Automated Issue Report Generation from Uncovered Code Segments

Diany Pressato

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Software Engineering) at

Concordia University

Montréal, Québec, Canada

April 2026

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Diany Pressato**

Entitled: **Automated Issue Report Generation from Uncovered Code Segments**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Zhijie Wang

_____ Examiner
Dr. Tse-Hsun Chen

_____ Supervisor
Dr. Shin Hwei Tan

Approved by _____
Dr. Denis Pankratov, Chair
Department of Computer Science and Software Engineering

_____ 2026

_____
Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Automated Issue Report Generation from Uncovered Code Segments

Diany Pressato

Developers are increasingly overwhelmed by AI-generated issue reports that lack actionability and reproducibility, eroding trust in automated bug detection tools. In this paper, we present IssueSpecter, an automated tool that finds bugs in uncovered code segments and generates prioritized, actionable issue reports. IssueSpecter combines coverage analysis with LLM-based defect identification, producing structured reports complete with severity ratings, reproduction steps, and suggested fixes. Our evaluation on 13 actively maintained Python projects generates 10,467 issue reports across all uncovered code segments, from which a rule-based ranking orders all issues per project. From this initial rule-based ranking, we filter the top 10 highest-severity, most descriptive issues per project, resulting in 130 candidate reports that are subsequently re-ranked by an LLM according to impact, scope, and urgency. Our manual investigation of these 130 issues confirms that 84.6% are valid or warrant further investigation, with only 15.4% classified as false positives. Furthermore, the LLM-based ranking outperforms rule-based ranking by 50% at P@3 and 41% in MRR. The identified bugs span a wide variety of types, from logic and boundary errors to security vulnerabilities and state consistency bugs. We further validate IssueSpecter through case studies reproducing real bugs surfaced from its generated reports, demonstrating its practical value for automatic bug discovery in real open-source Python projects. Furthermore, IssueSpecter achieves a higher valid bug rate in relation to CoverUp.

Acknowledgments

I would like to thank my advisor, Professor Shin Hwei Tan, for her immense support, guidance, and creativity. She genuinely cared about my research and always kept our research group close and united. I am also grateful for her dedication as a professor during my Master's and for all of our warm and enjoyable conversations.

I also thank Professor Dr. Zhijie Wang for serving as committee chair and examiner, and Professor Dr. Tse-Hsun (Peter) Chen for being an examiner, a great professor during my Master's and from whom I also learned how to bring out the best in students while being his Teaching Assistant.

Thanks to Honghao and Mariam for their research insights, and for the endless support I received from my husband. I am also grateful to my friends and all the professors and advisors I have had throughout my life for being part of the knowledge and person I am today, and last, I am grateful for the small everyday things that made this experience a pleasant one.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software projects continuously evolve, and despite best efforts in testing, a significant portion of code remains unexercised by existing test suites. These uncovered code segments are more likely to hold silent bugs that escape detection. Meanwhile, as AI-generated outputs become increasingly prevalent in software engineering workflows, establishing developer trust in those outputs has emerged as a challenge (Roychoudhury, Pasareanu, Pradel, & Ray, 2025). In the context of automated issue generation, this challenge is compounded by reports that lack actionability and reproducibility, the properties that developers need to evaluate and act upon AI-generated outputs. Thus, the challenge is generating issue reports that are prioritized by impact and immediately actionable, lowering the barrier for developers to triage the surfaced bugs that are prioritized by impact.

To the best of our knowledge, our approach is the first automated pipeline to combine coverage-guided segment identification with LLM-based defect detection to generate prioritized, structured issue reports from uncovered code segments. To address these challenges, we present IssueSpecter, a tool that automatically identifies bugs within uncovered code segments, undetected by existing test suites. Our tool combines code coverage analysis and heuristics-based ranking with LLM-based defect identification to automatically find probable bugs in uncovered code segments and generate complete structured issue reports alongside its severity ratings, reproduction steps, and suggested fixes. IssueSpecter generates issue reports while also ranking them by priority, so that developers are presented

with the most impactful probable bugs first, rather than an unordered flood of candidates.

A fundamental gap persists between proactive testing and automated resolution. Proactive Automated Test Generation (ATG) tools (Altmayer Pizzorno & Berger, 2025; Lukasczyk & Fraser, 2022; Schäfer, Nadi, Eghbali, & Tip, 2024a) increase code coverage but suffer from the oracle problem, in which a generated test that reaches a buggy path often encodes the incorrect behavior into its assertions, masking the defect rather than exposing it. Furthermore, the standard practice of filtering failing tests (Pan et al., 2024) systematically discards the very signals that expose validation failures, meaning coverage-driven suites may appear comprehensive while silently masking latent vulnerabilities. Reactive Issue Resolution agents (Adapa, A R K, Rahim, & Victor, 2025; S. Chen et al., 2026), on the other hand, possess the reasoning capacity to fix bugs but lack an activation signal for uncovered code, since they can only act once a human has already identified and submitted a bug report. Issue reports bridge this gap more effectively than generated tests: rather than producing an artifact that requires deep codebase familiarity to evaluate, and that provides no intrinsic guarantee of correct behavior, issue reports describe potential defects in natural language, provide reproduction steps, and suggest fixes, making them immediately interpretable and actionable, and serving as the precise activation signal that resolution agents need to operate on uncovered code.

While prior work has explored LLMs for test generation as a means of improving code coverage (Altmayer Pizzorno & Berger, 2025; Lemieux, Inala, Lahiri, & Sen, 2023), this does not imply that the generated tests are easily understood by developers, and does not directly imply that the generated tests have sufficient quality. Moreover, we argue that generating issue reports offers a more human-readable alternative for surfacing bugs in uncovered code. When an LLM generates a unit test, the resulting artifact is only meaningful to a developer who already understands the codebase structure, project conventions, and intended usage patterns in a context that is often non-trivial to acquire. Furthermore, a generated unit test provides no intrinsic guarantee of correctness from the human reviewer's perspective: a test may pass while asserting the wrong behavior, and without deep familiarity with the code, a developer has no straightforward way to judge whether the generated

test is actually exercising the intended functionality or simply encoding an existing bug. Issue reports, by contrast, are immediately interpretable: they describe a potential defect in natural language, provide reproduction steps, and suggest fixes, requiring no execution or deep codebase familiarity for a developer to evaluate and act upon. This makes IssueSpecter issue generation a more transparent and actionable complement to existing test suites, particularly for uncovered code segments where the absence of tests makes behavioral expectations unclear.

We evaluate IssueSpecter on 13 actively maintained open-source Python projects from the CodaMosa dataset, spanning diverse domains including web frameworks, automation platforms, developer tools, and data utilities. Our study aims to answer the following research questions:

RQ1: How effective is LLM-based issue report generation at detecting real bugs in uncovered code segments?

RQ2: Can LLM-based ranking further improve rule-based ranking for bug report prioritization?

RQ3: What types of bugs does IssueSpecter detect when analyzing uncovered code segments? Are these types diverse? How does project size and execution model affect the bug type distribution surfaced by IssueSpecter?

RQ4: Does test coverage level influence the quality of generated issues?

Our results show that 84.6% of top-ranked issues are valid or warrant further investigation, with only 15.4% manually classified as false positives. LLM-based ranking outperforms rule-based ranking by 50% at P@3 and 41% in MRR, with the surfaced probable defects spanning a diverse range of types across all nine categories of the taxonomy proposed in BugPilot (Sonwane et al., 2025), from logic errors and boundary conditions to security vulnerabilities and state consistency bugs. We validate IssueSpecter through case studies reproducing real-world bugs surfaced from its generated reports, including a path traversal vulnerability (CWE-22) and a silent data loss bug in a gzip decompressor.

Contributions. Our contributions are summarised as follows:

Tool: We present IssueSpecter, a novel automated pipeline that combines coverage-guided segment identification with LLM-based defect detection to generate prioritized, structured issue reports from uncovered code segments in Python open-source projects.

Study: We conduct an evaluation on 13 open-source Python projects, generating 10,467 issue reports and manually annotating 130 top-ranked issues, showing that 84.6% are valid or warrant further investigation. We show that LLM-based ranking consistently outperforms rule-based ranking across all evaluated metrics overall.

Case Studies: We validate IssueSpecter through selected case studies reproducing real bugs across diverse bug types, demonstrating its practical value for surfacing bugs in open-source Python projects.

Chapter 2

Literature Review

This chapter reviews the body of work most relevant to IssueSpecter, organized into five themes. Section 2.1 covers LLM-powered test generation and automated reporting, establishing the coverage-driven foundations that motivate our approach. Section 2.2 examines static bug detection techniques, both traditional and LLM-based, situating IssueSpecter's ranking design within this landscape. Section 2.3 discusses bug reproduction and automated issue resolution, reviewing how LLMs have been applied to structured bug report generation and software maintenance tasks. Section 2.4 presents the bug taxonomy adopted for classifying defects surfaced by IssueSpecter. Finally, Section 2.5 surveys automated code review research, highlighting the distinctions between review-oriented defect flagging and IssueSpecter's proactive, coverage-guided issue generation.

2.1 LLM-Powered Test Generation and Automated Reporting

Several techniques have been proposed to improve software test generation and automated reporting quality. Search-based software testing (SBST) approaches explore the input space to maximize code coverage, but often stall when faced with complex or uncovered code regions. To enhance the effectiveness of SBST, LLM-powered techniques have been proposed to complement it by generating tests targeting under-covered code, such as

CodaMosa (Lemieux et al., 2023) and CoverUp (Altmayer Pizzorno & Berger, 2025), as well as multi-agent frameworks that automate the entire testing workflow, from test generation to structured reporting (Sherifi, Slhoub, & Nembhard, 2025). More recently, LLM-based techniques have been proposed to automate the generation and improvement of structured, actionable issue reports directly from source code, including approaches based on instruction fine-tuning (Acharya & Ginde, 2025) and multi-agent pipelines (Choi & Yang, 2025).

CodaMosa is a hybrid search-based and LLM-driven test generation tool for Python that addresses coverage plateaus in standard SBST approaches (Lemieux et al., 2023). CodaMosa monitors Pynguin's (Lukasczyk & Fraser, 2022) search progress and, when coverage stalls, queries Codex to generate test cases targeting under-covered functions, which are then deserialized back into Pynguin's internal representation and used to re-seed the search. Evaluated across 486 Python modules drawn from 27 open-source projects, the same dataset adopted by IssueSpecter, CodaMosa achieves significantly higher coverage than pure SBST on 173 benchmarks and than a Codex-only baseline on 279 benchmarks. The benchmark suite itself, derived from the evaluation of Pynguin (Lukasczyk & Fraser, 2022) and BugsInPy (Widyasari et al., 2020), represents a challenging corpus of real-world Python code on which most SBST tools struggle to achieve high coverage. IssueSpecter repurposes this same corpus not for test generation but also for defect discovery: the code segments that CodaMosa targets to improve coverage are precisely the segments that IssueSpecter inspects for latent bugs in the most recent versions of these projects, making the two tools complementary perspectives on the same underlying problem of uncovered code.

CoverUp is a coverage-driven approach to LLM-based test generation for Python that is directly relevant to IssueSpecter's core premise (Altmayer Pizzorno & Berger, 2025). CoverUp first measures the coverage of any existing test suite using SlipCover (Altmayer Pizzorno & Berger, 2023), identifies functions and methods that lack line or branch coverage via Abstract Syntax Tree (AST) analysis, and iteratively prompts an LLM with focused excerpts of the uncovered code, embedding precise coverage information, and allowing the model to request additional context via a tool function. If generated tests fail or do not

improve coverage, CoverUp continues the dialogue with targeted feedback, requesting corrections before accepting any test. Evaluated across a benchmark of 425 Python modules derived from 35 open-source projects, CoverUp achieves a per-module median line+branch coverage of 80%, compared to 47% for CodaMosa (Lemieux et al., 2023) and 77% overall for MuTAP (Dakhel, Nikanjam, Majdinasab, Khomh, & Desmarais, 2024), with an ablation study confirming that coverage-based prompting, code context, and iterative error correction each contribute meaningfully to this result. CoverUp establishes that LLMs can be effectively directed toward uncovered code when given precise coverage signals, the same foundational insight that motivates IssueSpecter, which repurposes coverage analysis not to generate tests but to identify the uncovered segments most likely to contain latent defects, and tasks an LLM with detecting and reporting those defects rather than exercising them.

A previous work (Sherifi et al., 2025) proposes a multi-agent framework that leverages LLMs to automate the full software testing workflow, from unit test generation through to structured reporting. The framework comprises four core components: a web client for capturing user input, a Software Testing Agent that coordinates the overall workflow, an LLM backend responsible for entity extraction, test generation, and call graph construction, and a local development environment in which tests are executed. Upon receiving a user prompt, the system extracts project entities, locates the relevant source files, generates unit tests alongside their rationale using an LLM, and produces a comprehensive PDF report detailing test results, code coverage, and a call graph visualization. Evaluations across four applications, two in Python and two in Java, demonstrate consistently high test coverage, averaging 97.71% for Java projects and 93.45% for Python projects, with average end-to-end execution times of 86.7 and 80 seconds respectively. Notably, the bulk of execution time (approximately 62.8 seconds on average) is consumed by the LLM test generation step. A comparison of Gemini and ChatGPT as backend LLMs on the LibrarySystem project shows that ChatGPT achieves 98% coverage but requires approximately twice the generation time. Identified limitations include occasional failures in Java due to ambiguous prompts and compilation errors in generated scripts, as well as the framework's current

restriction to unit testing, with integration and system-level testing remaining unexplored. This work is relevant to IssueSpecter in two respects: (1) by confirming the viability of LLM-driven automated test generation as a complement to coverage-guided defect discovery, the high coverage rates achieved by Sherifi et al. (2025)'s framework underscore that LLMs can reliably exercise code paths, while IssueSpecter targets precisely those paths that existing test suites fail to reach; (2) the structured PDF reporting pipeline (Sherifi et al., 2025) parallels IssueSpecter's own commitment to generating actionable, structured issue reports, including severity ratings, reproduction steps, and suggested fixes, rather than raw defect signals, reinforcing the broader trend toward LLM-assisted, human-readable software quality artifacts.

2.2 Static Bug Detection: Traditional and LLM-Based Approaches

Static code analysis, the examination of source code without execution, has long served as a foundational technique for preemptively identifying coding errors and vulnerabilities before deployment. Traditional tools such as FindBugs (Ayewah, Pugh, Hovemeyer, Morgenthaler, & Penix, 2008), PMD (AlOmar, AlOmar, & Mkaouer, 2023), Coverity (Imtiaz, Murphy, & Williams, 2019), ESLint (Tómasdóttir, Aniche, & Van Deursen, 2020), and PyLint (Mohialden, Mahmood Hussien, Baker, & Joshi, 2023) share a common set of strengths and limitations: tools tailored to a single language offer precise detection of language-specific patterns, and rule-based approaches excel at well-defined issues, but struggle with complex or context-dependent bugs. False-positive management remains a persistent challenge. ESLint achieves relatively low false-positive rates while PMD faces challenges with redundant or incorrect alerts, and scalability to large codebases varies considerably across tools. More recent LLM-based approaches have addressed these limitations, with false-positive reduction as a standout benefit. The use of ChatGPT-4 to augment Infer (Mohajer et al., 2024) improves precision for Null Dereference bugs by 12.8% and for Resource Leak bugs by 43.13%. LLM4FPM (J. Chen et al., 2025) achieves an F1 score

above 99% and reduces false-positives by over 85%. A hybrid interleaving approach proposed by Chapman, Rubio-González, and Thakur (2024) improves recall from 52.55% to 77.83% while maintaining comparable precision, holding an F1-score of 0.804. The IRIS framework (Z. Li, Dutta, & Naik, 2025) doubles the detection rate of CodeQL and reduces false discovery rates by 5 percentage points. LLift integrated with UBITect (H. Li, Hao, Zhai, & Qian, 2024) and PYSIASSIST (Hong, Sun, Gao, & Tan, 2024) further demonstrate the value of LLM-based contextual reasoning for detecting subtle bugs that rule-based tools miss. However, these approaches share common limitations: computational overhead due to frequent LLM calls, context window constraints that hinder inter-procedural analysis, prompt sensitivity, and limited generalizability across languages and projects. The evidence collectively suggests that LLM-based tools outperform traditional ones in detection accuracy through superior contextual understanding, but that their computational cost means traditional tools retain practical value, and that combining both represents the most promising direction. This is directly pertinent to IssueSpecter, which combines a lightweight rule-based pre-ranking stage with an LLM-based re-ranking stage, seeking precisely this balance between efficiency and contextual reasoning.

2.3 Bug Reproduction and Automated Issue Resolution

Large language models have demonstrated strong potential across a range of software maintenance tasks, including test generation (Schäfer et al., 2024a) and automated program repair (Sobania, Briesch, Hanna, & Petke, 2023). Their application to bug-related tasks has gained particular attention in recent years.

A convolutional neural network-based approach to classify bug reports as valid or invalid using only textual data was proposed (He et al., 2020), while other related work (X. Chen, He, Li, He, & Chen, 2018) employed natural language processing and quantifiable indicators to assess bug report quality automatically. Building on these efforts, the CTQRS framework was proposed (Zhang, Zhao, Yu, & Chen, 2022), a bug-report quality assessment approach that systematically scores reports by combining morphological, relational, and analytical

indicators through dependency parsing, evaluating properties such as atomicity, completeness, conciseness, understandability, and reproducibility. Morphological indicators capture structural and linguistic aspects such as size, readability, and punctuation; relational indicators examine whether each standard field is properly provided, including itemization, environment information, and screenshots; and analytical indicators tap into deeper semantics by assessing how clearly a report describes interface elements, user behavior, and system defects. IssueSpecter differs from these approaches by not requiring existing bug histories, developer commits, or manually curated defect datasets. Instead, it targets uncovered code segments identified through coverage analysis, a systematic blind spot in existing test suites, and employs an LLM to detect latent defects and generate structured, actionable issue reports with severity ratings, reproduction steps, and suggested fixes.

Instruction fine-tuned LLMs can automatically transform casual, unstructured bug reports into high-quality, structured ones adhering to a standard template (Acharya & Ginde, 2025). This study evaluates three open-source instruction-tuned LLMs, Qwen 2.5, Mistral, and Llama 3.2, against ChatGPT-4o, measuring performance using CTQRS, ROUGE, METEOR, and SBERT. To construct the training dataset, 15,000 fixed and closed bug reports were mined from Bugzilla, filtering for reports containing steps to reproduce, expected behavior, actual behavior, and additional information, and further removing those with stack traces or code snippets and those scoring below 14 on CTQRS, yielding 3,966 high-quality reports. Since no large-scale paired dataset of unstructured and structured reports existed, synthetic unstructured summaries were generated using Llama 3 with a carefully designed prompt, retaining only those with SBERT similarity above 85% and cosine similarity above 80%, resulting in 3,903 training pairs. Models were fine-tuned using Low-Rank Adaptation (LoRA) via the Unsloth framework, targeting attention projection layers with a rank of 16, trained for 3 epochs with 4-fold cross-validation. Fine-tuned Qwen 2.5 achieves a CTQRS score of 77%, outperforming fine-tuned Mistral (71%) and Llama 3.2 (63%), and matching ChatGPT in 3-shot learning (75%); both Qwen and Mistral also surpass ChatGPT on ROUGE-1, with scores of 0.64 and 0.62 respectively against 0.44. The study further demonstrates cross-project generalizability, with fine-tuned models achieving up to 70%

CTQRS on unseen projects from diverse ecosystems such as Eclipse and GCC, highlighting the potential of instruction fine-tuning as a cost-effective, privacy-preserving alternative to proprietary models for structured bug report generation. With respect to missing information detection, fine-tuned Llama 3.2 model marginally outperforms the others in flagging absent fields, particularly `Actual Results` and `Expected Results`, while Qwen 2.5 and Mistral tend to infer missing content from context rather than explicitly flagging it, and correctly identify the missing Steps to Reproduce section in over 70% of cases. In contrast to IssueSpecter, this work focuses on restructuring existing human-written reports rather than generating issue reports automatically from source code.

AgentReport (Choi & Yang, 2025) was built directly on this line of work, being a multi-agent LLM pipeline that addresses the modularity, reproducibility, and scalability limitations of single-pipeline approaches such as that of Acharya and Ginde (2025). AgentReport integrates QLoRA-4bit lightweight fine-tuning, CTQRS-based structured prompting, Chain-of-Thought (CoT) reasoning, and FAISS-based one-shot exemplar retrieval within seven sequentially coordinated modules: Data, Prompt, Fine-tuning, Generation, Evaluation, Reporting, and Controller. The Data Agent preprocesses and partitions the same 3,966 Bugzilla summary report pairs into an 8:1:1 train/validation/test split with a fixed random seed. The Prompt Agent constructs inputs by combining CTQRS-structured instructions enforcing seven mandatory sections: summary, steps to reproduce, expected result, actual result, environment, evidence, and additional information, with a CoT self-check directive and a one-shot example. The Fine-tuning Agent applies QLoRA-4bit to Qwen2.5-7B-Instruct, and the Generation Agent employs deterministic decoding (temperature = 0) to ensure reproducible outputs. On the 3,966 Bugzilla pairs, AgentReport achieves 80.5% CTQRS, 84.6% ROUGE-1 Recall, 56.8% ROUGE-1 F1, and 86.4% SBERT, improving over the single-pipeline baseline by +3.5 percentage points in CTQRS, +23.6 points in Recall, and +1.4 points in SBERT, and outperforming GPT-4o in both 0-shot and 3-shot settings across all metrics. An ablation study confirms that each component contributes meaningfully: removing QLoRA fine-tuning causes the largest drop (F1 from 56.8 to 24.9; SBERT

from 86.4 to 82.5), removing one-shot retrieval reduces Recall from 84.6 to 70.0, and removing CTQRS prompting reduces structural completeness from 80.5 to 76.9. The study notes that the relatively modest ROUGE-1 F1 score reflects a design trade-off inherent to CTQRS-guided generation, which prioritises exhaustive coverage and completeness over conciseness, and identifies verbosity reduction as the primary target for future improvement. While AgentReport advances the quality of restructured reports, it similarly operates on pre-existing human-submitted reports, whereas IssueSpecter generates structured issue reports directly from uncovered code segments without relying on any prior bug history.

More broadly, prior work has employed LLMs to generate missing information in bug reports using ChatGPT (Bo et al., 2024), though ChatGPT has been found to generate incorrect information (Tanzil, Khan, & Uddin, 2024) and raises data privacy concerns in software engineering contexts (Cai et al., 2024). These limitations motivate the use of open-source, locally deployable models for bug-related generation tasks.

In the context of software engineering agents, BugPilot (Sonwane et al., 2025) introduce a pipeline for synthetic bug generation that leverages SWE agents to produce naturalistic bugs through realistic development workflows. Rather than intentionally injecting faults, BugPilot tasks agents with implementing new features in existing repositories, recording the resulting test failures as bugs, a process that more closely mirrors authentic development scenarios. Through qualitative and quantitative analysis, the study demonstrates that unintentionally generated bugs are more diverse and challenging for current frontier models than synthetically injected ones, providing more efficient training data for supervised fine-tuning and reinforcement learning. Unlike IssueSpecter, which targets latent defects in existing uncovered code, BugPilot synthesizes new bugs as a data generation mechanism rather than surfacing pre-existing ones.

An empirical study investigates what makes developer–ChatGPT conversations effective for software issue resolution, analysing 686 conversations shared within GitHub issue threads (Ehsani, Pathak, Parra, Haiduc, & Chatterjee, 2025). The study finds that only 62% of these conversations are helpful, with ChatGPT performing best on well-scoped tasks such as code generation and tool/library recommendations, but struggling with complex,

project-specific problems such as system-level debugging and refactoring. At the conversational level, helpful interactions are shorter, more readable, and exhibit higher semantic alignment, while unhelpful ones tend to be verbose, topic-shifting, and lacking sufficient context. The identified deficiencies in unhelpful responses, specifically incorrect information and lack of comprehensiveness, motivate IssueSpecter's validity assessment and ranking stages, which filter and prioritise LLM-generated issue reports rather than accepting them uncritically.

A complementary and directly relevant strand of research empirically evaluates the bug detection capabilities of LLMs when applied directly to source code, rather than to bug reports. A systematic evaluation of ChatGPT-4, Claude 3, and LLaMA 4 was conducted on a stratified benchmark spanning foundational programming errors, classic security vulnerabilities, and advanced production-grade bugs in C++ and Python (Mhatre, Nader, Diehl, & Gupta, 2026). The dataset integrates C++ snippets from SEED Labs and the OpenSSL project, and Python bugs from PyBugHive, a repository of manually validated bugs in NumPy and Pandas, with all samples locally compiled and executed to confirm reproducibility. A multi-stage, context-aware prompting protocol is used to simulate realistic debugging workflows, and a graded rubric assesses detection accuracy, reasoning depth, and remediation quality across five levels from no detection to complete detection. The study findings reveal a consistent pattern directly pertinent to IssueSpecter's design assumptions: for foundational bugs (null pointer dereferences, use-after-free errors, buffer overflows and pass-by-value semantics), all three LLMs demonstrate strong diagnostic capability, correctly identifying the nature, location, and remedy of errors and aligning with modern C++ best practices. Performance diverges markedly for more complex scenarios: ChatGPT-4 and Claude 3 consistently deliver richer contextual analyses, identifying chained exploitation paths, privilege boundary violations, and subtle issues such as numeric precision loss from casting `ULONG_MAX` to `double`, while LLaMA 4 frequently produces partial or surface-level explanations and misses subtler aspects such as file descriptor management or privilege context. For advanced real-world bugs from OpenSSL, LLMs struggle with

cross-abstraction reasoning, cryptographic correctness, state management, and API contract violations, and none of the models successfully proposed the correct minimal fix for algorithmic bugs such as the spiral matrix pattern on first pass. These findings provide empirical grounding for IssueSpecter's choice of GPT-class models as the LLM backend for defect identification: the demonstrated superiority of ChatGPT-4 and Claude 3 over LLaMA 4 in contextual reasoning about production-grade bugs supports the use of frontier models for generating valid and actionable issue reports from uncovered code segments.

Automated bug detection has been approached through a variety of paradigms. Traditional strategies mine real bugs from pull requests and commits in open-source repositories, requiring careful issue localisation and filtering. Alternatively, synthetic bug generation injects faults into existing codebases, enabling researchers to scale data without being constrained by the availability of real-world development data. SWE-smith (Yang et al., 2025), is a scalable pipeline for automatically generating software engineering training data from real Python codebases. Given any repository, SWE-smith constructs an execution environment and synthesizes task instances using four strategies: LM-based function modification and rewriting, procedural AST transformations, PR inversion, and bug combination. Candidates are validated by execution: only those that break one or more existing passing tests are retained, and LM-generated issue descriptions are automatically produced for each surviving instance. Applied to 128 GitHub repositories, SWE-smith produces 50,000 task instances at a fraction of the storage cost of prior datasets, enabling the training of SWE-agent-LM-32B which achieves 40.2% on SWE-bench Verified, a new open-weight state of the art. SWE-smith is relevant to IssueSpecter as a contrasting paradigm: where SWE-smith synthesizes bugs by deliberately perturbing existing code until tests break, IssueSpecter takes the complementary approach of identifying code that existing tests never reach and using an LLM to reason about latent defects already present, without requiring ground-truth bug injection.

2.4 Bug Taxonomy

To classify the defects surfaced by IssueSpecter, we adopt the bug taxonomy introduced in the context of BugPilot (Sonwane et al., 2025), a framework for synthetic bug generation via software engineering agents. The taxonomy comprises nine categories: Logic and Conditional Bugs, Input Validation and Boundary Handling Errors, State Consistency and Bookkeeping Bugs, Protocol and Specification Conformance Bugs, API and Signature Mismatch or Backward Compatibility Breaks, Incorrect Argument Forwarding and Inheritance Contract Breaks, Security and Sensitive Data Leakage Due to Logic Oversight, I/O and Filesystem Handling Bugs, and Copy Semantics and Mutability Aliasing Bugs. This taxonomy was derived through hierarchical summarisation of bugs from multiple datasets, including human-authored edits and synthetic collections, and has been shown to reflect the distribution of defects found in real-world open-source repositories (Sonwane et al., 2025). We adopt it as our annotation scheme to enable consistent classification and cross-project comparison of the issues generated by IssueSpecter.

2.5 Code Review

Traditional code review systems have long been used to identify issues in submitted code, but evidence suggests they focus predominantly on style, readability, and coding standards rather than bug-targeted defect detection. When LLMs generate review comments spanning readability, bugs, maintainability, and code design, developers preferentially resolve readability comments at 43.3% and bug-related comments at 41.9%, while maintainability and code design comments are resolved at lower rates of 36.2% and 28.6% respectively (Goldman et al., 2025). This reveals a mismatch between what LLM-based code review tools tend to produce and what developers actually find actionable; despite developers showing a clear preference for resolving readability and bug-related feedback, most LLM-based code review approaches predominantly flag maintainability and code design concerns, which are treated with lower priority in practice (Goldman et al., 2025). Differently, IssueSpecter is bug-targeted, avoiding emphasis on style and coding standards that may divert developer

attention.

A prior study further confirms this trend, finding that the majority of code review benchmarks cover broad quality concerns such as documentation, refactoring, and adherence to coding conventions, with bug detection remaining a secondary focus (Khan, Wang, Zhang, & Chen, 2026). Moreover, automated code review models perform best on syntax issues and code smells rather than on logic errors and security vulnerabilities, which require deeper contextual reasoning (Mo, Li, & Jiang, 2024). This mismatch between what code review tools produce and what developers actually need motivates the design of IssueSpecter: rather than prompting an LLM to generically review code, which risks producing low-priority style-oriented feedback, IssueSpecter explicitly targets bug discovery in uncovered code segments, prompting the model to identify defects and generate structured, actionable reports with reproduction steps and proposed fixes. While previous work applies LLMs to code review (Alami, Jensen, & Ernst, 2025; Goldman et al., 2025; Jin & Chen, 2026; Khan et al., 2026; S. Li et al., 2025; Lu et al., 2025; Mo et al., 2024; Sun et al., 2025), IssueSpecter distinguishes itself by being explicitly bug-targeted, proactively surfacing latent defects in untested code before any human review or code change is involved.

An analysis of which types of LLM-generated review comments developers most frequently resolve shows that 60% to 70% of comments remain unresolved due to insufficient clarity and specificity, and recommends focusing on actionability rather than text-similarity metrics such as BLEU (Goldman et al., 2025). LLM-generated and human code review comments differ in focus but complement each other, with LLM comments, particularly those about readability and bugs, being frequently resolved by developers, showing their practical value. While this work focuses on understanding which types of LLM-generated review comments developers resolve in practice, operating on human-submitted code changes (Goldman et al., 2025), IssueSpecter proactively generates structured issue reports with reproduction steps and fixes from uncovered code segments, without requiring any human-submitted change or reviewer involvement.

The performance of four LLM architectures for automated code defect detection was evaluated on 5,000 samples from 150 open-source repositories across Python, Java, JavaScript,

C++, and Go (Mo et al., 2024). It considers six defect types: security vulnerabilities, logic errors, performance issues, code smells, maintainability violations, and syntax inconsistencies. GPT-4 achieves the highest overall accuracy (78.4%) and excels at security and logic errors; CodeBERT performs best on syntax issues and code smells; GraphCodeBERT is strongest on maintainability; and Claude-3 shows balanced performance with strong explanations. It is worth noting that this work focuses only on evaluating how accurately each model can detect and classify defects that were already annotated in the dataset. Unlike to IssueSpecter, this work focuses only on classifying pre-annotated defects without proposing or validating any fixes, and does not target uncovered code segments.

A multi-role LLM framework with Chain-of-Thought reasoning is deployed in a C++ production environment, achieving a tenfold improvement in balancing bug detection coverage against false alarm rates (Lu et al., 2025), but without generating structured reports or proposed fixes as IssueSpecter.

Recent research has shifted from tasks like impact analysis and visualization to end-to-end peer review automation, which now makes up nearly 60% of datasets (Khan et al., 2026). The study highlights active areas like code revision and automated fix generation, but notes that current benchmarks still rely on static text-matching metrics, missing runtime checks to verify whether fixes compile or pass tests. It calls for dynamic evaluation methods and new benchmarks that include languages like Python and Go, which IssueSpecter helps address by evaluating Python projects and verifying that proposed fixes do not introduce regressions into the existing test suite.

The reliability of LLMs in judging whether code meets a given natural language requirement was studied (Jin & Chen, 2026), with the main finding being a systematic over-correction bias, where models often wrongly reject correct code. To address this, a Fix-guided Verification Filter is proposed, which compares the model's proposed fix with the original code and flips the verdict if the fix does not improve the code, significantly reducing false rejections. The study examines six bug categories: missing logic, redundant logic, operator misuse, variable misuse, value misuse, and function misuse, based on datasets

with injected faults in otherwise correct Python functions. In contrast, IssueSpecter targets naturally occurring latent defects in uncovered code segments of actively maintained open-source projects, without relying on fault injection.

RevAgent enhances code review automation by routing code changes through five specialized agents, each focusing on a different issue category: refactoring, bug fixing, testing, logging and documentation (S. Li et al., 2025). A critic agent then evaluates and selects the most appropriate review comment. RevAgent achieves 21.69% accuracy in bug fixing, significantly higher than previous systems, which rarely exceeded 5%, and unlike IssueSpecter, it reacts to submitted code changes rather than proactively addressing uncovered code and does not generate structured issue reports or propose fixes.

Although LLM-based code review may be used to find bugs, IssueSpecter differs in several important perspectives: (1) it operates *proactively* on uncovered code segments identified through coverage analysis, rather than reactively on code changes submitted for review; (2) it generates complete, structured issue reports including severity ratings, reproduction steps, and suggested fixes, rather than review comments that flag potential problems without proposing validated remediation; (3) it targets latent defects that already exist in untested code, whereas code review systems are triggered by human-submitted changes, meaning that bugs residing in uncovered segments may never be surfaced without a dedicated coverage-guided approach.

Chapter 3

Motivating Example

To motivate the need for a prioritization phase in IssueSpecter's pipeline, consider a concrete scenario with HTTPie, a widely used HTTP client library, having 2,123 source files and a test coverage of 32.2%, with 206 uncovered code segments present in the project, that are left as candidates for issue generation. These uncovered segments are compact in nature, spanning 15 lines on average and up to 111 lines at most, representing self contained blocks of untested logic, and by feeding these segments through our Issue Generation phase, IssueSpecter produces 618 issue reports in total, of which 592 identify potential bugs either in the code or in its documentation. 63 of these generated issues are flagged as having high severity, a number that is far more than a developer could reasonably triage in a single session. Without ranking, a developer might spend time on medium or low-severity reports while missing more critical issues. For example, a path traversal vulnerability (CWE-22) was identified among the generated reports for this project. This vulnerability was confirmed through manual reproduction in our case study in Section 7, and its full generated issue report is presented in Appendix A. Thus, this vulnerability was found in one of those compact uncovered segments, showing that even short untested parts of the code can contain serious bugs.

IssueSpecter addresses this through its two-stage reduction pipeline: a rule-based Repository-level Issue Selection phase that narrows the 618 reports, generated for the HTTPie project, down to the top 10 most important issues per project, followed by an

LLM-based Ranking phase that reorders those 10 by impact, scope, and urgency. For the HTTPie project, the LLM-based ranking achieves a perfect MRR score of 1.00, compared to 0.14 under rule-based ranking alone, placing the highest-priority issue (represented by a path traversal vulnerability), at the top of the ranked list. This example illustrates why the ranking is necessary, because without it, critical bugs risk being overlooked, even in projects with 1,028 unit tests such as HTTPie.

Table 3.1 details the top 10 issues ranked by the rule-based sorting criteria for the HTTPie project, alongside the rankings produced by the LLM and the human annotators. To understand why the rule-based approach is inferior in this case, note that all 10 top-selected HTTPie issues share the same severity level and OS score, as shown in Table 3.1. When these two criteria are tied across all candidates, the rule-based ranking degenerates into a simple word count ordering that is entirely agnostic to the actual impact of each probable bug.

Table 3.1: Ranking comparison for the HTTPie project.

Issue Title	**Priority Score**	**OS Score**	**Word Count**	**Golden Rank**	**Rule Rank**	**LLM Rank**
`session_hostname_to_dirname`: unsanitized inputs leading to path traversal / unsafe filenames	5	100	444	1	7	1
parse_content_type_header wrongly splits on semicolons inside quoted parameter values	5	100	531	2	1	4
Unbounded buffering in `BufferedPrettyStream.iter_body` can cause OOM	5	100	447	3	6	3
Fallback to localhost for hostless URLs causes session collisions for unixsocket URLs	5	100	459	4	4	2

As a result, the path traversal vulnerability in function `session_hostname_to_dirname` (CWE-22) is placed at position 7, simply because its description is shorter than those of

less critical issues ranked above it. The LLM-based ranker, by contrast, correctly elevates this security vulnerability to position 1, in agreement with the golden ranking produced by human annotators, while also correctly attributing low priority to issues such as the `output_file_specified`, demonstrating that LLM-based ranking provides meaningful utility when heuristic criteria alone are insufficient to differentiate among high-severity candidates.

Chapter 4

Methodology

Figure 4.1 shows the IssueSpecter pipeline to automatically generate and prioritize issue reports via a four-phase approach: (1) Coverage-based Code Segment Localization, (2) Issue Generation, (3) Repository-level Issue Selection and (4) LLM-based Ranking.

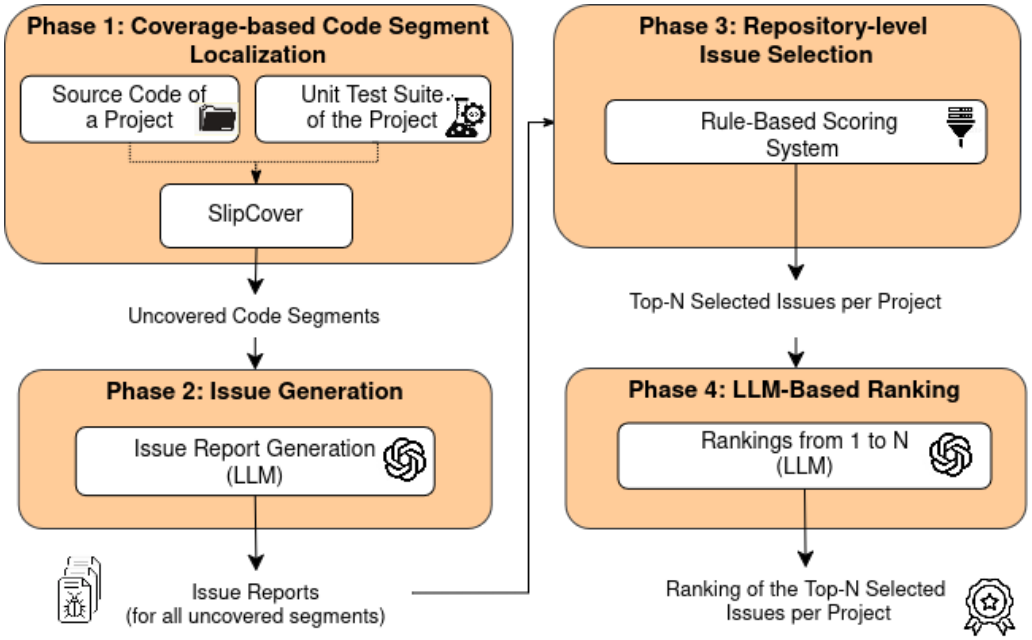

Figure 4.1: IssueSpecter's overall workflow.

Coverage-based Code Segment Localization: Instead of generating issues for all the source code within a given project, our key insight in issue generation is that *uncovered code segments correspond to untested segments that are more likely to contain bugs.* Given the

source code and the corresponding unit test suite of a project, IssueSpecter automatically identifies code segments that are uncovered by existing unit test suite. Specifically, we use SlipCover (Altmayer Pizzorno & Berger, 2023), a code coverage tool for Python that can compute a set of uncovered code with minimal overhead.

Issue Generation:

For each uncovered segment identified in the previous phase, IssueSpecter uses GPT-5-mini to identify potential defects by prompting the model to discover up to three distinct bugs per uncovered segment and generate a corresponding issue report for each. Prompting for multiple issues per segment increases the likelihood of surfacing diverse defect types within the same code block, while the subsequent ranking stage ensures that only the most impactful issues are surfaced to the developer.

Specifically, our prompt described in Figure 4.2 specifies that every generated issue should contain: a severity category (Critical, High, Medium, Low, Very Low, or Minor) assigned by the LLM, the affected operating systems, an in-depth description with step-by-step reproduction instructions, and suggested code fixes for the identified defect.

Role: You are an expert Python developer and test engineer. You will be provided with a Python code snippet extracted from {PROJECT_NAME}.

Task: Analyze the code, identify up to three distinct defects, and generate corresponding bug reports. In the absence of bugs, include a boolean indicator explicitly stating that no bug was found.

For each bug you identify, generate an issue report and provide an associated pull request suggestion. After completing the issue report, organize all found issues according to the precise JSON schema output. For every proposed fix, ensure the code maintains original functionality and style.

For each bug, assess its severity (minor, very low, low, medium, high, or critical) and specify affected operating system(s). Indicate whether documentation inconsistencies are involved and propose a Python code fix when appropriate, preserving original functionality and coding style.

Input: Python code snippet to be analyzed: {CODE_SNIPPET}

JSON Output: a JSON array of three objects, each with a boolean indicating whether a bug is present. If a bug exists, the object includes title, summary, bug severity, OS, generated issue, a boolean for inconsistent documentation, and fixed code, ensuring consistent, parseable output.

Figure 4.2: Simplified prompt template used for automated bug identification and reporting during our Issue Generation phase. The {CODE_SNIPPET} represents the uncovered code segment that is fed as input to the LLM prompt.

Repository-level Issue Selection:

The Issue Generation phase can produce a large number of issue reports across a project's uncovered code segments. With hundreds of issues per project, the goal of this phase is to select a list of important issues considering all uncovered code segments within a project. Specifically, we design a rule-based filtering approach based on prior research on bug report prioritization (Alenezi & Banitaan, 2013) in this phase.

In a broader sense, IssueSpecter can be viewed as an LLM-based approach to bug prediction, identifying code locations likely to contain defects. Rather than replacing traditional bug prioritization techniques, our rule-based ranking stage is explicitly grounded in established criteria from prior work on bug report prioritization (Alenezi & Banitaan, 2013), using severity, platform scope (OS), and description completeness as ranking signals.

Our selection strategy employs a three-step hierarchical ranking mechanism, as summarized in Table 4.1. Issues are ranked according to the following priority order:

Table 4.1: Three-step hierarchical ranking criteria for Repository-level Issue Selection, applied in order of precedence.

Order	Criterion	Description
1	Priority	Severity grouping: Critical, High, Medium, Low, Very Low, Minor.
2	OS Score	100 for "all"; else distinct OS count (higher = more platforms).
3	Word Count	Description length (higher = more complete).

(1) Bug Severity Priority: Issues are first grouped into six severity levels, with higher severity groups taking absolute precedence. **(2) Operating System Impact:** Within each severity group, issues are then sorted by the number of affected operating systems. Issues affecting all platforms receive the highest OS score of 100, while others receive a score equal to the number of distinct OS labels (e.g., 2 for an issue affecting Windows and macOS). Issues with higher OS scores are ranked first. **(3) Description Completeness:** Finally, among issues with identical severity and OS scores, we rank by word count, using description length as a proxy for completeness under the assumption that longer issues contain more detailed reproduction steps and context.

This hierarchical ranking ensures that a critical bug affecting one platform is prioritized over a low-severity bug affecting all platforms, and that cross-platform issues within the same severity level are ranked above single-platform issues. For each project, we select the top 10 highest-ranked issues following this ordering, for subsequent LLM-based ranking evaluation, in a way that these issues represents a list of critical issues with detailed descriptions.

LLM-based Ranking: The third phase employs an LLM-based ranking, in which GPT-5-mini evaluates each project's top 10 issues selected by the rule-based selection, and then assigns a priority ranking from 1 (most important) to 10 (least important) for each issue according to *impact*, *scope* and *urgency*, as illustrated in the prompt template in Figure 4.3.

IssueSpecter produces two outputs: the rule-based ranking derived from Phase 3, and the LLM ranking for the same set of top-selected issues per project, where the ranking process is performed independently for each project. Issues whose proposed code breaks the project's original test suite are ranked lower, with the ranking decreasing further as the number of failing tests increases for a given fix.

Role: You are a software engineering expert tasked with ranking up to 10 issue reports from a specified project according to urgency, scope and bug impact.

Task: Each issue includes associated attributes: *bug severity, affected OS, number of unit tests failing* from the project test suite when the proposed Python fix is applied, and *word count*.
Evaluate each issue based on logical validity, alignment with best practices and documentation, and support from external technical resources, when available, and assess the issue's fit with the project's architecture and design patterns. Additionally, evaluate the impact of failed tests, noting if a proposed fix passes all tests. Rank issues in descending order of impact and urgency, with "1" as the highest. If required data is missing or unverifiable, clearly indicate this in the output.

JSON Output: For each ranked issue, generate a structured Markdown report under the *validity report* field. The final output should be a valid JSON object with the ranking of each issue, with each object containing *issue id, reasoning, validity classification* (a boolean to indicate if the issue is valid or not), *confidence rating*, and the *validity report*.

Input JSON: The JSON object input to process:
`{JSON_WITH_ISSUES_ATTRIBUTES_TO_BE_RANKED}`

Figure 4.3: Simplified version of the prompt template used during the LLM-based Ranking.

Chapter 5

Dataset

Our evaluation consists of a total of 13 open-source projects, as shown in Table 5.1.

Table 5.1: Characteristics of the 13 selected Python projects from CodaMosa dataset. Metrics include number of files, mean and maximum lines of code per file, test coverage percentage, commit hash at analysis time, and number of existing tests. The dataset exhibits substantial diversity in size, coverage, and testing maturity.

Project	#Files	Mean	Max	Cov.	Hash	#Tests
Ansible	1,573	94.5	2,235	46.2%	22721	2,062
Cookiecutter	89	70.5	590	88.6%	af1d7	382
Dataclasses-JSON	39	94.9	358	94.8%	dc639	324
Docstring Parser	1,001	277.5	8,510	94.7%	43407	230
HTTPie	2,123	211.8	8,664	32.2%	5b604	1,028
isort	5,049	215.1	20,748	16.0%	77bc6	536
Mimesis	90	577.0	13,068	99.1%	70b11	8,268
PySnooper	1,013	147.7	5,158	40.1%	0561a	78
Sanic	2,860	214.4	8,664	96.2%	a64d7	1,693
Thonny	328	184.4	3,432	37.3%	55ac2	3
Tornado	111	258.2	2,560	30.7%	d30ef	1,317
tqdm	67	85.5	1,427	71.0%	0ed5d	150
Youtube-DL	902	152.3	5,158	62.5%	1e109	2,767
Total	15,245	—	—	—	—	18,838
Mean	1,173	198	6,120	62.3%	—	1,449

Specifically, from the 27 projects in the CodaMosa dataset (Lemieux et al., 2023), we selected projects that are actively maintained (e.g, with commits after 2023), excluding 11 inactive projects. Active maintenance is a critical selection criterion for our study, as we generate issues for the latest version of the project, validate generated bug reports by submitting them as issues and pull requests to the respective repositories, enabling us to obtain future community feedback. In addition, three projects with recent activity were excluded due to technical incompatibilities with SlipCover during coverage analysis (i.e., dependency conflicts that prevented the extraction of uncovered code segments). Overall, the 13 selected projects cover diverse domains that encompass developer tools, web frameworks, data utilities, automation systems, and multimedia applications, ensuring that IssueSpecter captures bugs across different programming paradigms.

The projects exhibit diversity in terms of complexity and test coverage, with project sizes varying considerably, ranging from compact codebases like TQDM with 67 files to bigger ones, such as iSort with 5,049 files, with this variation also reflecting in LOC metrics. Thus, this indicates significant differences in architectural patterns and code organization across projects with this diversity in project characteristics increasing the likelihood of encountering varied defect types. We focus on widely used Python projects that have been used for prior evaluation (Alshahwan et al., 2024; Altmayer Pizzorno & Berger, 2025; Bhatia, Gandhi, Kumar, & Jalote, 2024; Dakhel et al., 2024; El Haji, Brandt, & Zaidman, 2024; Jiri, Emese, & Medlen, 2024; Krodinger, Lukasczyk, & Fraser, 2025; Lemieux et al., 2023; Long et al., 2025; Lukasczyk & Fraser, 2022; Lukasczyk, Kroiß, & Fraser, 2023; Schäfer, Nadi, Eghbali, & Tip, 2024b, 2024c; Yuan et al., 2024). To prevent data leakage, we restrict each project to the specific commit hash recorded in Table 5.1, which corresponds to the latest version of the project at the time of the experiment. This ensures that our analysis reflects the state of the codebase at evaluation time. This mitigates the risk that GPT-5-mini was exposed to subsequent commits, bug fixes, or issue reports filed after that snapshot during its training, which could otherwise inflate the apparent bug detection rate.

We restrict our evaluation to Python projects to control for language specific factors. By fixing the programming language while varying project domains, we isolate the effect

of domain diversity on bug detection without introducing language-related features that would complicate interpretation of results.

Test coverage varies widely across our dataset, from highly tested projects like Mimesis (99.09%) and Dataclasses-JSON (94.75%) to those with substantial uncovered code such as iSort (16.02%) and Tornado (30.65%), with the number of existing tests also varies substantially, from minimal test suites like Thonny, to comprehensive ones like Mimesis, reflecting different maturity levels across projects.

The uncovered segments identified by SlipCover serve as the direct input to IssueSpecter's issue generation phase. Table 5.2 reports the uncovered segments that IssueSpecter selected as targets for issue generation across the 13 projects. The number of uncovered segments per project varies substantially, from 24 in iSort to 1,832 in Ansible, reflecting the combined influence of codebase size and test coverage level. Across all projects, uncovered segments span an average of 15 to 25 lines, suggesting that the uncovered regions tend to be compact, self-contained blocks of untested logic rather than large untested modules. This compactness is well-suited to LLM-based analysis, as focused, bounded segments provide sufficient context for the model to reason about potential defects, while remaining small enough to be processed within a single prompt.

Table 5.2: For each project, we report the total number of uncovered segments identified by SlipCover, along with the maximum, minimum, and average number of source lines spanned by those segments.

Project	**# Uncovered Segments**	**Lines Maximum**	**Lines Minimum**	**Lines Average**
Ansible	1832	364	1	20.77
Cookiecutter	25	101	2	24.32
Dataclasses-JSON	55	84	1	17.40
Docstring Parser	51	151	1	23.78
HTTPie	206	111	1	15.09
iSort	24	30	4	10.83
Mimesis	300	63	1	11.27
PySnooper	35	176	1	18.14
Sanic	322	188	1	21.98
Thonny	28	135	2	25.46
Tornado	284	193	1	18.35
tqdm	44	59	1	16.09
Youtube-DL	283	901	1	26.00

Chapter 6

Evaluation

Since IssueSpecter generates a large number of issue reports automatically, a manual annotation was conducted prior to evaluation to establish the ground truth for assessing bug validity and ranking effectiveness. Two annotators independently reviewed the top 10 issues selected by the rule-based ranking stage per project, derived from the output of Phase 3 of IssueSpecter's pipeline, as described in Chapter 4, totaling 130 issues across 13 projects. These 130 issues, already filtered and ordered by the rule-based criteria of severity, platform scope, and description completeness, serve as the shared input to both the LLM-based ranker and the human annotators, ensuring that the ranking comparison is conducted over the same candidate set. Each issue was classified as a Valid Bug, Requires Further Investigation, or Invalid, following the scheme described in Table 6.3.

Annotators also produced a ground-truth priority ranking for each project's 10 issues from 1 (highest priority) to 10 (lowest priority). These annotated rankings serve as the ground truth against which both rule-based and LLM-based ranking strategies are evaluated throughout this chapter.

6.1 Ranking Evaluation Metrics

To evaluate the effectiveness of both rule-based and LLM-based ranking approaches, we employ five complementary metrics that assess ranking quality from different perspectives.

All metrics compare the predicted ranking against the golden standard ranking produced by human annotators, with higher values indicating better performance.

Mean Reciprocal Rank (MRR): MRR measures how quickly users can find the most critical issue by computing the reciprocal of the rank position where the highest-priority issue appears. Formally, MRR is defined as:

$$\text{MRR} = \frac{1}{\text{rank}_1} \tag{1}$$

where rank_1 is the position of the highest-priority issue in the ranked list. For example, if the top-priority issue is ranked at position 3, the reciprocal rank is $1/3 \approx 0.33$. MRR is particularly relevant for our use case because developers typically begin triaging with the first issue they see; thus, placing the most critical bug at position 1 yields MRR = 1.0 (perfect), while ranking it at position 10 yields MRR = 0.1. This metric directly reflects the practical scenario where developers have limited time and need the highest-impact issue surfaced immediately.

Precision at K (P@K): Precision@K measures the proportion of relevant issues among the top-K ranked items:

$$\text{P@K} = \frac{|\{\text{relevant items}\} \cap \{\text{top-K items}\}|}{K} \tag{2}$$

An issue is considered "relevant" at position K if it should legitimately appear in the top-K positions according to the golden standard ranking. For instance, P@3 = 0.67 indicates that 2 out of the 3 top-ranked issues should indeed be in the top 3. We report P@1, P@3, P@5, and P@7 to capture ranking quality at different cutoff points, reflecting scenarios where developers examine varying numbers of issues. P@1 is critical for identifying whether the single most important issue is correctly ranked first, while P@5 and P@7 assess whether the ranking maintains quality as developers triage more issues.

Mean Average Precision (MAP@K): MAP@K computes the average precision across all positions up to K, providing a more comprehensive view of ranking quality than Precision@K alone. Formally:

$$\text{MAP@K} = \frac{1}{K} \sum_{i=1}^{K} \text{P@}i \tag{3}$$

where P@i is the precision at position i. MAP considers not just whether relevant issues appear in the top-K, but also at which positions they appear, rewarding rankings that place multiple high-priority issues early in the list. For each position from 1 to K, we calculate the precision and average these values. MAP@K thus captures both the recall of high-priority issues within the top-K and the ordering quality, making it well-suited for evaluating whether our ranking approaches consistently surface important bugs throughout the top portion of the ranked list rather than concentrating them at a single position.

Expected Reciprocal Rank (ERR): ERR models a behavior where users examine issues sequentially from top to bottom and stop once they find a satisfactory issue to investigate. Formally:

$$\text{ERR} = \sum_{r=1}^{n} \frac{1}{r} \cdot R_r \cdot \prod_{i=1}^{r-1} (1 - R_i) \tag{4}$$

where R_r is the probability of satisfaction at rank r, computed as:

$$R_r = \frac{2^{\text{relevance}_r} - 1}{2^{\text{max\_relevance}}} \tag{5}$$

and relevance_r is the relevance score of the item at rank r (derived from golden ranking: relevance = max_grade + 1 − golden_rank, where max_grade = 10 in our evaluation). Unlike MRR, which assumes only the top-ranked issue matters, ERR accounts for the graded relevance of all issues in the ranking. At each position r, ERR computes the probability that a user will be satisfied based on the issue's relevance grade, and the probability that the user is still searching (i.e., has not yet found a sufficiently relevant issue). This metric is usually employed in Recommender Systems, and is appropriate for our context because developers may investigate several issues before finding one worth immediate action, and higher-ranked issues should have higher relevance to maximize the probability of early satisfaction. ERR values closer to 1.0 indicate that high-relevance issues appear early in the ranking.

Normalized Discounted Cumulative Gain at K (NDCG@K): NDCG@K evaluates ranking quality by considering both the relevance of issues and their positions, with a logarithmic discount applied to lower positions. Formally:

$$\text{DCG@K} = \sum_{i=1}^{K} \frac{\text{rel}_i}{\log_2(i+1)} \tag{6}$$

$$\text{NDCG@K} = \frac{\text{DCG@K}}{\text{IDCG@K}} \tag{7}$$

where rel_i is the relevance score of the item at position i (derived from golden ranking: $\text{rel} = n + 1 - \text{golden\_rank}$, where $n = 10$ is the number of items), and IDCG@K is the Ideal DCG computed by sorting items by their relevance scores in descending order. Relevance scores are derived from golden standard rankings (e.g., golden rank 1 receives relevance 10, rank 2 receives relevance 9, and so on). The Discounted Cumulative Gain (DCG) sums these relevance scores, discounted by position using $\log_2(i+1)$, penalizing relevant issues that appear lower in the ranking. NDCG normalizes DCG by dividing by the Ideal DCG (IDCG), which represents the best possible ranking. NDCG ranges from 0 to 1, with 1.0 indicating a perfect ranking. NDCG is particularly valuable because it accounts for graded relevance rather than treating all issues as equally important, which aligns with the reality that some bugs are far more critical than others.

Together, these metrics provide a comprehensive evaluation framework: MRR focuses on finding the single most critical issue, Precision@K and MAP@K assess the overall quality of the top-K subset, ERR models developer behavior with graded relevance, and NDCG@K captures position-weighted ranking quality across multiple cutoffs. By reporting all five metrics, we can characterize ranking performance from multiple practical perspectives relevant to bug triage workflows.

6.2 RQ1: LLM-based issue generation effectiveness

6.2.1 Analysis of the Top 10 Issues Across 13 Projects

Tables 6.1 and 6.2 present the breakdown of the 10,467 issues generated by IssueSpecter across the 13 selected projects, covering six severity levels (Critical, High, Medium, Low, Minor, and Very Low), a count of unclassified issues where no severity was assessed by the LLM, and an independent flag for inconsistent documentation.

Table 6.1: Overview of LLM-generated issues per project. For each project we report the total number of generated issues, the number of issues in which an actual bug was found and the percentage of issues addressing bugs.

Project	**Total Issues**	**Issues with Bugs**	**% with Bug**
Ansible	5496	5212	94.83%
Cookiecutter	75	71	94.67%
Dataclasses-JSON	165	161	97.58%
Docstring-Parser	153	144	94.12%
HTTPie	618	592	95.79%
iSort	72	69	95.83%
Mimesis	900	855	95.00%
PySnooper	105	98	93.33%
Sanic	966	929	96.17%
Thonny	84	82	97.62%
Tornado	852	803	94.25%
tqdm	132	124	93.94%
Youtube-DL	849	789	92.93%
Total	**10467**	**9929**	**94.86%**

Table 6.2: Severity distribution and issues with inconsistent documentation. This table presents the breakdown of issues across severity levels (Critical (C), High (H), Medium (Me), Low (L), Very Low (VL), Minor (Mi)), the number of unclassified issues (Unc.), and the number of issues flagged as containing inconsistent documentation (Incon. Doc.).

Project	**Severity**						**Unc.**	**Incon. Doc.**
	C	**H**	**Me**	**L**	**VL**	**Mi**		
Ansible	4	648	1824	1301	175	1260	284	392
Cookiecutter	0	6	34	20	2	9	4	9
Dataclasses-JSON	1	20	44	29	4	63	4	12
Docstring-Parser	0	8	45	33	14	44	9	28
HTTPie	0	63	211	155	34	129	26	39
iSort	0	1	21	23	8	16	3	10
Mimesis	1	58	270	258	96	172	45	236
PySnooper	0	8	37	27	2	24	7	4
Sanic	5	108	270	230	49	267	37	139
Thonny	0	6	25	21	8	22	2	6
Tornado	2	114	241	207	62	177	49	116
tqdm	0	17	35	32	8	32	8	19
Youtube-DL	0	105	299	225	33	127	60	62
Total	**13**	**1162**	**3356**	**2561**	**495**	**2342**	**538**	**1072**

The Issue Generation phase achieved a high error detection rate, with 9,929 out of 10,467 issues (94.9%) flagged as containing at least one actual defect in the analyzed uncovered code segment. This rate should be interpreted with caution rather than taken as a direct measure of bug detection accuracy, since LLMs tend to exhibit a confirmation bias when prompted to find bugs, gravitating toward reporting a defect even in correct code. Our manual annotation of the top-130 ranked issues mitigates this concern in practice, and reveals a more nuanced picture, as summarised in Table 6.3, reaching an agreement of 80.3%.

Table 6.3: Manual assessment of 130 top-ranked generated issues by two human annotators. Valid bugs exhibit high confidence of existence; issues requiring further investigation appear plausible but require execution for confirmation or represent edge cases; invalid issues are hallucinations or misunderstandings of project usage.

Bug Assessment	**Count**	**Percentage**
Valid Bug	49	37.7%
Requires Further Investigation	61	46.9%
Invalid	20	15.4%
Total	**130**	**100%**

As shown in Table 6.3, 84.6% of top-ranked issues are either confirmed valid bugs (37.7%) or require further investigation (46.9%), with only 15.4% classified as outright invalid, corresponding to hallucinations or misunderstandings of the project's intended usage patterns. Practitioners should be aware that exposing developers to a non-trivial rate of invalid reports risks eroding trust in LLM-assisted tools over time, reinforcing the importance of the ranking and filtering stages.

Regarding defect types, Table 6.11 shows that the validated issues skew heavily toward *Logic and Conditional Bugs*, with 43 cases accounting for 33.1% of all annotated issues, and *Input Validation, Boundary, and Sentinel Handling Errors*, with 39 cases accounting for 30.0%. Together, these categories make up over 63% of the total annotated issues. This concentration suggests that LLMs are particularly sensitive to control-flow and boundary-handling problems residing in uncovered code, precisely the category of defects that test suites most commonly fail to exercise. Security-related defects, categorized as *Security and Sensitive Data Leakage Due to Logic Oversight*, were less frequent but nonetheless present, with four identified cases. Our case studies confirm that at least one of these represents a medium-to-high severity path-traversal vulnerability (CWE-22) in a widely used HTTP client library described in Section 7.

Finding 1: IssueSpecter generates top-ranked issues of which 84.6% are either confirmed valid bugs (37.7%) or require further investigation (46.9%), with only 15.4% classified as outright false positives. Our defect distribution suggests that LLM-based analysis of uncovered code segments can be effective at identifying control-flow and boundary-handling defects.

Implication 1: IssueSpecter issue generation from uncovered code segments could be a practical approach to complement existing test suites by systematically ranking potential bugs that may warrant further investigation.

Severity distribution is shown in Table 6.1 (for project error details) and Table 6.2 (for severity breakdown and unclassified issues). The severity distribution is dominated by Medium issues, with 3,356 cases accounting for 32.1% of the total, followed by Low severity issues, with 2,561 cases representing 24.5%. In contrast, Critical issues are rare, with only 13 instances comprising less than 0.1% of all reported issues, reflecting the expected distribution of defects in actively maintained open-source projects. The scarcity of Critical and High severity issues means that the ranking stage can more selectively surface the most impactful candidates, and developers are presented with a small, focused set of high-priority issues to triage rather than an undifferentiated flood of reports.

Table 6.4 reports the distribution of the number of bugs generated per uncovered segment. Across all 13 projects, IssueSpecter identifies 3 bugs for the vast majority of segments (84.77%), with only 15.10% of segments yielding 2 bugs and 0.23% yielding a single bug. For a segment where the LLM identifies 3 bugs, all 3 generated issues are flagged as containing an error in the code; for a segment yielding 2 bugs, 2 issues are flagged as containing an error and 1 is flagged as having no bugs; and for a segment yielding only 1 bug, 2 of the 3 generated issues are flagged as having no error. The proportion of segments yielding 3 bugs is consistent with the known tendency of LLMs to exhibit confirmation bias when explicitly prompted to identify bugs.

Table 6.4: Distribution of the number of bugs generated per uncovered segment across all 13 projects. For each project, we report the total number of uncovered segments and the proportion of segments yielding 1, 2, or 3 bugs. The vast majority of segments yield 3 bugs (80% to 93%), with segments yielding only 1 bug being rare across all projects (≤4.55%).

		Count			**Percentage (%)**		
Project	**#Segments**	**1 Bug**	**2 Bugs**	**3 Bugs**	**1 Bug**	**2 Bugs**	**3 Bugs**
Ansible	1832	1	282	1549	0.05	15.39	84.55
Cookiecutter	25	0	4	21	0.00	16.00	84.00
Dataclasses-JSON	55	0	4	51	0.00	7.27	92.73
Docstring-Parser	51	0	9	42	0.00	17.65	82.35
HTTPie	206	1	24	181	0.49	11.65	87.86
iSort	24	0	3	21	0.00	12.50	87.50
Mimesis	300	1	43	256	0.33	14.33	85.33
PySnooper	35	0	7	28	0.00	20.00	80.00
Sanic	322	1	35	286	0.31	10.87	88.82
Thonny	28	0	2	26	0.00	7.14	92.86
Tornado	284	1	47	236	0.35	16.55	83.10
tqdm	44	2	4	38	4.55	9.09	86.36
Youtube-DL	283	0	60	223	0.00	21.20	78.80
Total	**3489**	**8**	**527**	**2958**	**0.23**	**15.10**	**84.77**

Beyond severity, IssueSpecter also assigns each generated issue a set of affected operating systems, as reported in Table 6.5. The distribution reveals that the vast majority of issues are flagged as cross-platform, affecting Linux, Windows, and macOS in equal proportions across most projects. We expect this, since the 13 projects under study are platform-agnostic Python libraries, whose source code rarely contains OS-specific logic that would justify attributing a defect to a single platform. We note that Ansible is an exception, showing a slightly higher proportion of Linux-targeted issues, consistent with its Linux-centric operational identity as an IT automation platform. Taken together, these results suggest that the LLM demonstrates reasonable platform awareness when assigning

OS targets rather than arbitrarily selecting a single OS.

Table 6.5: OS distribution of generated issues per project. Percentages are computed per project over Linux, Windows, macOS, and Unspecified counts.

Project	**Linux**	**Windows**	**macOS**	**Unspecified**
Ansible	4715 (39%)	4247 (35%)	2568 (21%)	549 (5%)
Cookiecutter	71 (33%)	71 (33%)	71 (33%)	4 (2%)
Dataclasses-JSON	158 (35%)	158 (35%)	136 (30%)	4 (1%)
Docstring-Parser	140 (34%)	141 (34%)	124 (30%)	9 (2%)
HTTPie	556 (33%)	564 (34%)	536 (32%)	30 (2%)
iSort	64 (33%)	66 (34%)	59 (31%)	3 (2%)
Mimesis	836 (34%)	837 (34%)	768 (31%)	51 (2%)
PySnooper	98 (33%)	98 (33%)	98 (33%)	7 (2%)
Sanic	909 (34%)	912 (34%)	847 (31%)	42 (2%)
Thonny	73 (33%)	73 (33%)	73 (33%)	2 (1%)
Tornado	758 (33%)	755 (33%)	735 (32%)	52 (2%)
tqdm	124 (33%)	124 (33%)	121 (32%)	8 (2%)
Youtube-DL	706 (34%)	707 (34%)	624 (30%)	61 (3%)
Total	**9208**	**8753**	**6760**	**822**

Table 6.6 reports the average word count of issue descriptions generated by IssueSpecter, broken down by severity level. Critical and High severity issues consistently receive longer descriptions across most projects (e.g., Ansible Critical: 409.5 words, High: 385.1 words), while Very Low and Minor issues receive the shortest, with 292.5 and 337.2 words respectively. This suggests that the LLM allocates more contextual detail to issues it judges as more impactful, which is a desirable property since higher-severity reports benefit most from detailed reproduction steps and fix suggestions. There are three exceptions, Mimesis, PySnooper, and iSort, where this trend does not hold at certain severity levels; however, in each case the affected category contains only one or two representative issues, making these anomalies attributable to a small sample size rather than any systematic behavior of IssueSpecter.

Table 6.6: Average word count of generated issue descriptions per project, broken down by severity level, ordered from most to least severe: Critical, High, Medium, Low, Very Low, and Minor. A dash indicates no issues of that severity were generated for the given project. Minor denotes issues of lower severity than Very Low.

Project	**Critical**	**High**	**Medium**	**Low**	**Very Low**	**Minor**
Ansible	409.5	385.1	366.8	331.1	292.5	337.2
Cookiecutter	—	422.7	345.2	332.8	241.5	342.6
Dataclasses-JSON	373.0	376.6	374.5	333.1	271.8	329.8
Docstring-Parser	—	348.6	349.9	314.9	273.9	308.2
HTTPie	—	350.3	356.9	316.2	291.1	315.0
iSort	—	338.0	399.7	334.4	290.4	353.2
Mimesis	226.0	361.0	342.5	308.5	271.1	296.0
PySnooper	—	395.6	353.9	319.3	347.0	299.5
Sanic	321.0	385.2	360.1	326.6	290.8	339.2
Thonny	—	368.5	398.2	306.6	287.0	314.8
Tornado	401.5	384.3	366.7	323.6	304.0	333.6
tqdm	—	405.6	351.0	338.9	324.4	321.3
Youtube-DL	—	365.4	339.4	314.3	277.5	317.1

6.2.2 Comparison with CoverUp

We compare IssueSpecter against CoverUp (Altmayer Pizzorno & Berger, 2025), the state-of-the-art coverage-driven test generation tool, in terms of bug detection effectiveness. We run CoverUp using the same model as IssueSpecter (GPT-5-mini) for a fair comparison, and we also match the number of evaluated artifacts per project, ensuring that the number of evaluated artifacts are the same for each tool. That is, for each project, CoverUp generates k unit tests over a set of uncovered segments, while we apply the IssueSpecter pipeline to the same segments and select the top-k ranked issues, so that both approaches produce exactly k outputs per project. This process resulted in 168 artifacts per tool across

all evaluated projects (168 unit tests generated in total and 168 issue reports generated in total).

We manually inspected all outputs from both tools, classifying each as Valid Bug, Requires Further Investigation, or Invalid, following the same annotation procedure described in the first paragraph of Chapter 6. For CoverUp, we execute the generated unit tests alongside the project's existing test suite to identify regressions. For IssueSpecter, we apply each issue's proposed fix to the original program and execute the test suite to check for regressions introduced by the patch.

Table 6.7 summarizes the annotation results, with IssueSpecter achieving a higher valid bug rate (41.7% vs. 33.3%) and a lower invalid rate (19.0% vs. 23.8%).

Table 6.7: Comparison of bug assessment outcomes between CoverUp and IssueSpecter.

Bug Assessment	**CoverUp**		**IssueSpecter**	
	Count	**(%)**	**Count**	**(%)**
Valid Bug	56	33.3%	70	41.7%
Requires Further Investigation	72	42.9%	66	39.3%
Invalid	40	23.8%	32	19.0%
Total	168	100%	168	100%

Beyond the quantitative difference, IssueSpecter additionally provides structured reports with reproduction steps and candidate fixes, which CoverUp does not produce. Interpreting the intent and validity of a generated unit test requires the developer to understand both the test and the surrounding codebase, whereas IssueSpecter's reports are more immediately understood.

CoverUp may identify a defective segment, but the generated tests often assert correct behavior without exercising the actual fault, allowing buggy code to pass while increasing coverage. We note that many tests rely on excessive mocking, replacing surrounding logic with stubs such that assertions validate the mock behavior rather than the real system. In addition, a non-trivial portion of tests target trivial code (e.g., simple guards or defaults) where no meaningful defects can be exposed, making them ineffective as fault detectors.

Taken together, these facts suggest that CoverUp optimizes for coverage rather than fault detection: tests may satisfy coverage criteria while remaining insensitive to underlying bugs, reflecting the oracle problem in automated test generation.

Finding 2: Comparison with CoverUp. IssueSpecter achieves a higher valid bug rate than CoverUp (41.7% vs. 33.3%) and a lower false positive rate (19.0% vs. 23.8%) under identical evaluation conditions, using the same model and the same number of evaluated artifacts per project.

Implication 2: Generating issue reports from uncovered code segments is an effective strategy for bug detection. Beyond the quantitative difference, IssueSpecter generates structured, human-readable issue reports with reproduction steps and proposed fixes, whereas CoverUp may produce unit tests that often assert correct behavior without exercising the actual fault, target trivial code paths, or rely on excessive mocking, reflecting the oracle problem in automated test generation.

6.3 RQ2: LLM-based ranking effectiveness

Annotation Setup. To evaluate the effectiveness of both rule-based and LLM-based ranking, two annotators independently reviewed the top 10 issues selected per project, totalling 130 issues across 13 projects. Each issue was classified according to the bug taxonomy from BugPilot (Sonwane et al., 2025), with the distribution shown in Table 6.11, and assessed for bug validity according to the categories presented in Table 6.3. Annotators also produced a ground-truth ranking of each project's 10 issues from 1 (highest priority) to 10 (lowest priority), using the same three criteria employed by the LLM-based ranker: *impact* (severity of effect on functionality or users, prioritizing issues that break core features), *scope* (breadth of affected users, features, or components), and *urgency* (prioritizing security vulnerabilities and regressions that risk system or user safety, while de-prioritizing rare edge cases). These annotated rankings serve as ground truth for evaluating both ranking strategies.

Tables 6.8, 6.9, and 6.10 present an evaluation of rule-based ranking versus LLM-based ranking across all 13 projects, using metrics such as Precision(P@k), Normalized Discounted Cumulative Gain (NDCG@k), Mean Reciprocal Rank (MRR), Expected Reciprocal Rank (ERR), and Mean Average Precision (MAP), as described in Chapter 6.

Table 6.8: Precision at k comparison between the rule based approach and LLM-based ranking across 13 projects. Values represent the proportion of valid or potential bugs among the top-k ranked issues. Bold indicates superior performance.

Project	**Rule-Based Ranking**			**LLM-based Ranking**		
	P@1	**P@3**	**P@5**	**P@1**	**P@3**	**P@5**
ansible	1.00	0.67	0.60	1.00	**1.00**	**0.80**
cookiecutter	0.00	0.33	0.60	0.00	**0.67**	**0.80**
dataclasses_json	1.00	0.33	0.60	1.00	**1.00**	**0.80**
docstring_parser	0.00	**0.67**	0.60	0.00	0.33	0.60
httpie	0.00	0.33	0.60	**1.00**	**0.67**	**1.00**
isort	0.00	0.00	0.60	0.00	**0.33**	0.60
mimesis	0.00	0.33	0.40	0.00	0.33	0.40
pysnooper	0.00	0.67	0.40	0.00	0.67	**0.60**
sanic	0.00	0.67	0.80	0.00	**1.00**	0.80
thonny	0.00	0.33	0.40	0.00	0.33	0.40
tqdm	0.00	0.33	0.40	0.00	0.33	0.40
tornado	0.00	0.33	0.60	0.00	**0.67**	**0.80**
youtube_dl	0.00	0.33	0.60	**1.00**	**0.67**	0.60
Total	2.00	5.32	7.20	**4.00**	**8.00**	**8.60**

Precision. LLM ranking achieves higher precision at all cutoff points. At P@1, LLM ranking correctly places the highest-priority issue in 4 projects, compared to only 2 for rule-based ranking, showing a 100% improvement at the position most relevant for developers who can inspect only one issue. This advantage increases at P@3, where LLM ranking achieves a cumulative score of 8.00, while rule-based ranking scores 5.32, around 50% of improvement. At P@5, LLM ranking maintains a consistent 19% improvement (8.60 vs. 7.20). Notable project highlights include HTTPie, which improves at P@1 from 0 to a perfect score under LLM ranking and achieves perfect precision at P@5, and Youtube-DL, which reaches a perfect P@1 under LLM ranking compared to 0 with rule-based ranking.

NDCG. LLM ranking shows higher cumulative NDCG at every cutoff point, as indicated in Table 6.9. NDCG rewards relevance while increasingly penalizing relevant items placed

lower in the ranking. At NDCG@1, LLM ranking scores 9.90, compared to 9.40 for rule-based ranking. This gap widens progressively, reaching 11.98 for LLM versus 11.61 for rule-based at NDCG@10. Individual project improvements are notable in Cookiecutter, increasing by a factor of 6 at NDCG@1, and in HTTPie, by a factor of 2.5. Tornado improves from 0.70 to 0.80. The consistent improvements at NDCG@10 across most projects suggest that LLM ranking performs better across the entire top-10 list, not just at the top positions.

Table 6.9: Comparison of ranking performance using NDCG@k.

Project	NDCG@K									
	Rule-based					LLM				
	@1	@3	@5	@7	@10	@1	@3	@5	@7	@10
ansible	1.00	0.94	0.85	0.88	0.95	1.00	**1.00**	**0.97**	**0.92**	**0.98**
cookiecutter	0.10	0.53	0.65	0.70	0.79	**0.60**	**0.85**	**0.87**	**0.92**	**0.93**
dataclasses_json	1.00	0.72	0.73	0.81	0.90	1.00	**1.00**	**0.98**	**0.96**	**0.99**
docstring_parser	0.90	**0.82**	0.81	**0.92**	0.93	0.90	0.80	0.81	0.89	0.93
httpie	0.40	0.65	0.73	0.77	0.85	**1.00**	**0.94**	**0.98**	**0.96**	**0.98**
isort	0.60	0.51	0.66	0.68	0.82	**0.80**	0.51	**0.67**	**0.72**	**0.85**
mimesis	0.70	0.77	0.72	0.76	0.89	0.70	0.77	0.72	0.76	0.89
pysnooper	0.90	0.72	0.63	0.75	0.87	0.90	**0.83**	**0.85**	**0.84**	**0.94**
sanic	**0.50**	**0.80**	**0.88**	**0.90**	**0.91**	0.20	0.65	0.76	0.76	0.83
thonny	**0.70**	**0.77**	**0.69**	**0.83**	**0.89**	0.10	0.50	0.57	0.70	0.78
tqdm	0.90	0.81	0.74	0.79	0.91	0.90	0.81	0.74	0.79	0.91
tornado	0.70	0.71	0.67	0.74	0.85	**0.80**	**0.89**	**0.85**	**0.86**	**0.93**
youtube_dl	0.70	0.78	0.79	0.74	0.90	**1.00**	**0.94**	**0.89**	**0.97**	**0.97**
Total	9.40	9.53	9.82	10.53	11.61	**9.90**	**10.49**	**10.66**	**11.32**	**11.98**

MRR, ERR, and MAP. As shown in Table 6.10, LLM ranking achieves a cumulative MRR of 7.21, compared to 5.11 for rule-based ranking, showing a 41% improvement. HTTPie improves from MRR = 0.14 to 1.00, and Youtube-DL from 0.25 to 1.00. For ERR, LLM ranking scores 8.83, while rule-based scores 7.22, showing a 22% improvement, with

significant gains in HTTPie, Youtube-DL, and PySnooper. Finally, MAP improves from 8.05 to 9.33, showing a 16% improvement, with HTTPie, Ansible, Youtube-DL, and iSort showing the largest individual gains.

Table 6.10: MRR, ERR, and MAP metrics comparison

Project	**Rule-Based Ranking**			**LLM Ranking**		
	MRR	**ERR**	**MAP**	**MRR**	**ERR**	**MAP**
ansible	1.00	1.00	0.78	1.00	1.00	**0.89**
cookiecutter	0.10	0.41	0.57	**0.20**	**0.64**	**0.76**
dataclasses_json	1.00	1.00	0.71	1.00	1.00	**0.93**
docstring_parser	0.50	0.51	**0.70**	0.50	0.51	0.66
httpie	0.14	0.58	0.63	**1.00**	**1.00**	**0.86**
isort	0.20	0.24	0.49	**0.33**	**0.36**	**0.56**
mimesis	0.25	0.38	0.58	0.25	0.38	0.58
pysnooper	0.50	0.51	0.58	0.50	**0.75**	**0.72**
sanic	0.17	**0.63**	**0.76**	**0.50**	0.62	0.72
thonny	**0.25**	0.39	**0.60**	0.10	**0.57**	0.56
tqdm	0.50	0.50	0.56	0.50	0.50	0.56
tornado	0.25	0.44	0.53	**0.33**	**0.50**	**0.69**
youtube_dl	0.25	0.63	0.56	**1.00**	**1.00**	**0.82**
Total	5.11	7.22	8.05	**7.21**	**8.83**	**9.33**

Despite these improvements, accurately retrieving the single most relevant issue remains challenging for both approaches. However, LLM ranking successfully reaches an average of 66% of projects at P@5, offering meaningful practical value over heuristics alone.

Finding 3: LLM-based ranking consistently outperforms rule-based ranking across all evaluated metrics, with the advantage being most significant at top positions, which are most critical for developers who can only triage a small number of issues.

Implication 3: Rule-based heuristics alone are insufficient for reliably identifying the highest-priority bugs. Incorporating LLM-based re-ranking as a second stage provides a substantial improvement, particularly for identifying the most impactful issue per project, a valued concern in resource-constrained maintenance workflows.

6.4 RQ3: Bug types diversity and project domain

Table 6.11 presents the distribution of bug categories across the 130 manually annotated issues, and Section 7 provides selected case studies spanning projects from different domains. The results reveal that our approach surfaces a diverse range of defect types across all nine categories from the taxonomy proposed in BugPilot (Sonwane et al., 2025), with no single category accounting for more than one third of all validated issues, with this diversity reflecting the breadth of Python defect patterns that LLM-based analysis of uncovered segments is capable of exposing.

Table 6.11: Distribution of bug categories across validated issues generated by GPT-5-mini, manually annotated according to the taxonomy of BugPilot (Sonwane et al., 2025), obtaining a Cohen Kappa agreement of 82%. Logic errors and input validation issues dominate, while I/O handling and mutability bugs are less frequently detected (N=130 validated issues).

Bug Category	**Count**
Logic and Conditional Bug	43
Input Validation, Boundary, and Sentinel Handling Error	39
State Consistency, Bookkeeping, and Caching Bug	14
Protocol and Specification Conformance Bug	12
API, Signature Mismatch, or Backward Compatibility Break	10
Incorrect Argument Forwarding, Constructor, or Inheritance Contract Break	6
Security and Sensitive Data Leakage Due to Logic Oversight	4
IO, Filesystem, or Resource Handling Bug	1
Copy Semantics, Mutability Aliasing, or In Place Mutation of Inputs	1
Total	**130**

Domain-agnostic patterns. Logic and conditional bugs, accounting for 43 cases and 33.1%, and input validation errors, with 39 cases and 30.0%, dominate the distribution regardless of project domain. This pattern holds consistently across web frameworks (Sanic, Tornado), developer tools (PySnooper, Thonny), data utilities (Mimesis, Dataclasses-JSON),

and automation systems (Ansible), suggesting that these two categories represent a fundamental class of defects that systematically escapes test coverage in Python projects, likely because they manifest in error-handling paths, boundary conditions, and exceptional inputs.

Domain-specific observations. At finer granularity, domain-specific patterns emerge. Security-related defects, specifically *Security and Sensitive Data Leakage Due to Logic Oversight*, with 4 cases, were found in projects that handle external input or perform filesystem and network operations. Both confirmed security vulnerabilities in our case studies arose from uncovered input-handling paths. State consistency and bookkeeping bugs, with 14 cases, were disproportionately present in projects with asynchronous or streaming architectures. API and signature mismatch bugs, with 10 cases, along with type handling and inheritance contract violations, with 6 cases, were most prevalent in large, multi-component projects such as Ansible, where Python 2 to Python 3 compatibility issues introduce type system inconsistencies, as illustrated by the argument splitter bug in Case 4 of Section 7. Less frequent categories include I/O and filesystem handling bugs (1 case) and copy semantics and mutability aliasing bugs (1 case).

IssueSpecter surfaces a diverse spectrum of defect types spanning all nine categories of the adopted taxonomy, with no single category dominating, and with logic errors and input validation failures being the most prevalent across all domains, representing 63% combined, while domain-specific patterns emerge at finer granularity: security and I/O bugs cluster in projects with external input handling, state consistency bugs are more prevalent in asynchronous and streaming architectures, and type mismatch bugs concentrate in large legacy codebases that may be undergoing Python 2-to-3 migration. The consistent prevalence of logic and boundary errors across all domains in our dataset confirms that LLM-based issue generation can be applicable without requiring domain-specific tuning. The diversity of surfaced defect types further suggests that IssueSpecter is not biased toward a narrow class of bugs, making it a viable general-purpose complement to test suites across different Python project contexts.

Finding 4: IssueSpecter detects a diverse spectrum of defect types across nine taxonomy categories, with logic errors and input validation failures dominating (63% combined) regardless of domain, and security, state consistency, and type mismatch bugs clustering in projects with matching architectural characteristics.

Implication 4: The prevalence of logic and boundary errors across all domains suggests that LLM-based issue generation requires no domain-specific tuning. This shows the ability of IssueSpecter to detect diverse types of defects.

6.5 RQ4: Test Coverage Level Influence

Table 5.2 summarises the uncovered segments that IssueSpecter selected as targets for issue generation across the 13 projects, and Table 6.1 reports the corresponding issue volumes, to allow an investigation of the relationship among coverage level, codebase size, and the quantity of generated issues.

Projects with low test coverage naturally expose larger uncovered surfaces. iSort (16.0 % coverage) and Tornado (30.7 % coverage) produced 24 and 284 uncovered segments, yielding 72 and 852 generated issues respectively. Conversely, highly covered projects such as Mimesis (99.1 %) and Dataclasses-JSON (94.8 %) still produced 300 and 55 uncovered segments and 900 and 165 issues respectively, demonstrating that even near-complete test suites leave exploitable gaps that LLM-based analysis can surface.

Codebase size interacts substantially with coverage to determine issue volume. Ansible, with 1,573 files and with 46.2 % of test coverage, dominates the dataset with 5,496 generated issues from 1,832 uncovered segments, over half of all generated issues. This highlights that large codebases with moderate coverage present an important target for automated issue generation.

Most projects cluster around an average of 15 to 25 lines per uncovered segment, suggesting that the uncovered regions tend to be compact, self-contained blocks rather than large untested modules. This compactness may partly explain the high error detection rate reported in RQ1, suggesting that focused, bounded segments provide sufficient context for the LLM to reason about potential defects without the noise of surrounding unrelated code.

Severity distribution does not appear to vary systematically with coverage level. Both low-coverage projects (e.g., iSort, Tornado) and high-coverage projects (e.g., Mimesis, Sanic) produce issues with similar severity profiles dominated by Medium and Low categories, suggesting that the type of uncovered code determines the severity of surfaced defects.

Finding 5: Issue volume is driven by the combined effect of test coverage and codebase size: low-coverage, large projects (e.g., Ansible) generate the most issues, while even near-complete test suites (e.g., Mimesis at 99.1 %) leave exploitable uncovered gaps that contain valid bugs. Uncovered segments tend to be compact (spanning from 15 to 25 lines on average), providing focused targets for LLM-based defect analysis.

Implication 5: Automated issue generation and subsequent ranking may be more impactful for large, under-tested codebases, but remains valuable even for mature, well-tested projects. Coverage percentage alone is not a reliable indicator of residual defect risk; this suggests that developers may benefit from complementing segment-level coverage tools (e.g., SlipCover) alongside LLM-based analysis to identify latent bugs.

A natural question is whether the assumption that uncovered code is more likely to contain bugs holds in practice. We note that test coverage alone is not a reliable indicator of defect absence: a code segment being executed by a test does not imply that the test exercises the correct behavior, as tests may assert incorrect outcomes, rely on excessive mocking, or fail to capture edge cases, a well-known limitation referred to as the oracle problem (Altmayer Pizzorno & Berger, 2025). Moreover, our results empirically support this assumption: across 13 actively maintained Python projects, 84.6% of top-ranked issues generated from uncovered segments were confirmed as valid bugs or warranted further investigation, suggesting that uncovered segments represent a systematic blind spot where latent defects concentrate.

Chapter 7

Case Study

To evaluate bug report quality, two annotators manually reviewed the top-10 most ranked issues for each project, comprising the analysis of 130 issues, using the following scheme: *Valid Bug* represents a high confidence the defect exists, accounting for 37.7% of the top-ranked issues;*Requires Further Investigation* indicates plausible bug when manually inspecting the issue, but not fully reproduced, probably presenting edge cases, accounting for 46.9% of the top-ranked issues; and *Invalid* bugs, accounting for logical inconsistencies, or cases where IssueSpecter misunderstood the project's intended usage patterns, in a way that a user would never encounter the issue while using the project. As shown in Table 6.3, 84.6% of generated reports are worth of investigation, since they represent valid or potential bugs, and 15.4% are confirmed to be false positives. Given the cost of reproducing bugs, we prioritized execution for high confidence Valid Bugs to confirm practical impact, and some of these issues are presented in this Section.

We compare IssueSpecter's findings against the unit tests generated by CoverUp (Altmayer Pizzorno & Berger, 2025) for the same uncovered segments by manually inspecting whether the CoverUp-generated unit test was capable of detecting the same bug identified by IssueSpecter for our 7 case studies. As summarized in Table 7.1, CoverUp failed to detect any of the seven bugs surfaced by IssueSpecter.

Table 7.1: Comparison of IssueSpecter case studies against CoverUp-generated unit tests for the same uncovered segments. In all seven cases, CoverUp failed to detect the bug identified by IssueSpecter.

Case Study Title	**CoverUp Behavior**
Path traversal in HTTP client	Only benign session names tested; traversal payloads are never passed, and path-escape assertions trivially pass.
OOM vulnerability in HTTP client	Payloads of $\leq$16 bytes used; the unbounded accumulation bug requires large responses and a missing safety guard undetectable by structural coverage.
Data loss in Tornado gzip	Concatenated-member gzip input was never tested.
Type mismatch in Ansible splitter	The `bytes`/`str` crash on any normal string input was never exercised.
Sensitive data leakage in Configuration Management Tool	The `__repr__` credential-leakage vulnerability was never exercised.
Incorrect exception detection in PySnooper	Negative-indexing bug triggers were never tested as input
Type constraint violation in Cookiecutter	Only hashable string options tested; the `TypeError` crash requires dict/list options as test input to trigger the bug.

7.1 Case 1: Path Traversal Vulnerability in HTTP Client Library

Using IssueSpecter, we identified a path traversal vulnerability in a widely used HTTP client library. Based on the contribution guideline of the project, we have submitted the

vulnerability to the developer via email and it is currently pending developers' confirmation (we have omitted the project name for security concerns). Figure 7.1 shows the vulnerable function `session_hostname_to_dirname()`, which accepts two string parameters, `hostname` and `session_name` for constructing file paths, but fails to properly validate user input. At line 3 in Figure 7.1, the function performs insufficient validation by replacing colons with underscores in the `hostname` parameter where the `session_name` parameter has no validation. At line 4 in Figure 7.1, both parameters are passed to `os.path.join()` without further checks, which in practice makes it possible for an attacker to supply traversal sequences (e.g., `../`) or absolute paths and write files beyond the intended session directory.

```
1  def session_hostname_to_dirname(hostname: str,
2                                  session_name: str) -> str:
3      hostname = hostname.replace(':', '_')
4      return os.path.join(SESSIONS_DIR_NAME, hostname,
5                          f'{session_name}.json')
```

Figure 7.1: Vulnerable path construction function in a HTTP library

We identify three root causes of this issue: (1) the function does not perform any kind of validation on input length or dangerous characters that could be used for directory traversal; (2) the path construction phase treats user input as always being trustworthy, incorporating it directly into filesystem operations; and (3) no mechanism exists to ensure that the resulting path remains within the intended sessions directory. When reproducing the issue, we generated test artifacts demonstrating that malicious input containing `../../../tmp/ESCAPED` successfully created files outside the expected directory structure, escaping into parent directories in relation to the user's configuration folder. Following prior work of BugPilot (Sonwane et al., 2025), we classify this as a "Sensitive data leakage due to logic oversight" bug, corresponding to CWE-22 (Improper Limitation of a Pathname to a Restricted Directory). Moreover, we consider this a medium to high severity issue because it enables arbitrary file write and read operations outside the intended directory boundaries. Exploiting this vulnerability could allow an attacker to access sensitive data or corrupt critical system files, potentially leading to privilege escalation.

The proposed fix generated by IssueSpecter in Figure 7.2 introduces a validation

mechanism that strips directory traversal components using `os.path.basename()` at line 2 and replacing unsafe characters with the use of a regular expression at line 3, ensuring the function remains secure independently.

```
1  def _sanitize_component(value: str) -> str:
2      value = os.path.basename(value)
3      return re.sub(r'[^A-Za-z0-9._-]', '_', value)
4
5  def session_hostname_to_dirname(hostname: str,
6                                  session_name: str) -> str:
7      hostname = hostname.replace(':', '_')
8      hostname = _sanitize_component(hostname)
9      session_basename = _sanitize_component(session_name)
10     return os.path.join(SESSIONS_DIR_NAME, hostname,
11                         f'{session_basename}.json')
```

Figure 7.2: Fixed path construction with input sanitization proposed by IssueSpecter

The CoverUp-generated test for this uncovered segment parametrizes five hostname and session name combinations, all using safe alphanumeric values such as `"mysession"` and `"multi"`. The test was designed to maximize line coverage, never supplying a path-traversal input such as `"../../../tmp/evil"` and not verifying that the resolved path remains within the sessions directory.

7.2 Case 2: Out of Memory Vulnerability in HTTP Client Library

Using IssueSpecter, we identified an out-of-memory vulnerability in the same HTTP client library from Case 1. Figure 7.3 shows the vulnerable method `BufferedPrettyStream.iter_body()`, which is responsible for formatting response bodies, but accumulating all response chunks into a byte array without any size validation or limits. At line 5 in Figure 7.3, the method iterates through response chunks of fixed size. And at line 11, each chunk is appended to the `body` byte array using `extend()` without verifying if the accumulated size exceeds memory limits, causing out of memory conditions when processing large responses.

Together with the issue, we have included the manually crafted test that reproduced the issue. The test involves creating a response exceeding 1 gigabyte, demonstrating that the method attempts to buffer the entire response until the system exhausts all the available memory, requiring a hard reboot of the hardware.

```
1  def iter_body(self) -> Iterable[bytes]:
2      converter = None
3      body = bytearray()
4
5      for chunk in self.msg.iter_body(self.CHUNK_SIZE):
6          if not converter and b'\0' in chunk:
7              converter = self.conversion.get_converter(
8                                          self.mime)
9              if not converter:
10                 raise BinarySuppressedError()
11         body.extend(chunk)  # no size limit here
12
13     if converter:
14         self.mime, body = converter.convert(body)
15
16     yield self.process_body(body)
```

Figure 7.3: Vulnerable buffering method in the HTTP client library codebase

As root cause of the issue, the method implements an accumulation without any maximum buffer size constraint, with no existing validation to check the total accumulated size before extending the byte array in line 11 depicted in Figure 7.3, where `body.extend(chunk)` is called; and finally, the method continues buffering regardless of available system memory.

Following prior work (Sonwane et al., 2025), we classify this as a state consistency, bookkeeping, and caching bug, and moreover, we consider this a medium severity issue with both availability and security implications. While CLI usage by end users results in local system crashes affecting only the individual user, the vulnerability becomes more severe with the potential of the library being integrated into web services, automated systems, or CI/CD pipelines where malicious inputs could disrupt operations for multiple users.

The proposed fix generated by IssueSpecter in Figure 7.4 introduces a configurable maximum buffer size, with a class constant `MAX_BUFFER_SIZE` defining the buffer limit at line 2 of the code. From lines 16 to 20, the fix adds a size check that raises a descriptive

`ValueError` when the accumulated buffer exceeds the defined limit, allowing the system to fail with an error message rather than exhausting system memory. In addition, this also maintains backward compatibility for responses below the threshold.

```
class BufferedPrettyStream(PrettyStream):
    MAX_BUFFER_SIZE = 10 * 1024 * 1024  # 10 MiB

    def iter_body(self) -> Iterable[bytes]:
        converter = None
        body = bytearray()

        for chunk in self.msg.iter_body(self.CHUNK_SIZE):
            if not converter and b'\0' in chunk:
                converter = self.conversion.get_converter(
                                            self.mime)
                if not converter:
                    raise BinarySuppressedError()
            body.extend(chunk)

            if len(body) > self.MAX_BUFFER_SIZE:
                raise ValueError(
                    f"Response body exceeds maximum "
                    f"buffered size of "
                    f"{self.MAX_BUFFER_SIZE} bytes")

        if converter:
            self.mime, body = converter.convert(body)

        yield self.process_body(body)
```

Figure 7.4: Fixed buffering method with size limit proposed by IssueSpecter

The CoverUp-generated test for this segment employs three small handcrafted payloads, the largest being 14 bytes. The test confirms that `iter_body()` correctly accumulates chunks and processes them, representing a correct behaviour for small inputs. However, the out-of-memory bug is a missing limit check verification: the safety guard that should prevent unbounded accumulation does not exist in the unit test code; thus, it executes correctly on small inputs and raises no error. The IssueSpecter issue identifies this missing checking, explains the real-world impact of a multi-gigabyte response causing a system freeze, and proposes the concrete `MAX_BUFFER_SIZE` guard needed.

7.3 Case 3: Data Loss in Tornado's Gzip Decompressor

IssueSpecter identified a data loss vulnerability in Tornado's gzip decompression implementation. Figure 7.5 shows the vulnerable method `GzipDecompressor.decompress()`, which processes gzip compressed data but only decompresses the first member when handling concatenated gzip streams, discarding all subsequent members. According to the gzip specification, a valid gzip file can contain multiple members concatenated together, and decompressors must process all of them to retrieve complete data.

```
1  def decompress(self, value: bytes,
2                 max_length: int = 0) -> bytes:
3      data = value
4      out = bytearray()
5
6      for chunk in iterate_chunks(data):
7          result = self.decompressobj.decompress(chunk)
8          out.extend(result)
9
10     return bytes(out)
```

Figure 7.5: Vulnerable gzip decompression method in Tornado's codebase

At line 7 in Figure 7.5, the method delegates decompression to `decompress()`, which is a zlib decompressor object. At line 8, the decompressed chunk is appended to the output buffer. However, the method never checks the `unused_data` attribute of the decompressor object, which contains any remaining bytes after the first gzip member ends. As a consequence, when the loop returning at line 10, any subsequent concatenated members remain in `unused_data` but are never processed, resulting in data loss in a silent manner. After submitting the issue to the Tornado repository, another GitHub user added a comment confirming that they have encountered this issue when following the reproduction steps of our report, acknowledging the validity of our submitted issue [1]. Later, the same user opened a pull request adopting the fix we proposed in our issue report [2].

The root cause of this issue is incomplete state tracking in the decompression logic. The method fails to check the `unused_data` attribute of the zlib decompressor object after

[1] https://github.com/tornadoweb/tornado/issues/3560
[2] https://github.com/tornadoweb/tornado/pull/3577

processing, which signals when a gzip member has ended and additional concatenated members remain. By assuming a single member gzip stream and returning immediately after the first decompression completes, the implementation ignores the standard gzip behavior of supporting concatenated members. When reproducing the issue, we generated test artifacts by creating two separate gzip compressed data segments, concatenating them together, and passing them to the decompressor. The result contained only the first segment of the decompressed data, with the second segment completely absent from the output, demonstrating data loss without any warning or error indication.

Following prior work (Sonwane et al., 2025), we classify this as a state consistency, bookkeeping, and caching bug. Moreover, we consider this a medium severity issue because it causes silent data loss in production systems that rely on multi-member gzip streams, a pattern commonly encountered in HTTP streaming responses and various streaming applications.

The proposed fix generated by IssueSpecter in Figure 7.6 introduces logic to handle concatenated gzip members, in which at line 9, after extending the output buffer, the fix checks the `unused_data` attribute using `getattr()`. From lines 12 to 16, when unused data is detected, the fix reassigns it to the `data` variable, creating a new decompressor instance to process the next member, and continuing the loop. This continues until all concatenated members are decompressed.

```
def decompress(self, value: bytes,
               max_length: int = 0) -> bytes:
    data = value
    out = bytearray()

    while True:
        result = self.decompressobj.decompress(data)
        out.extend(result)

        unused = getattr(self.decompressobj,
                         "unused_data", b"")
        if unused:
            data = unused
            self.decompressobj = zlib.decompressobj(
                                    16 + zlib.MAX_WBITS)
            continue

        break

    return bytes(out)
```

Figure 7.6: Fixed gzip decompression method with concatenated member handling proposed by IssueSpecter

This bug was not addressed by CoverUp, since `GzipDecompressor.decompress()` was never directly exercised by any generated unit test.

7.4 Case 4: Type Mismatch Bug in Ansible's Argument Splitter

Using IssueSpecter, we identified a type mismatch bug in Ansible's argument parsing module, with Figure 7.7 showing the vulnerable function `split_args()`, in which crashes with a `TypeError` on Python 3 when processing normal string input. This breaks argument parsing for module invocations using legacy syntax. At line 2 in Figure 7.7, the function unconditionally encodes string input to bytes using UTF-8 encoding. At line 4, the function attempts to split the converted object to bytes using a string delimiter `'\n'`. However, on Python 3, bytes objects cannot be split using string delimiters, causing a `TypeError` that crashes the function. This same issue occurs throughout the function wherever string literals are used as delimiters or in string operations on the bytes object.

```
1  def split_args(args):
2      args = args.encode('utf-8')  # args is now bytes
3      do_decode = True
4      items = args.split('\n')  # TypeError: can't split
5                                # bytes with str delimiter
6
7      # ... rest of function uses string operations on bytes
```

Figure 7.7: Vulnerable argument splitting function in Ansible's codebase

The root cause of this issue is a Python 2 to Python 3 compatibility problem, with the encode and decode logic appearing to be an attempt done using Python 2 to normalize character encoding handling, but it breaks Python 3 compatibility where strings and bytes are separated types. Python 3 code should operate on strings by default, and only decode bytes when explicitly receiving them as input.

Following prior work (Sonwane et al., 2025), we classify this as a type handling bug, and also consider this as a high severity issue because it breaks argument parsing functionality for any module invocation using legacy syntax, affecting core Ansible functionality on Python 3 environments.

The proposed fix generated by IssueSpecter in Figure 7.8 removes unnecessary encode logic and normalizes input handling. From lines 2 to 7, instead of encoding strings to bytes,

the fix checks if the input is already bytes and only then decodes it to a string. On line 9, the function strips the string input, and on line 10, the function now correctly splits a string with a string delimiter, eliminating the type mismatch.

```
1  def split_args(args):
2      # normalize input to str, decode bytes if necessary
3      if isinstance(args, bytes):
4          try:
5              args = args.decode('utf-8')
6          except UnicodeDecodeError:
7              args = args.decode('latin-1')
8
9      args = args.strip()
10     items = args.split('\n')  # correctly splits str
11                               # with str delimiter
12
13     # ... rest of function remains unchanged
```

Figure 7.8: Fixed argument splitting function with proper type handling proposed by IssueSpecter

The `bytes/str` crash on a string input was never exercised by CoverUp-generated unit tests for this case study.

7.5 Case 5: Incorrect Exception Detection in PySnooper's Tracer

IssueSpecter identified an incorrect exception detection bug in PySnooper's tracing system. Figure 7.9 shows the vulnerable code in the `trace()` function, which incorrectly reports "Call ended by exception" for normal function returns when the frame's last instruction index is -1. At line 1 in Figure 7.9, the code accesses `frame.f_code.co_code[frame.f_lasti]` without validating whether the index is non negative. In Python's tracing system, `frame.f_lasti` indicates the last instruction executed, but this value can be -1 in certain cases like early function calls, some exception states, or in specific Python versions; thus, when `f_lasti` equals -1, Python's negative indexing mechanism returns the last byte of the bytecode instead of raising an error.

Opcodes are low-level bytecode instructions executed by Python, and PySnooper's tracer

```
code_byte = frame.f_code.co_code[frame.f_lasti]
if not isinstance(code_byte, int):
    code_byte = ord(code_byte)
ended_by_exception = (
    event == 'return'
    and arg is None
    and opcode.opname[code_byte] not in RETURN_OPCODES
)
```

Figure 7.9: Vulnerable exception detection code in PySnooper's tracer

examines the opcode at the last executed instruction to determine whether a function ended normally via a return statement or abnormally due to an exception. The root cause of this issue is the lack of boundary validation before array indexing, with the code assuming `frame.f_lasti` is always a valid positive index into the bytecode array, but Python's tracing API does not guarantee this. Then, when `f_lasti` is -1, the negative index accesses the wrong position in the bytecode, retrieving an arbitrary opcode that may not be in the set of return opcodes. This causes the condition `opcode.opname[code_byte] not in RETURN_OPCODES` to evaluate to true even for normal returns, incorrectly flagging them as exceptions.

When reproducing the issue, we created a mock frame with `f_lasti = -1` and bytecode ending with a NOP instruction, as shown in Figure 7.10. It confirmed that a normal return event was incorrectly classified as an exception because the negative index read the NOP opcode instead of the actual return instruction.

```
# Create bytecode with NOP as last instruction
bytecode = bytes([
    opcode.opmap['LOAD_CONST'], 0,
    opcode.opmap['RETURN_VALUE'], 0,
    opcode.opmap['NOP'], 0,  # last byte
])

frame = Mock()
frame.f_code.co_code = bytecode
frame.f_lasti = -1  # triggers negative indexing

# Buggy code reads NOP (last byte) instead of
# RETURN\_VALUE, misclassifying the return
result = buggy_ended_by_exception(frame, 'return', None)
# Returns True (wrong classification)
```

Figure 7.10: Test case demonstrating incorrect opcode reading with negative index

Following prior work (Sonwane et al., 2025), we classify this as an Input Validation, Boundary, and Sentinel Handling Error. Moreover, we consider this a medium severity issue because it produces incorrect debugging information that can mislead developers during debugging sessions, potentially causing them to investigate non-existent exceptions.

The proposed fix generated by IssueSpecter in Figure 7.11 adds comprehensive validation before accessing the bytecode. On line 2, the fix uses `getattr()` with a default value of -1 and checks that `f_lasti >= 0` before attempting to access the array. From lines 3 to 10, the fix wraps the bytecode access in a try-except block for additional safety, handling any unexpected errors gracefully. At lines 12-18, similar defensive coding is applied when looking up the opcode name. At lines 20-23, the final exception detection logic only proceeds if a valid opcode name was successfully retrieved. This defensive approach prevents incorrect classification and ensures the tracer behaves correctly even when encountering edge cases with invalid instruction indices.

```
code_byte = None
if getattr(frame, 'f_lasti', -1) >= 0:
    try:
        code_byte = frame.f_code.co_code[frame.f_lasti]
    except Exception:
        code_byte = None
    else:
        if not isinstance(code_byte, int):
            code_byte = ord(code_byte)

opcode_name = None
if code_byte is not None:
    try:
        opcode_name = opcode.opname[code_byte]
    except Exception:
        opcode_name = None

ended_by_exception = (
    event == 'return'
    and arg is None
    and (opcode_name is not None
         and opcode_name not in RETURN_OPCODES)
)
```

Figure 7.11: Fixed exception detection with boundary validation proposed by IssueSpecter

Negative input values, such as `lasti=-1`, were never tested as input by CoverUp. In addition, detecting this boundary condition requires knowing that Python's `frame.f_lasti` can legally be negative, an implementation detail of the CPython tracing API, a peculiarity that IssueSpecter was able to detect.

7.6 Case 6: Type Constraint Violation in Cookiecutter's Prompt Handler

IssueSpecter identified a type constraint violation in Cookiecutter's prompt handling system. Figure 7.12 shows the vulnerable code in the `read_user_choice()` function, which crashes with a `TypeError` when option values are unhashable types such as dictionaries or lists and custom prompts are provided through the CLI. From lines 7 to 9 in Figure 7.12, when custom prompts are configured, the code attempts to check if an option value `p` exists

in the prompts mapping using `if p in prompts[var_name]`; however, this check uses the option value directly as a dictionary lookup key. On line 7, when `p` is an unhashable type like a dictionary, Python raises a `TypeError` because dictionaries and lists cannot be used as dictionary keys due to their mutable nature.

```
1  if prompts and var_name in prompts:
2      # ... extract question ...
3      choice_lines = (
4          f"[bold magenta]{i}[/] - [bold]"
5          f"{prompts[var_name][p]}[/]"
6          if p in prompts[var_name]    # crashes if p is
7                                       # unhashable (dict/list)
8          else f"[bold magenta]{i}[/] - [bold]{p}[/]"
9          for i, p in choice_map.items()
10     )
```

Figure 7.12: Inflexible prompt handling code in Cookiecutter's codebase

The root cause of this issue is an implicit assumption about data types in the prompt mapping logic. The code assumes all option values are hashable types that can be used as dictionary keys, but Cookiecutter's configuration format allows arbitrary JSON structures as option values, including nested dictionaries and lists. In Python, only immutable types like strings, numbers, and tuples can be used as dictionary keys, while mutable types like dictionaries and lists are unhashable. Thus, when users configure templates with structured configuration options, such as cloud provider settings containing multiple nested parameters, the direct dictionary lookup fails. When reproducing the issue, we created a Cookiecutter template with cloud configuration options (such as containing provider, region, and instance count fields) represented as dictionaries. Attempting to use custom prompts with these options triggered the `TypeError: unhashable type: 'dict'` crash, preventing the template generation.

Following prior work (Sonwane et al., 2025), we classify this as a type handling bug. Moreover, we consider this a medium severity issue because it prevents users from creating templates with structured configuration options, limiting Cookiecutter's flexibility for complex use cases requiring hierarchical settings.

The proposed fix generated by IssueSpecter in Figure 7.13 introduces a helper function

to handle both hashable and unhashable option values in a safer manner. From lines 3 to 10 in Figure 7.13, the `_label_for()` function attempts to look up the label using the raw value within a try-except block. On line 5, if the value is unhashable and raises a `TypeError`, the exception is caught, and the function falls back to using the string representation of the value as the lookup key at line 8, and if no custom label is found, the function returns the string representation of the value as a fallback. Then, from lines 12 to 14, this function is used in the generator expression to retrieve labels for all option types, eliminating the crash.

```
1  if prompts and var_name in prompts:
2      # ... extract question ...
3      def _label_for(value):
4          try:
5              label = prom_map.get(value)
6          except TypeError:  # value is unhashable
7              label = None
8          if label is None:
9              label = prom_map.get(str(value))
10         return label if label is not None else str(value)
11
12     choice_lines = (
13         f"[bold magenta]{i}[/] - [bold]{_label_for(value)}[/]"
14         for i, value in choice_map.items()
15     )
```

Figure 7.13: Fixed prompt handling with unhashable type support proposed by IssueSpecter

The CoverUp-generated test for this segment reaches all target lines and most branches, using four test cases that all pass string options such as `["opt1", "opt2"]`. Strings are always hashable, so the `if p in prompts[var_name]` check never raises a `TypeError`. The test exercises the same branch as the bug but with safe inputs, as shown below:

```
# CoverUp test_coverup_4.py - all test cases use string options
options = ["opt1", "opt2"]   # hashable, no crash
options = ["a", "b"]
options = ["alpha", "beta"]

# The bug: dict options cause TypeError at this line
if p in prompts[var_name]:   # p is a dict -> TypeError
```

Figure 7.14: CoverUp test uses only hashable string options, missing the unhashable type bug in Cookiecutter's prompt handler

CoverUp's input selection picks the simplest type that reaches a branch (a string), aiming to improve branch coverage, and misses the case where the public API accepts any list element as an option value, including unhashable dicts. IssueSpecter identifies the precise input class that breaks the invariant and demonstrates that this is a real scenario, such as structured cloud configuration options in a Cookiecutter template.

7.7 Case 7: Security Vulnerability in Configuration Management Tool

Our pipeline revealed a security vulnerability in a popular configuration management project, whose name is anonymized. As an additional context, a `__repr__` method is the official string representation of an object in Python. In this case study, a `__repr__` method in a handler class that contains a Python dictionary variable that stores sensitive information such as passwords, API tokens, and private keys through object representations, without any type of filtering to hide these sensitive information, creating an opportunity for unintentional information disclosure. Thus, this dictionary is populated from multiple sources, including configuration parameters provided by users, loop variables and variables marked with internal logging suppression flags (the latter are variables that have been explicitly tagged by developers or users to indicate they contain sensitive information and

should not appear in logs or output).

This is classified as a "Security / sensitive - data leakage due to logic oversight" bug (Sonwane et al., 2025), corresponding to the CWE-532 vulnerability (Insertion of Sensitive Information into Log File). Moreover, this is a medium severity security issue because it enables credential exposure through multiple vectors including exception tracebacks, debug logging, interactive debugging sessions, and monitoring systems that may capture object representations.

By manually inspecting other files belonging to the codebase of this project, we noticed that the development team had already implemented protective measures to prevent variable dumping in other files. But this same protection was not extended to the file inclusion handler class that this case study addresses, creating a security gap that may be silent for developers and users. In our issue generation pipeline, the proposed code for fixing this issue generated by the GPT-5-mini aligns with existing security practices already established in other parts of the system, by implementing sensitive key detection and omission in the `__repr__` method while using Python's `reprlib` module to limit output size and prevent excessive data exposure. CoverUp, in contrast, did not exercise this credential-leakage vulnerability found by IssueSpecter.

Chapter 8

Threats to Validity

To reduce bias in the ranking process and bug-type annotation, disagreements between the two annotators were resolved through discussion. The LLM employed for issue generation and ranking (GPT-5-mini) is inherently non-deterministic, which can lead to variability in outputs across different runs. Additionally, the rule-based ranking approach depends on heuristics—such as word count as an indicator of completeness that were based on previous work (Alenezi & Banitaan, 2013) to capture the true quality of an issue. Our prompt design also limits the number of reported bugs per segment to three, which may result in some defects being overlooked.

Moreover, the LLM may demonstrate a tendency toward confirmation bias, identifying defects even in correct code segments, thereby increasing the likelihood of false positives. However, our approach shows a false positive rate of only 19%, which is slightly lower than the invalid rate of 23.8% observed in our comparison results with CoverUp (Altmayer Pizzorno & Berger, 2025), a state-of-the-art test generation tool evaluated under the same conditions. This suggests that, when combined with a ranking and filtering stage, the confirmation bias introduced by bug-targeted prompting does not substantially inflate false positives in practice.

An alternative design to IssueSpecter would be to combine multiple uncovered segments into a single prompt, potentially enabling the detection of inter-segment bugs that span multiple code blocks. However, this approach introduces two practical challenges: (1)

combining segments risks exceeding the LLM's context window, reducing the quality and coherence of the generated reports; and (2) bugs that span multiple uncovered segments are difficult to reproduce and localize. We therefore treat each uncovered segment independently, acknowledging that cross-segment defects represent a limitation of the current approach and a direction for future work.

Finally, the evaluation focuses exclusively on Python projects, which restricts the generalizability of the results to other programming languages. Differences in language features, ecosystems, and testing practices may affect both the nature of detected bugs and the performance of LLM-based analysis. By keeping the programming language constant while varying project domains, we aim to isolate the impact of domain diversity on bug detection without introducing confounding language-specific factors.

Chapter 9

Conclusion and Future Work

Uncovered code segments often contain hidden defects, as they are neither executed by test suites nor rigorously reviewed by developers. IssueSpecter builds on this observation through a simple but effective pipeline: it identifies uncovered segments via coverage analysis, tasks an LLM with detecting and reporting defects within those segments, filters and ranks the resulting reports by severity and impact, and presents developers with a prioritized, structured list of probable bugs, without requiring any modifications to the existing test suite. For developers with limited time and resources, this prioritization lets them focus on the most probable impactful defects.

Our results demonstrate that, despite the tendency of LLMs to over-report defects, automated issue generation has the potential of identifying real bugs. With 84.6% of top-ranked reports either confirmed as valid bugs or warranting further investigation, and with case studies reproducing real-world vulnerabilities such as a path traversal vulnerability (CWE-22) and a silent data loss bug in a gzip decompressor, IssueSpecter shows that false positives need not undermine the practical value of LLM-based defect detection, since effective ranking and filtering stages can be employed. The human-readable nature of generated issue reports further lowers the barrier for developers to quickly act upon the surfaced bugs, even without deep familiarity with the affected code.

Although IssueSpecter relies on an LLM to generate issue reports, it does not eliminate the developer from the process. To mitigate the risk of blindly trusting AI-generated

outputs, our pipeline incorporates a rule-based ranking stage grounded in established bug prioritization criteria (Alenezi & Banitaan, 2013), selecting issues based on severity, platform scope, and description completeness: properties that reflect issue quality independently of the LLM's judgment. Furthermore, our case studies demonstrate that IssueSpecter is capable of surfacing real, reproducible bugs in actively maintained open-source projects.

Our findings highlight that LLM-based ranking provides substantial improvements over rule-based heuristics alone, and that the diversity of surfaced defect types spans all nine categories of the adopted taxonomy (Sonwane et al., 2025) without any domain-specific tuning. IssueSpecter demonstrates effectiveness across projects of vastly different characteristics: from compact, highly tested codebases such as Mimesis with 99.1% coverage and 90 files, to large, undertested projects such as Ansible with over 1,500 files and 46.2% coverage, and across diverse programming paradigms spanning web frameworks, automation platforms, developer tools, and data utilities. This robustness across project sizes, maturity levels, and domain contexts suggests that IssueSpecter does not require domain-specific tuning or a particular level of testing maturity to be effective, making it a broadly applicable complement to existing test suites.

Directions remain open for future investigation, since IssueSpecter is currently limited to Python projects, as it relies on SlipCover for coverage analysis. Thus, extending the pipeline to support other programming languages and their respective coverage tools would broaden its applicability.

In addition, the issue generation phase currently prompts the LLM to identify up to three defects per uncovered segment, and future work could explore strategies that dynamically adjust this limit based on segment characteristics or at the repository level, tied to the project characteristics.

Since our evaluation relies on manual annotation of the top-ranked issues as ground truth, future work could investigate automated or semi-automated validation mechanisms to more rigorously assess the correctness of the suggested repairs. Currently, IssueSpecter only verifies that the proposed fix does not break the project's existing test suite, which is a necessary condition for correctness, but more rigorous validations can be explored.

One direction would be to generate unit tests alongside the issue generation phase, and execute the proposed fixes against these extended test suites to provide stronger evidence that the fix is correct and does not introduce regressions. Another direction would be to leverage mutation testing, a technique that introduces small artificial faults into the code and checks whether the test suite detects them. In the context of IssueSpecter, mutation testing could be used to evaluate whether a generated fix genuinely addresses the identified defect, by verifying that mutants introduced around the fixed code are caught by the tests. Together, these approaches could provide a more scalable and objective measure of identifying defects.

In addition, due to its advantageous simplicity of receiving a self-contained code segment as input into the prompt, most prominent source of false positives in IssueSpecter derives from the LLM receiving only the uncovered code segment in isolation, without access to broader repository-level context, such as how the project is intended to be used, previously reported issues, or the surrounding architectural conventions of the codebase. Future work could address this limitation by incorporating searches or agentic workflows in which the LLM is able to navigate the repository and retrieve relevant context. Moreover, our pipeline flags documentation inconsistencies along with code-level defects, yet these were not analyzed in our current evaluation. [1]

[1]The author used ChatGPT (OpenAI) for the purpose of correcting grammar and improving wording.

Appendix A

Example of a Generated Issue Report

To illustrate the structure and content of IssueSpecter's generated issue reports, we present the full report produced for Case Study 1, described in Section 7, which corresponds to a path traversal vulnerability (CWE-22) identified in a widely used HTTP client library. The report exemplifies the structured format generated by IssueSpecter, including a detailed description of the vulnerability, its root causes, step-by-step reproduction instructions, expected behavior, and a proposed Python code fix with input validation. This format is representative of the reports produced across all projects in our evaluation, demonstrating how IssueSpecter presents probable defects in a human-readable and immediately actionable form.

Issue Report — session_hostname_to_dirname allows unsafe path components and is vulnerable to path traversal

bug security CWE-22

Severity: High **Type:** Path Traversal

Description

The function `session_hostname_to_dirname` builds a filesystem path from user-provided `hostname` and `session_name` values. It currently only replaces `:` with `_` on the hostname and then joins with `SESSIONS_DIR_NAME`. This is insufficient to prevent path traversal or other unsafe filename/path manipulations. If `session_name` or `hostname` contain path separators, absolute-path prefixes, or traversal sequences (e.g. `\textcolor{red!70!black}{../}`), `os.path.join`/`os.path.basename` behavior can lead to files being created or read outside the intended sessions directory.

Root cause(s)

- No sanitization of `session_name` (and limited sanitization of `hostname`) before using them as path components.
- Not preventing values that start with a path separator or contain `../` sequences.

Steps to reproduce

(1) Call `session_hostname_to_dirname('example.com', '../etc/passwd')`

(2) Observe that the resulting path may point outside the intended sessions directory (depending on the environment and values), potentially enabling an attacker to read/write arbitrary files.

Example (Linux)

```
>>> session_hostname_to_dirname('host', '../etc/passwd')
'sessions/host/../etc/passwd.json'  # if later used for file
    creation, may write outside sessions dir
```

Expected behavior

- The function should return a path rooted under the sessions directory and never permit upward traversal or absolute paths.

- Hostname and `session_name` should be sanitized to strip path parts and replace unsafe characters with safe alternatives.

Proposed fix (description)

- Normalize and sanitize `hostname` and `session_name` before composing the path:
  - Strip path components (`os.path.basename`) so embedded directories are removed.
  - Replace characters that are not safe in filenames (allow only alphanumerics, dot, underscore, hyphen) with a safe placeholder such as underscore.
  - Ensure `session_name` gets a `.json` extension, but keep it sanitized first.
- Use a known sessions directory variable (fallback if undefined) and join sanitized components.

Proposed Python code fix (complete function)

```
import os
import re

def _sanitize_component(value: str) -> str:
    """Return a filesystem-safe basename for a path
       component.
    - Strip directory components via os.path.basename
    - Replace unsafe characters with underscore
    """
    if not isinstance(value, str):
        value = str(value)
    # basename removes any leading path components and
       prevents absolute paths
```

```
    value = os.path.basename(value)
    # allow alphanumerics, dot, underscore, hyphen; replace others with '_'
    return re.sub(r'[^A-Za-z0-9._-]', '_', value)

def session_hostname_to_dirname(hostname: str, session_name: str) -> str:
    # host:port => host_port (replace colons first so IPv6/port delimiters become safe)
    hostname = hostname.replace(':', '_')
    hostname = _sanitize_component(hostname)
    session_basename = _sanitize_component(session_name)
    # ensure .json extension (if not present)
    if not session_basename.lower().endswith('.json'):
        session_basename = f"{session_basename}.json"
    sessions_dir = globals().get('SESSIONS_DIR_NAME', 'sessions')
    return os.path.join(sessions_dir, hostname, session_basename)
```

Other notes

This change preserves the original return format but ensures returned paths are constrained to safe components. It mitigates path traversal and prevents malicious or malformed names from escaping the sessions directory.

Figure A.1: Full IssueSpecter-generated issue report for the path traversal vulnerability (CWE-22) in `session_hostname_to_dirname` (Case Study 1, Section 7).

References

Acharya, J., & Ginde, G. (2025). Can we enhance bug report quality using llms?: An empirical study of llm-based bug report generation. In *Proceedings of the 29th international conference on evaluation and assessment in software engineering* (p. 994–1003). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3756681.3756995 doi: 10.1145/3756681.3756995

Adapa, C., A R K, A., Rahim, R., & Victor, A. (2025). A multi-agent ai framework for agile workflow automation, issue resolution, and developer performance evaluation. In *2025 ieee international conference for women in innovation, technology and entrepreneurship (icwite)* (p. 1-6). doi: 10.1109/ICWITE64848.2025.11306978

Alami, A., Jensen, V., & Ernst, N. (2025, October). Accountability in code review: The role of intrinsic driversnbsp;and the impact of llms. *ACM Trans. Softw. Eng. Methodol.*, *34*(8). Retrieved from https://doi.org/10.1145/3721127 doi: 10.1145/3721127

Alenezi, M., & Banitaan, S. (2013). Bug reports prioritization: Which features and classifier to use? In *2013 12th international conference on machine learning and applications* (Vol. 2, p. 112-116). doi: 10.1109/ICMLA.2013.114

AlOmar, E. A., AlOmar, S. A., & Mkaouer, M. W. (2023). On the use of static analysis to engage students with software quality improvement: An experience with pmd. In *2023 ieee/acm 45th international conference on software engineering: Software engineering education and training (icse-seet)* (p. 179-191). doi: 10.1109/ICSE-SEET58685.2023.00023

Alshahwan, N., Chheda, J., Finogenova, A., Gokkaya, B., Harman, M., Harper, I., ...

Wang, E. (2024). Automated unit test improvement using large language models at meta. In *Companion proceedings of the 32nd acm international conference on the foundations of software engineering* (p. 185–196). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3663529.3663839 doi: 10.1145/3663529.3663839

Altmayer Pizzorno, J., & Berger, E. D. (2023). Slipcover: Near zero-overhead code coverage for python. In *Proceedings of the 32nd acm sigsoft international symposium on software testing and analysis* (p. 1195–1206). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3597926.3598128 doi: 10.1145/3597926.3598128

Altmayer Pizzorno, J., & Berger, E. D. (2025, June). Coverup: Effective high coverage test generation for python. *Proc. ACM Softw. Eng.*, *2*(FSE). Retrieved from https://doi.org/10.1145/3729398 doi: 10.1145/3729398

Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J. D., & Penix, J. (2008). Using static analysis to find bugs. *IEEE Software*, *25*(5), 22-29. doi: 10.1109/MS.2008.130

Bhatia, S., Gandhi, T., Kumar, D., & Jalote, P. (2024). Unit test generation using generative ai: A comparative performance analysis of autogeneration tools. In *Proceedings of the 1st international workshop on large language models for code* (p. 54–61). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3643795.3648396 doi: 10.1145/3643795.3648396

Bo, L., Ji, W., Sun, X., Zhang, T., Wu, X., & Wei, Y. (2024). Chatbr: Automated assessment and improvement of bug report quality using chatgpt. In *Proceedings of the 39th ieee/acm international conference on automated software engineering* (p. 1472–1483). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3691620.3695518 doi: 10.1145/3691620.3695518

Cai, Z., Chen, J., Chen, W., Wang, W., Zhu, X., & Ouyang, A. (2024, 05). F-codellm: A federated learning framework for adapting large language models to practical software development. In (p. 416-417). doi: 10.1145/3639478.3643533

Chapman, P. J., Rubio-González, C., & Thakur, A. V. (2024). Interleaving static analysis and llm prompting. In *Proceedings of the 13th acm sigplan international workshop on the state of the art in program analysis* (p. 9–17). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3652588.3663317 doi: 10.1145/3652588.3663317

Chen, J., Xiang, H., Zhao, Z., Li, L., Zhang, Y., Ding, B., ... Xiong, S. (2025). *Utilizing precise and complete code context to guide llm in automatic false positive mitigation.* Retrieved from https://arxiv.org/abs/2411.03079

Chen, S., Lin, S., Shi, Y., Lian, H., Gu, X., Yun, L., ... Wang, Q. (2026). *Swe-exp: Experience-driven software issue resolution.* Retrieved from https://arxiv.org/abs/2507.23361

Chen, X., He, J., Li, X., He, T., & Chen, Z. (2018, 03). Automated quality assessment for crowdsourced test reports of mobile applications. In (p. 368-379). doi: 10.1109/SANER.2018.8330224

Choi, S., & Yang, G. (2025). Agentreport: A multi-agent llm approach for automated and reproducible bug report generation. *Applied Sciences*, *15*(22). Retrieved from https://www.mdpi.com/2076-3417/15/22/11931 doi: 10.3390/app152211931

Dakhel, A. M., Nikanjam, A., Majdinasab, V., Khomh, F., & Desmarais, M. C. (2024). Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology*, *171*, 107468. Retrieved from https://www.sciencedirect.com/science/article/pii/S0950584924000739 doi: https://doi.org/10.1016/j.infsof.2024.107468

Ehsani, R., Pathak, S., Parra, E., Haiduc, S., & Chatterjee, P. (2025, 11). What characteristics make chatgpt effective for software issue resolution? an empirical study of task, project, and conversational signals in github issues. *Empirical Software Engineering*, *31*. doi: 10.1007/s10664-025-10745-8

El Haji, K., Brandt, C., & Zaidman, A. (2024). Using github copilot for test generation in python: An empirical study. In *Proceedings of the 5th acm/ieee international conference on automation of software test (ast 2024)* (p. 45–55). New York, NY, USA:

Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3644032.3644443 doi: 10.1145/3644032.3644443

Goldman, S., Lin, H. Y., Pasuksmit, J., Thongtanunam, P., Tantithamthavorn, K., Wang, Z., ... Wu, M. (2025). What types of code review comments do developers most frequently resolve? In *2025 40th ieee/acm international conference on automated software engineering (ase)* (p. 3760-3765). doi: 10.1109/ASE63991.2025.00312

He, J., Xu, L., Fan, Y., Xu, Z., Yan, M., & Lei, Y. (2020). Deep learning based valid bug reports determination and explanation. In *2020 ieee 31st international symposium on software reliability engineering (issre)* (p. 184-194). doi: 10.1109/ISSRE5003.2020.00026

Hong, S., Sun, H., Gao, X., & Tan, S. H. (2024). Investigating and detecting silent bugs in pytorch programs. In *2024 ieee international conference on software analysis, evolution and reengineering (saner)* (p. 272-283). doi: 10.1109/SANER60148.2024.00035

Imtiaz, N., Murphy, B., & Williams, L. (2019). How do developers act on static analysis alerts? an empirical study of coverity usage. In *2019 ieee 30th international symposium on software reliability engineering (issre)* (p. 323-333). doi: 10.1109/ISSRE.2019.00040

Jin, H., & Chen, H. (2026). *Are llms reliable code reviewers? systematic overcorrection in requirement conformance judgement.* Retrieved from https://arxiv.org/abs/2603.00539

Jiri, M., Emese, B., & Medlen, P. (2024). Leveraging large language models for python unit test. In *2024 ieee international conference on artificial intelligence testing (aitest)* (p. 95-100). doi: 10.1109/AITest62860.2024.00020

Khan, T. I., Wang, S., Zhang, H., & Chen, T.-H. (2026). *A survey of code review benchmarks and evaluation practices in pre-llm and llm era.* Retrieved from https://arxiv.org/abs/2602.13377

Krodinger, L., Lukasczyk, S., & Fraser, G. (2025). *Combining type inference and automated unit test generation for python.* Retrieved from https://arxiv.org/abs/

2507.01477

Lemieux, C., Inala, J. P., Lahiri, S. K., & Sen, S. (2023). Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 ieee/acm 45th international conference on software engineering (icse)* (p. 919-931). doi: 10 .1109/ICSE48619.2023.00085

Li, H., Hao, Y., Zhai, Y., & Qian, Z. (2024, April). Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proc. ACM Program. Lang.*, *8*(OOPSLA1). Retrieved from https://doi.org/10.1145/3649828 doi: 10.1145/ 3649828

Li, S., Wang, D., Thongtanunam, P., Wang, Z., Yu, J., & Chen, J. (2025). *Issue-oriented agent-based framework for automated review comment generation.* Retrieved from https://arxiv.org/abs/2511.00517

Li, Z., Dutta, S., & Naik, M. (2025). Llm-assisted static analysis for detecting security vulnerabilities. In *International conference on learning representations.* Retrieved from https://arxiv.org/abs/2405.17238

Long, J., Qin, R., Jiang, Z., Duan, J., Li, S., & Qu, X. (2025). A python unit test generation method based on fine-tuned language models and coverage. In *2025 18th international congress on image and signal processing, biomedical engineering and informatics (cisp-bmei)* (p. 1-6). doi: 10.1109/CISP-BMEI68103.2025.11259411

Lu, J., Jiang, L., Li, X., Fang, J., Zhang, F., Yang, L., & Zuo, C. (2025). Towards practical defect-focused automated code review. In *Proceedings of the 42nd international conference on machine learning.* JMLR.org.

Lukasczyk, S., & Fraser, G. (2022). Pynguin: automated unit test generation for python. In *Proceedings of the acm/ieee 44th international conference on software engineering: Companion proceedings* (p. 168–172). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3510454.3516829 doi: 10 .1145/3510454.3516829

Lukasczyk, S., Kroiß, F., & Fraser, G. (2023, 01). An empirical study of automated unit test generation for python. *Empirical Software Engineering*, *28*. doi: 10.1007/

s10664-022-10248-w

Mhatre, A., Nader, N., Diehl, P., & Gupta, D. (2026). *Can llms find bugs in code? an evaluation from beginner errors to security vulnerabilities in python and c++.* Retrieved from https://arxiv.org/abs/2508.16419

Mo, T., Li, P., & Jiang, Z. (2024, May). Comparative analysis of large language models' performance in identifying different types of code defects during automated code review. *Annals of Applied Sciences*, *5*(1). Retrieved from https://annalsofappliedsciences.com/index.php/aas/article/view/28

Mohajer, M. M., Aleithan, R., Harzevili, N. S., Wei, M., Belle, A. B., Pham, H. V., & Wang, S. (2024). Effectiveness of chatgpt for static analysis: How far are we? In *Proceedings of the 1st acm international conference on ai-powered software* (p. 151–160). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3664646.3664777 doi: 10.1145/3664646.3664777

Mohialden, Y., Mahmood Hussien, N., Baker, E., & Joshi, K. (2023, 08). A comparative analysis of python code-line bug-finding methods. , *3*.

Pan, J., Wang, X., Neubig, G., Jaitly, N., Ji, H., Suhr, A., & Zhang, Y. (2024). Training software engineering agents and verifiers with swe-gym. *ArXiv*, *abs/2412.21139*. Retrieved from https://api.semanticscholar.org/CorpusID:275133330

Roychoudhury, A., Pasareanu, C., Pradel, M., & Ray, B. (2025, 02). *Ai software engineer: Programming with trust.* doi: 10.48550/arXiv.2502.13767

Schäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2024a). An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, *50*(1), 85-105. doi: 10.1109/TSE.2023.3334955

Schäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2024b). An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, *50*(1), 85-105. doi: 10.1109/TSE.2023.3334955

Schäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2024c). An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, *50*(1), 85-105. doi: 10.1109/TSE.2023.3334955

Sherifi, B., Slhoub, K., & Nembhard, F. (2025, 10). The potential of large language models in automating software testing: From generation to reporting. In (p. 199-211). doi: 10.1007/978-3-032-08649-5_13

Sobania, D., Briesch, M., Hanna, C., & Petke, J. (2023, 01). *An analysis of the automatic bug fixing performance of chatgpt.* doi: 10.48550/arXiv.2301.08653

Sonwane, A., White, I., Lee, H., Pereira, M., Caccia, L., Kim, M., ... others (2025). Bugpilot: Complex bug generation for efficient learning of swe skills. *arXiv preprint arXiv:2510.19898*.

Sun, T., Xu, J., Li, Y., Yan, Z., Zhang, G., Xie, L., ... Zhu, Y. (2025). Bitsai-cr: Automated code review via llm in practice. In *Proceedings of the 33rd acm international conference on the foundations of software engineering* (p. 274–285). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3696630.3728552 doi: 10.1145/3696630.3728552

Tanzil, M. H., Khan, J. Y., & Uddin, G. (2024). Chatgpt incorrectness detection in software reviews. In *Proceedings of the ieee/acm 46th international conference on software engineering.* New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3597503.3639194 doi: 10.1145/3597503.3639194

Tómasdóttir, K. F., Aniche, M., & Van Deursen, A. (2020). The adoption of javascript linters in practice: A case study on eslint. *IEEE Transactions on Software Engineering*, *46*(8), 863-891. doi: 10.1109/TSE.2018.2871058

Widyasari, R., Sim, S. Q., Lok, C., Qi, H., Phan, J., Tay, Q., ... Ouh, E. L. (2020). Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (p. 1556–1560). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3368089.3417943 doi: 10.1145/3368089.3417943

Yang, J., Lieret, K., Jimenez, C. E., Wettig, A., Khandpur, K., Zhang, Y., ... Yang, D. (2025). SWE-smith: Scaling data for software engineering agents. In *The thirty-ninth*

annual conference on neural information processing systems datasets and benchmarks track. Retrieved from https://openreview.net/forum?id=63iVrXc8cC

Yuan, Z., Liu, M., Ding, S., Wang, K., Chen, Y., Peng, X., & Lou, Y. (2024, July). Evaluating and improving chatgpt for unit test generation. *Proc. ACM Softw. Eng.*, *1*(FSE). Retrieved from https://doi.org/10.1145/3660783 doi: 10.1145/3660783

Zhang, H., Zhao, Y., Yu, S., & Chen, Z. (2022). Automated quality assessment for crowdsourced test reports based on dependency parsing. *2022 9th International Conference on Dependable Systems and Their Applications (DSA)*, 34-41. Retrieved from https://api.semanticscholar.org/CorpusID:253123434