate the existing list instead of rebinding: self.records.clear() (or del self.records[:]) so any existing references see the cleared list. - Similarly, avoid rebinding stream if other parts hold a reference to it. Either reset the existing StringIO in-place (seek(0); truncate(0)) or, if rebinding is acceptable, ensure callers don't rely on identity. Prefer in-place clearing: self.stream.seek(0); self.stream.truncate(0). - Also review any other places that reassign handler.records or caplog.records (e.g. caplog.clear implementation) — they should prefer in-place clearing when external aliases may exist.
STAGE 3: PROBLEM SYNTHESIS
Original Problem Statement
caplog.get_records and caplog.clear conflict
Description
`caplog.get_records()` gets decoupled from actual caplog records when `caplog.clear()` is called. As a result, after `caplog.clear()` is called, `caplog.get_records()` is frozen: it does not get cleared, nor does it get new records.
During test set up it is [set to the same list](https://github.com/pytest-dev/pytest/blob/28e8c8582ea947704655a3c3f2d57184831336fd/src/_pytest/logging.py#L699) as `caplog.records`, but the latter gets [replaced rather than cleared](https://github.com/pytest-dev/pytest/blob/28e8c8582ea947704655a3c3f2d57184831336fd/src/_pytest/logging.py#L345) in `caplog.clear()`, which diverges the two objects.
Reproductive example
```python
import logging
def test(caplog) -> None:
def verify_consistency() -> None:
assert caplog.get_records("call") == caplog.records
verify_consistency()
logging.warning("test")
verify_consistency()
caplog.clear()
verify_consistency() # fails: assert [<LogRecord: ..., y, 8, "test">] == []
```
Environment details
Arch Linux, Python 3.9.10:
```
Package Version
---------- -------
attrs 21.4.0
iniconfig 1.1.1
packaging 21.3
pip 22.0.4
pluggy 1.0.0
py 1.11.0
pyparsing 3.0.8
pytest 7.1.1
setuptools 60.10.0
tomli 2.0.1
wheel 0.37.1
```
Augmented Problem Statement
Issue Description
caplog.get_records() becomes decoupled from caplog.records after calling caplog.clear(): the list returned previously by get_records remains populated and no longer receives new LogRecord objects, while caplog.records becomes an empty (new) list. Root cause: shared mutable list identity is broken because clear() (via handler.reset()) replaces list objects by assignment (self.records = []) instead of clearing them in-place. Parts of the fixture (setup/stash/get_records) hold references to the original list; reassigning a new list leaves those references pointing at the old, "frozen" list.
Reproduction Steps
- Minimal test reproducer:
```python
import logging
def test(caplog) -> None:
def verify_consistency() -> None:
assert caplog.get_records("call") == caplog.records
verify_consistency()
logging.warning("test")
verify_consistency()
caplog.clear()
verify_consistency() # fails: assert [<LogRecord: ... "test">] == []
```
- What happens internally:
- At fixture setup the caplog fixture / LogCaptureFixture wires caplog.records and the handler's records (and/or stash entry) to the same list object.
- caplog.clear() calls LogCaptureFixture.clear() which delegates to the handler.reset() implementation.
- handler.reset() assigns a fresh list to self.records (self.records = []), breaking identity: existing external references still point to the old list and will not be cleared nor receive newly appended records.
- Observed error pattern:
- After clear(), one view shows old records while the other shows an empty list. Assertions comparing caplog.get_records("call") and caplog.records fail.
- Additional quick reproduction checks to run:
- Before and after clear(), check ids: id(caplog.records) vs id(caplog.get_records("call")).
- Hold a reference r = caplog.get_records("call"); call caplog.clear(); logging.warning(...); assert r is emptied/updated if API promises in-place clearing.
Expected Behavior
- After caplog.clear():
- caplog.get_records(...) and caplog.records should reflect the cleared state (empty).
- Subsequent logging should append to the same list object visible from both caplog.get_records(...) and caplog.records.
- In short: clearing must preserve the identity of the shared underlying containers (or, if snapshots are intended, get_records must return snapshots consistently).
Exploration Hints
Files to Examine:
- src/_pytest/logging.py - central implementation of the caplog fixture, LogCaptureFixture, and LogCaptureHandler; contains the setup that wires records and the clear/reset logic referenced in the report (~L345 and ~L699 in the referenced version).
- (Within logging.py) places that initialize stash entries and bind caplog.records to the handler's list: inspect fixture setup code where lists are assigned/aliased.
- Any other plugin or code that manipulates the logging handler or stash entries for caplog (not present in the analysis but should be audited if available).
Key Classes/Functions:
- LogCaptureHandler.reset() - likely contains self.records = [] and is the high-probability place that reassigns the list; also examine any stream handling (StringIO) reset logic.
- LogCaptureHandler.emit() - appends to self.records; verify it always appends to the authoritative list.
- LogCaptureFixture.clear() - public caplog.clear() API that delegates to handler.reset(); if it reassigns caplog-level attributes, inspect those assignments.
- LogCaptureFixture.get_records(when) - reads from the stash and returns lists per phase; confirm whether it returns live references or copies.
- caplog fixture / fixture setup code - sets up the shared reference between caplog.records and handler.records/stash; check how the aliasing is done.
Areas of Interest:
- Pattern: replacing mutable containers (e.g., self.records = []) vs in-place clearing (.clear() or del [:]) — reassignment breaks external references.
- Handler vs fixture ownership: if handler keeps records and fixture aliases them, both must mutate the same object on reset.
- Any caching in get_records (returning previously computed lists) must be invalidated on clear or rewritten to return current data.
- StringIO/stream reset: if stream objects are rebound, consumers holding the old stream will be decoupled; consider seek/truncate instead of reassign.
- Thread-safety if clearing happens concurrently with logging (out of scope for immediate fix but worth noting).
Fix Hints
High Confidence Locations:
- src/_pytest/logging.py: LogCaptureHandler.reset() - highest likelihood cause; contains list/stream reassignment.
- src/_pytest/logging.py: LogCaptureFixture.clear() - entry point for caplog.clear() and may reassign caplog-level attributes.
- src/_pytest/logging.py: fixture setup code around where caplog.records / stash are set to the handler's records (~line numbers noted in report).
Implementation Hints:
- Prefer in-place clearing of shared containers:
- Replace patterns like self.records = [] with self.records.clear() (or del self.records[:]) so existing references remain valid.
- For textual stream resets, prefer self.stream.seek(0); self.stream.truncate(0) instead of reassigning a new StringIO.
- Ensure get_records() reads the current authoritative list (handler.records / stash entry) each call rather than returning a cached list reference captured at setup; alternatively, if snapshot semantics are intended, make that explicit (and return copies).
- If replacing lists is required for other reasons, ensure all places that previously referenced the old object are updated to point at the new object (less preferable and more error-prone).
- Limitations and trade-offs:
- If the API intentionally exposed snapshots, changing to in-place clearing is a behavioral change (but matches existing user expectations and the failing test).
- If other code expects list identity to change on clear, it may need adjustment; adding tests that assert identity/visibility will detect regressions.
Alternative Hypotheses:
- get_records() may be returning a cached snapshot (so it appears frozen) and clear() does not invalidate that cache — fixing could be to invalidate cache instead of changing reset semantics.
- clear() may replace the handler object entirely (swap handler instances) rather than clearing its internal list; if so, code that cached handler.records will be stale — either preserve handler instance or update cached references.
- The stash mapping used by get_records may not be updated when caplog.records is rebound; either update the stash on rebind or avoid rebinds.

Figure 14: CodeScout visualization for `pytest-dev__pytest-10051` with GPT-5-mini.

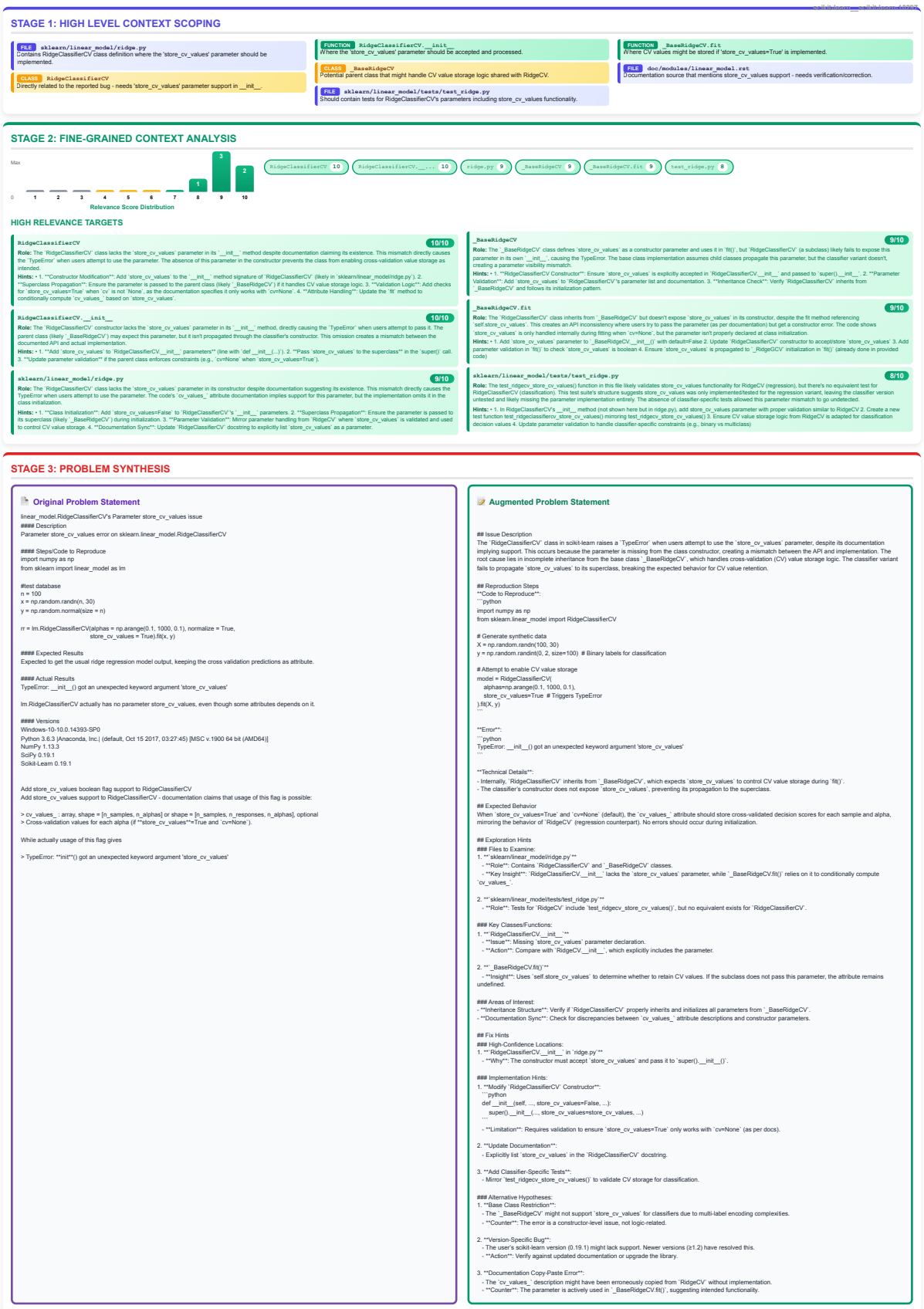


Figure 15: CodeScout visualization for `scikit-learn__scikit-learn-10297` with DeepSeek-R1.

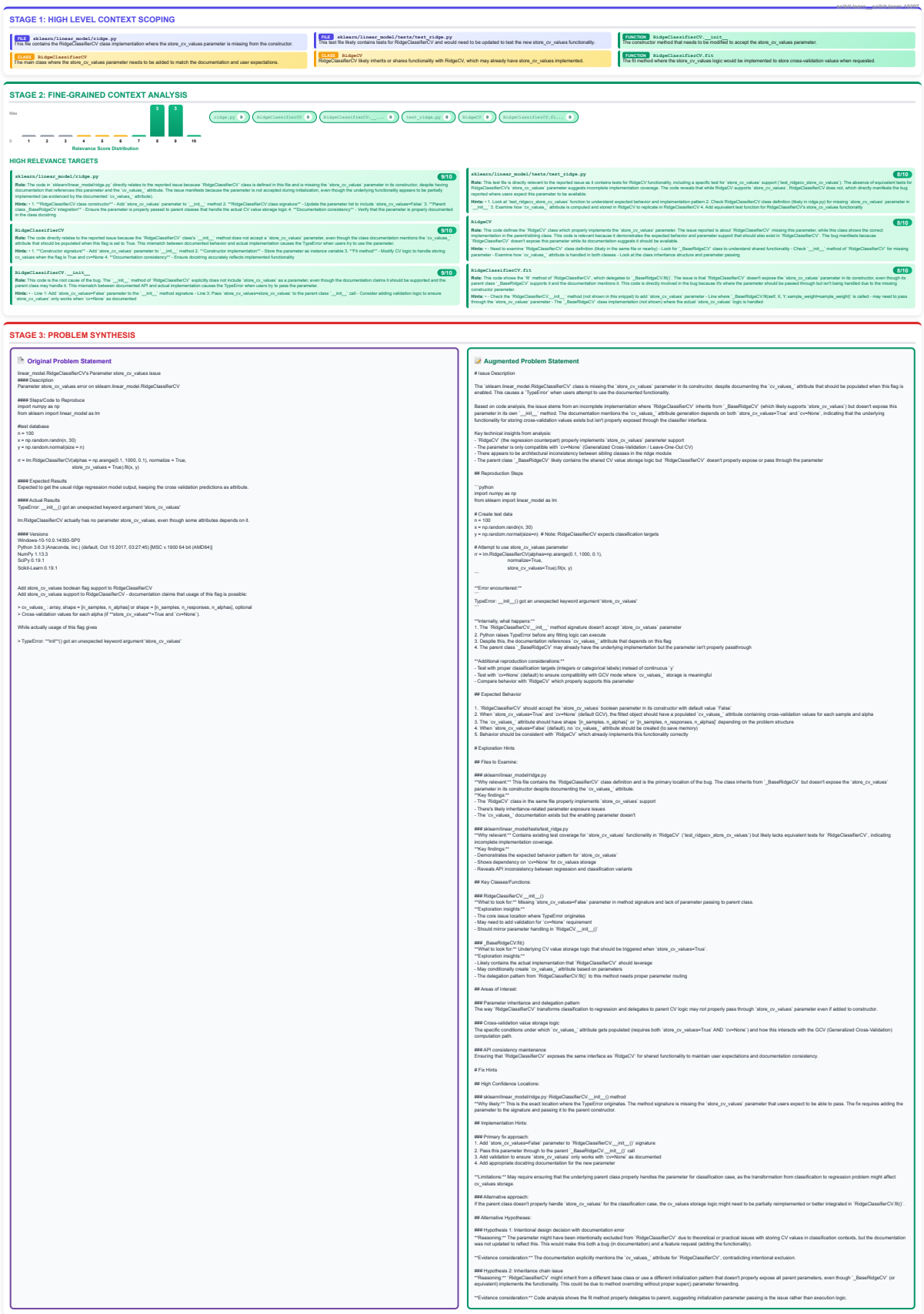


Figure 16: CodeScout visualization for `scikit-learn__scikit-learn-10297` with Qwen3-Coder.

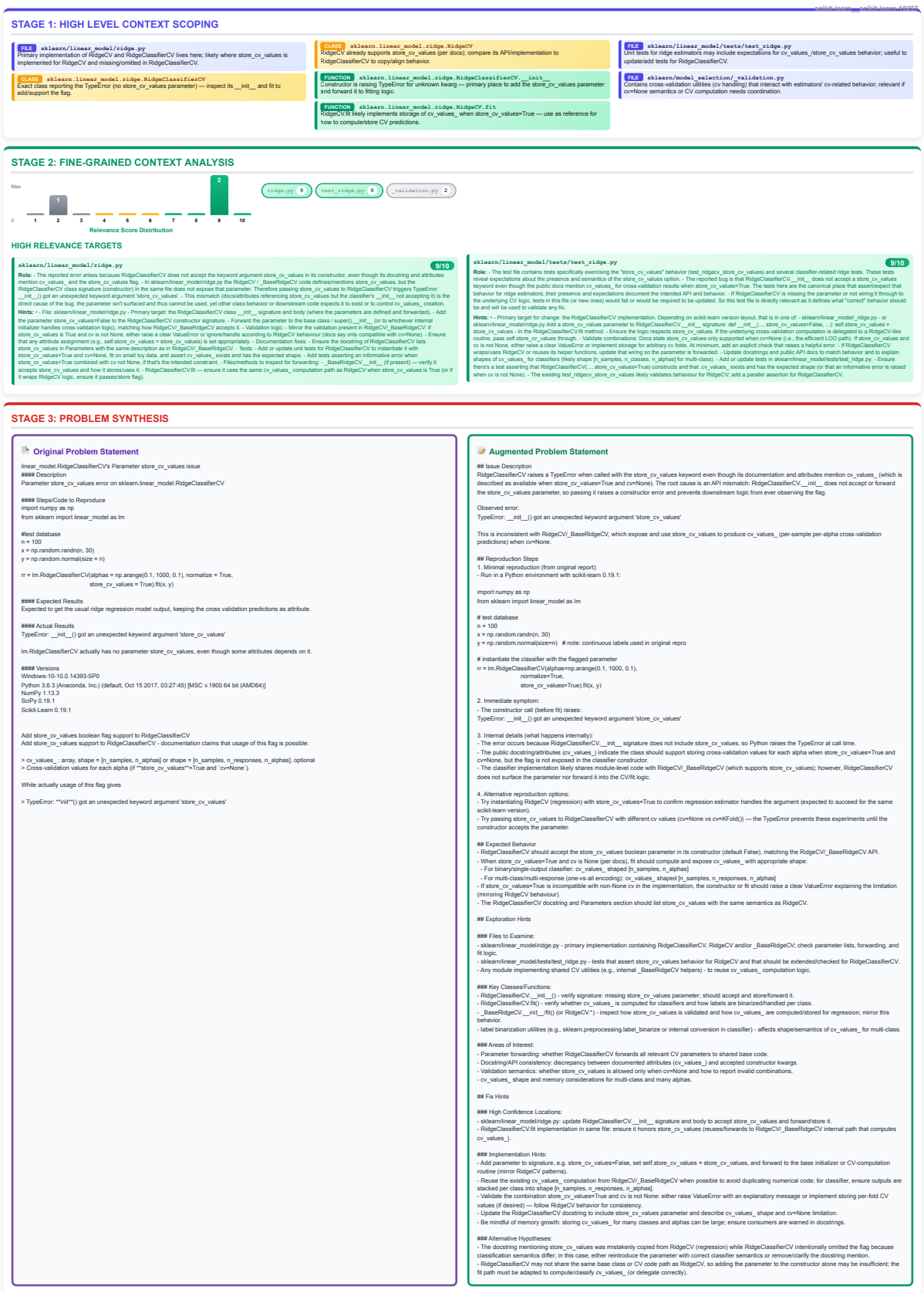


Figure 17: CodeScout visualization for `scikit-learn__scikit-learn-10297` with GPT-5-mini.

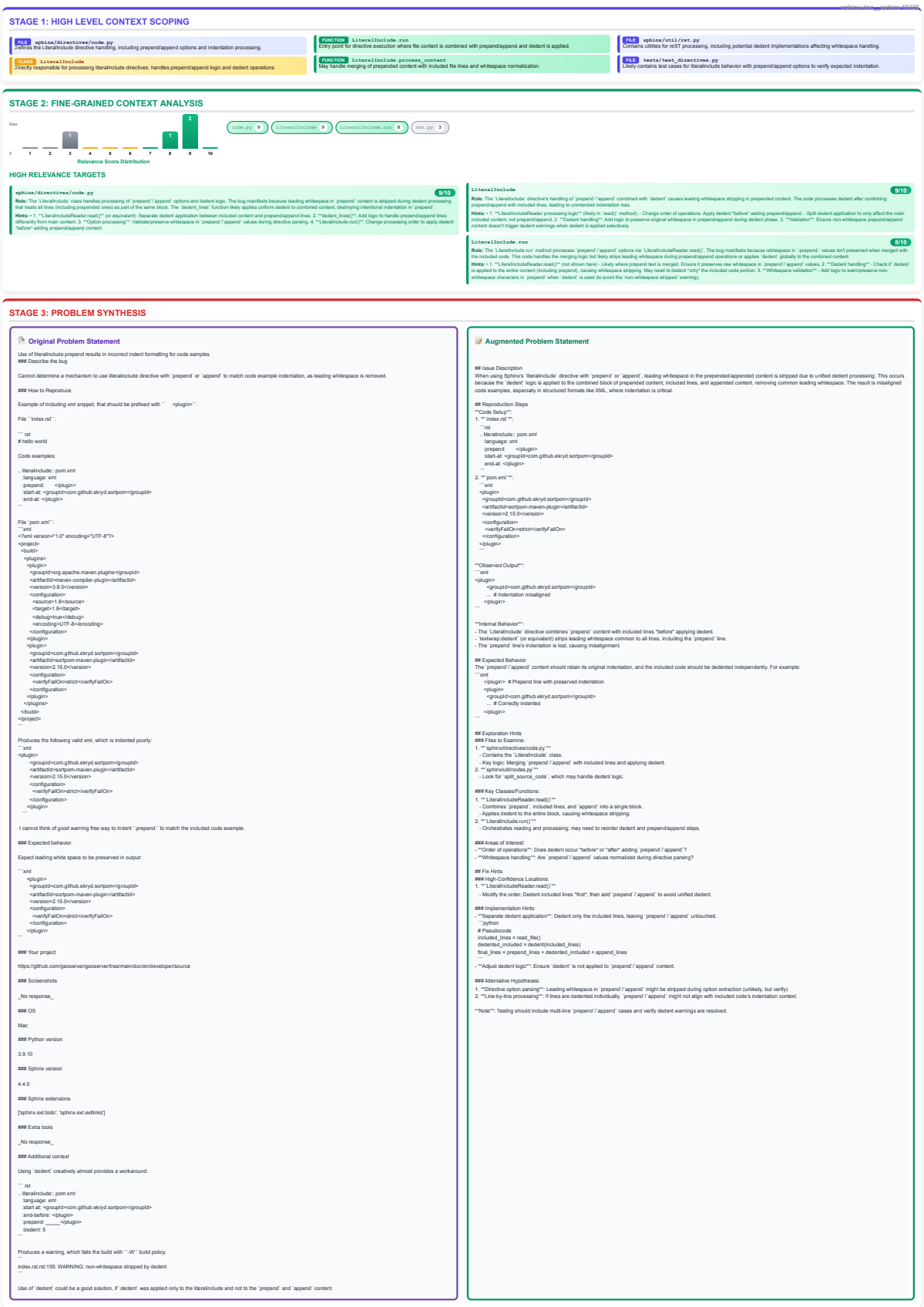


Figure 18: CodeScout visualization for `sphinx-doc__sphinx-10323` with DeepSeek-R1.

Figure 19: CodeScout visualization for `sphinx-doc__sphinx-10323` with Qwen3-Coder.

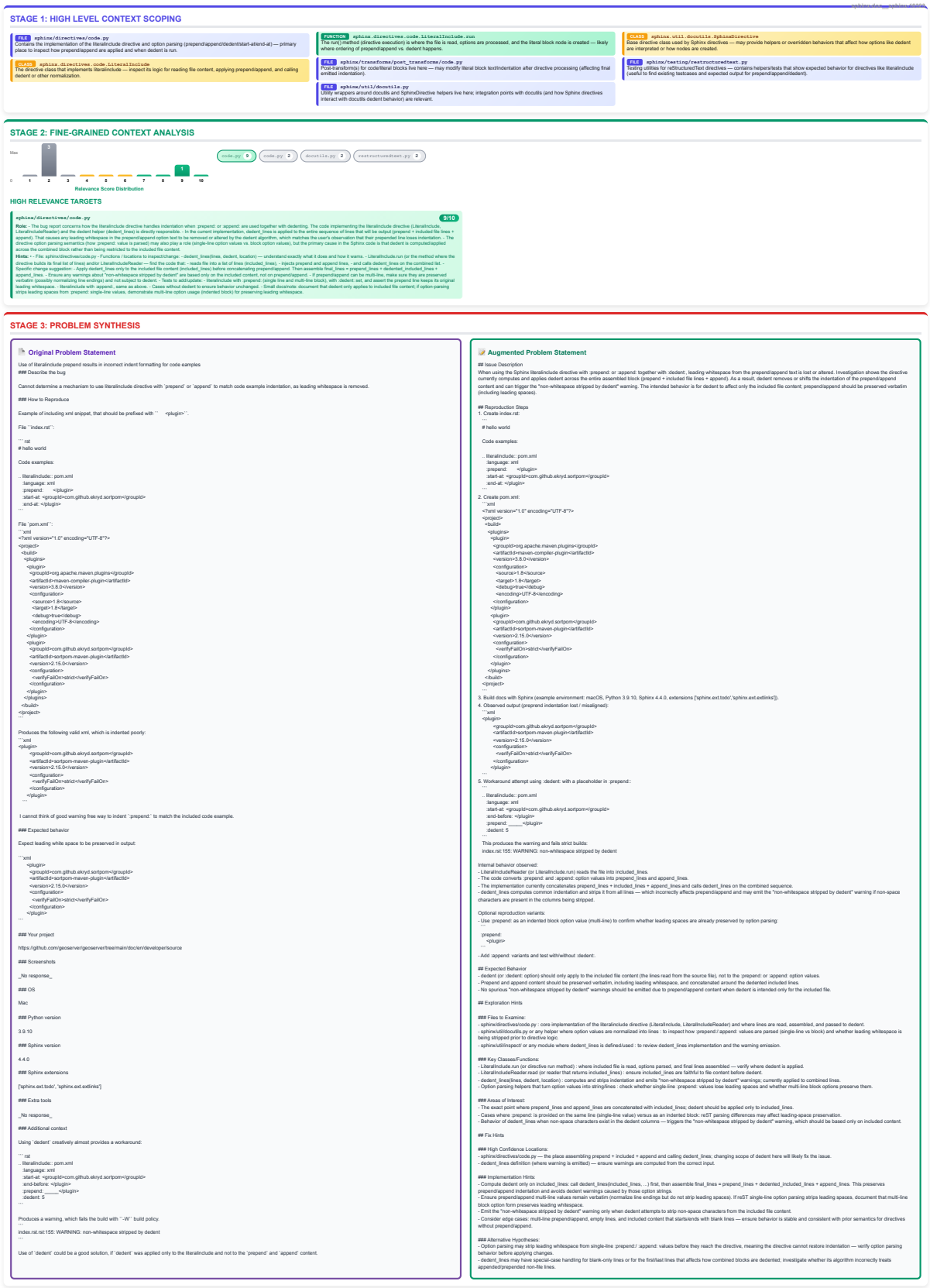


Figure 20: CodeScout visualization for `sphinx-doc__sphinx-10323` with GPT-5-mini.

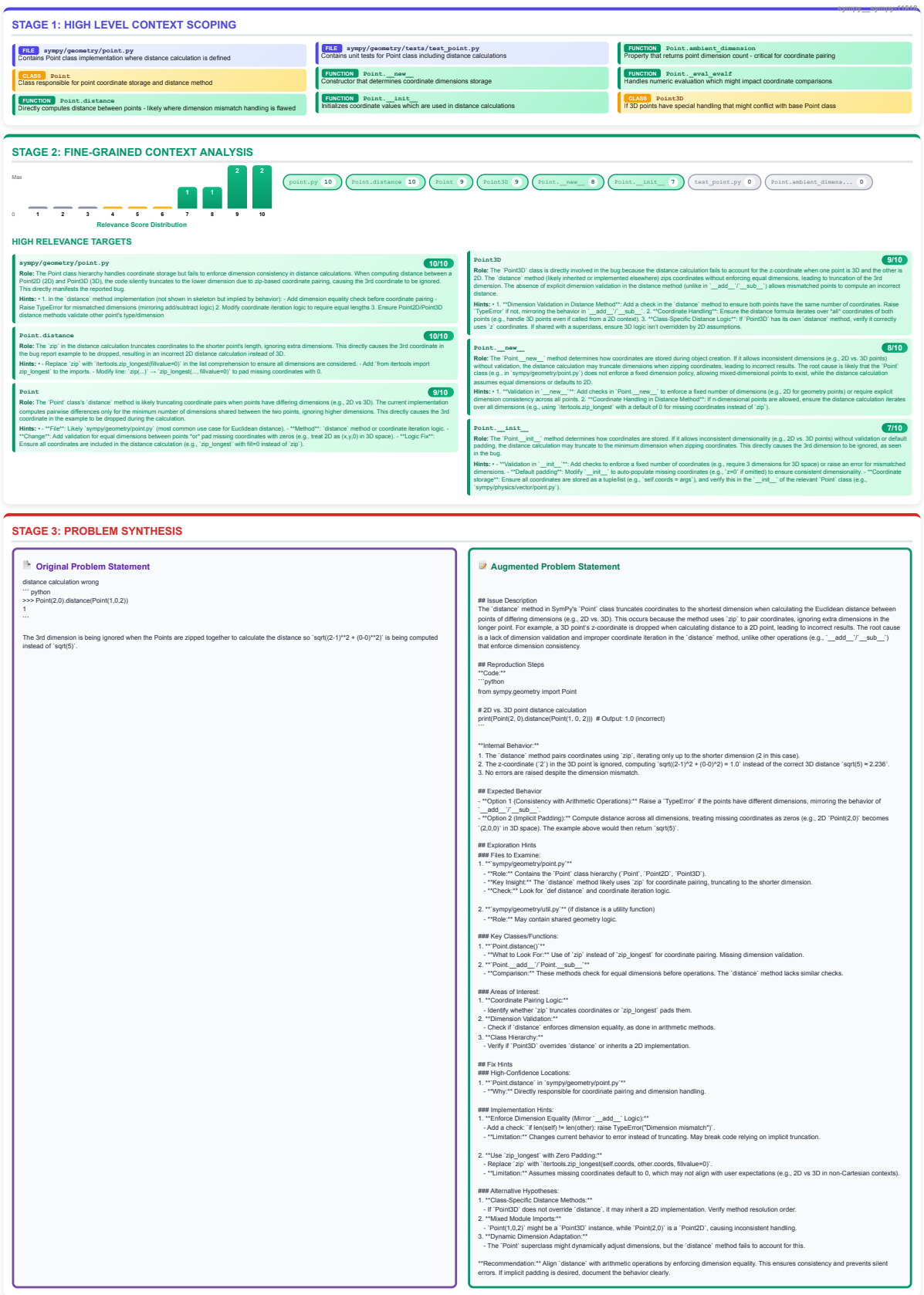


Figure 21: CodeScout visualization for `sympy__sympy-11618` with DeepSeek-R1.

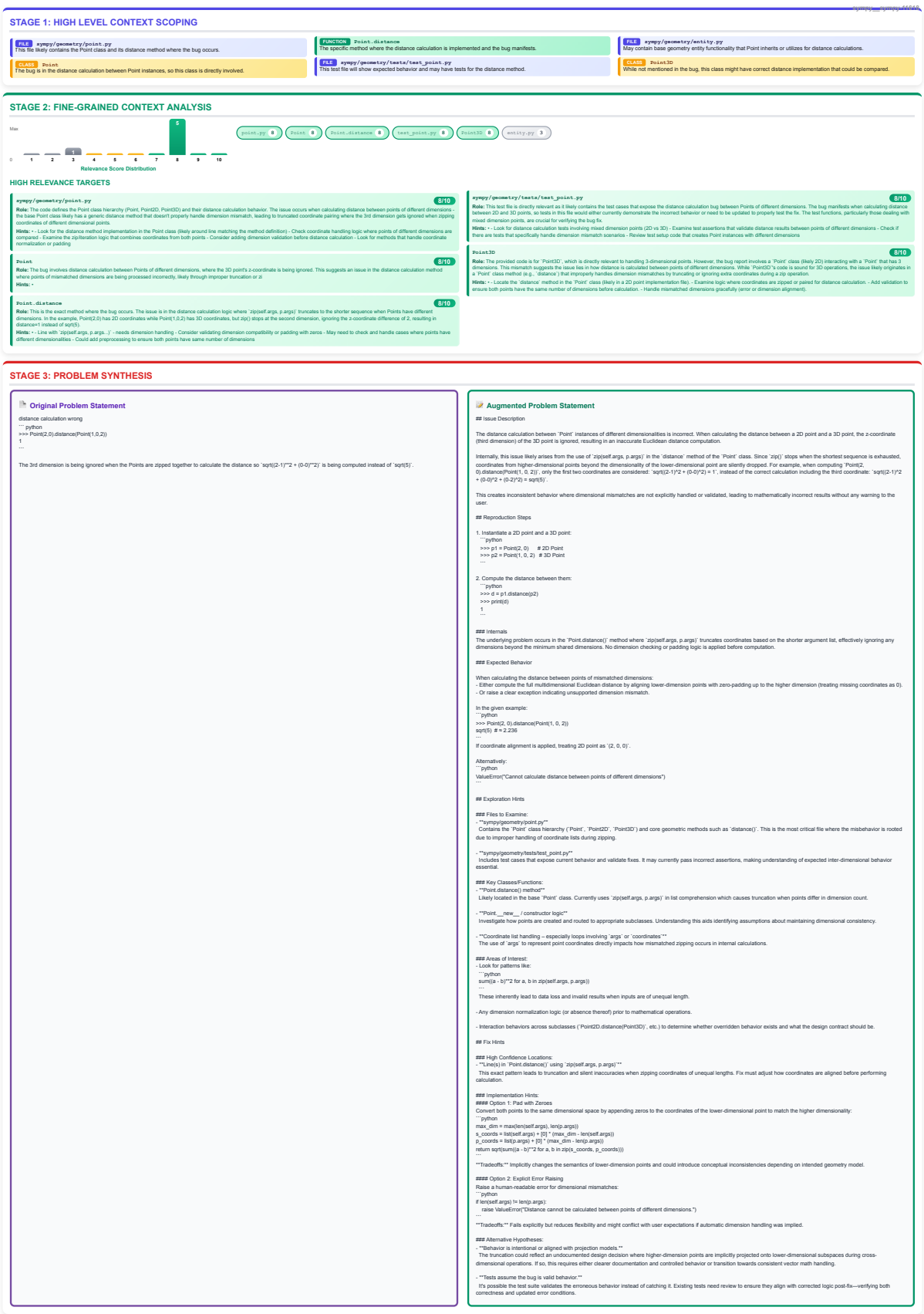


Figure 22: CodeScout visualization for `sympy__sympy-11618` with Qwen3-Coder.

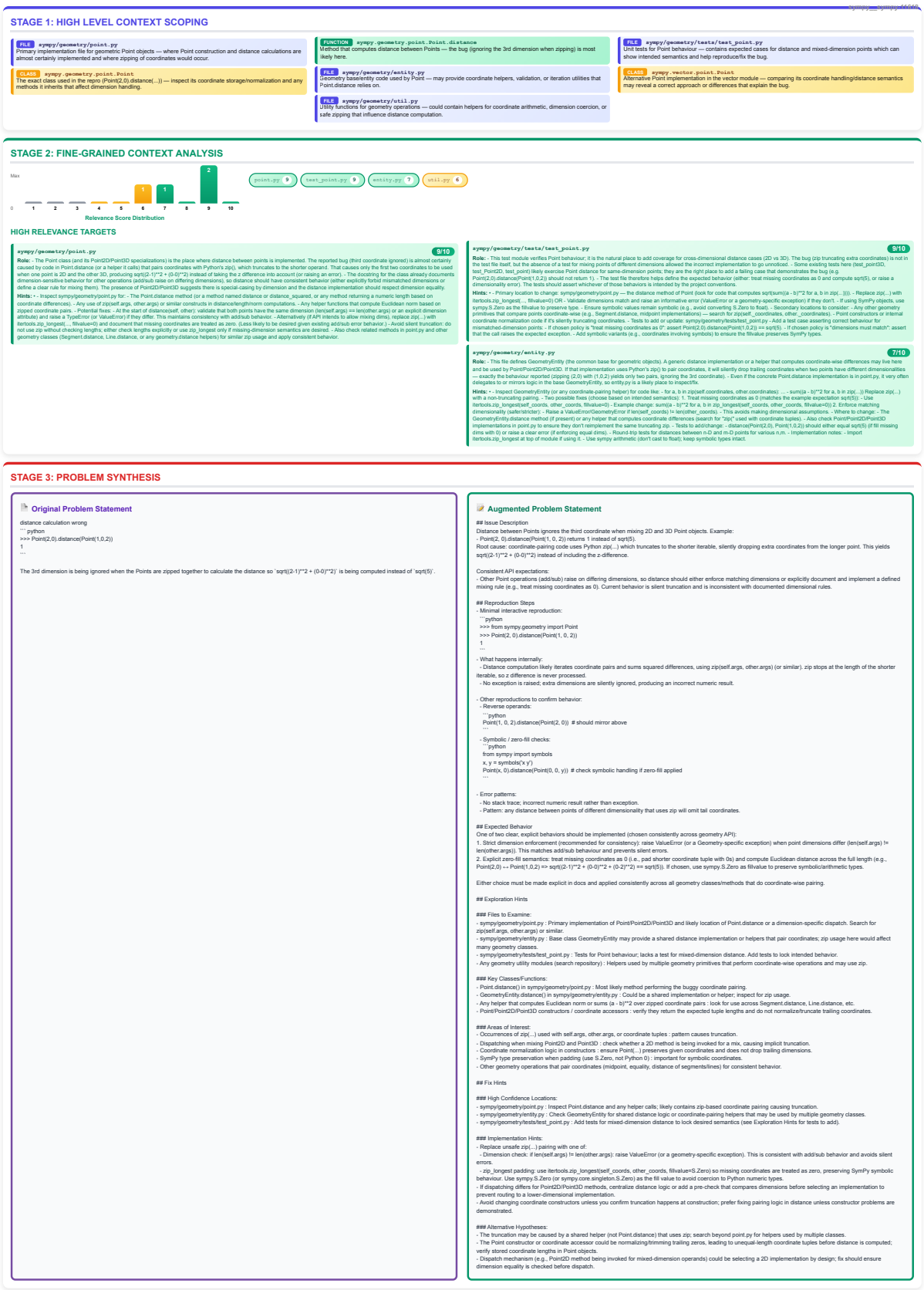


Figure 23: CodeScout visualization for `sympy__sympy-11618` with GPT-5-mini.

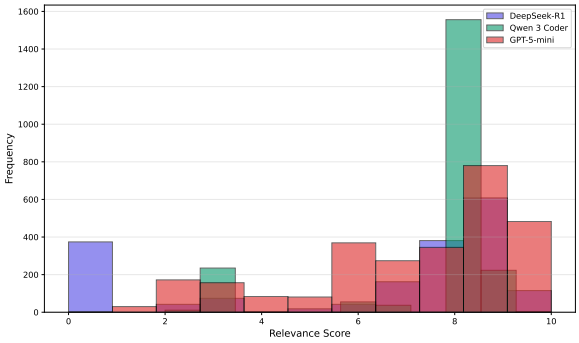


Figure 24: CodeScout relevance score distribution across all three methods.

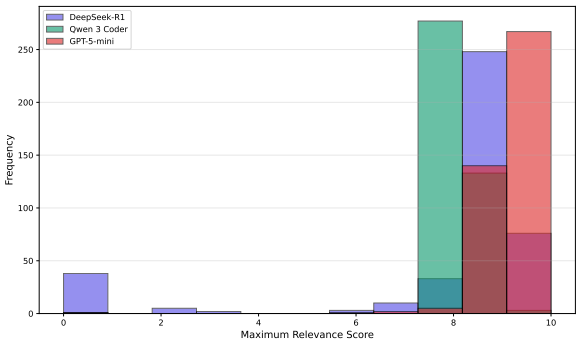


Figure 25: CodeScout maximum relevance score distribution per instance.

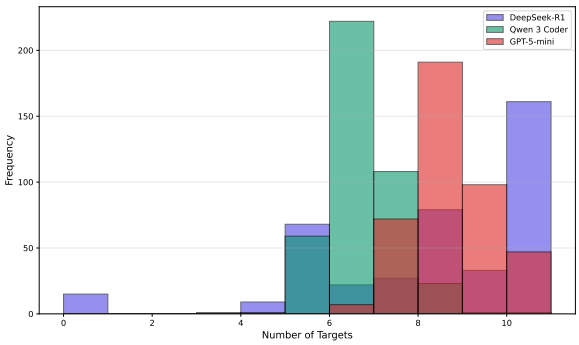


Figure 26: CodeScout number of targets distribution per instance.

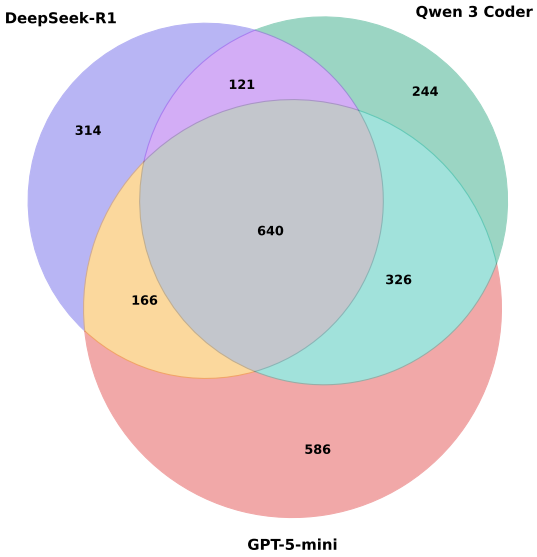
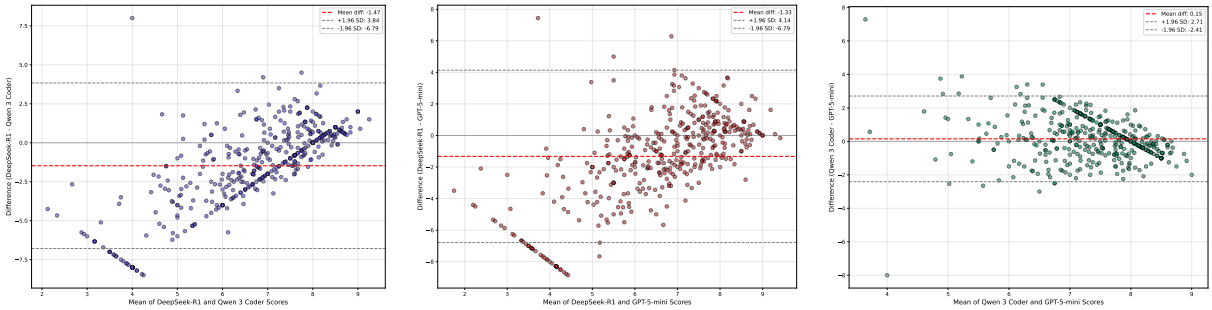
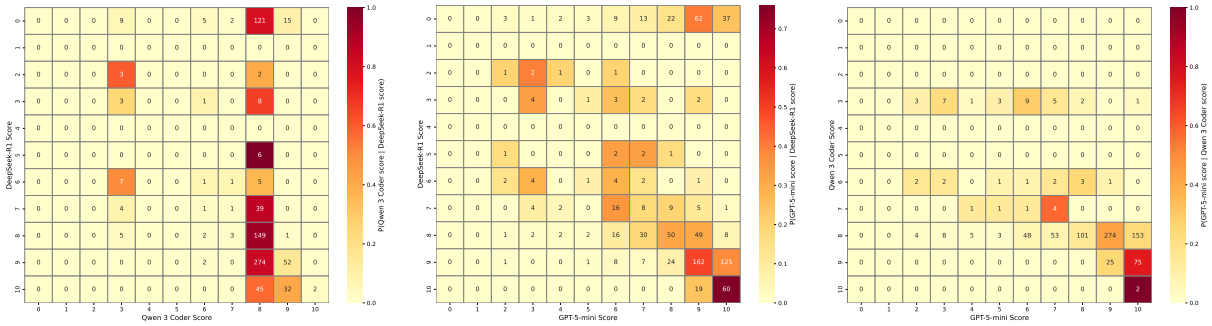


Figure 27: CodeScout target coverage: Venn diagram showing unique and shared exploration targets.



(a) DeepSeek-R1 vs Qwen3-Coder (b) DeepSeek-R1 vs GPT-5-mini (c) Qwen3-Coder vs GPT-5-mini

Figure 28: CodeScout agreement analysis: Bland-Altman plots comparing average scores between LLMs.



(a) DeepSeek-R1 vs Qwen3-Coder (b) DeepSeek-R1 vs GPT-5-mini (c) Qwen3-Coder vs GPT-5-mini

Figure 29: CodeScout score agreement heatmaps between methods for common targets.