International Journal of Engineering and Techniques - Volume 12 Issue 3, May-June - 2026

Automated Bug Detection Using Artificial Intelligence: A Systematic Review of LLM-Enhanced and Agentic Approaches

Ulgade Shivani Sangram[1], Sagar Choudhary[2] ,Rimmy[3]

[1] B.Tech Student, Department of Computer Science and Engineering, Quantum University, Roorkee, Uttarakhand, India

[2] ,[3]Assistant Professor, Department of Computer Science and Engineering, Quantum University, Roorkee, Uttarakhand, India.

Abstract

Software defects remain among the most expensive risks in modern software engineering. Traditional quality assurance depends on static analyzers, dynamic tests, and manual review, yet these methods struggle with semantic complexity, alert fatigue, and limited scalability. Large Language Models (LLMs) and autonomous code agents have introduced a new paradigm for automated bug detection, fault localization, and program repair. This paper presents a systematic review of peerreviewed and widely cited literature on AI-based bug detection published primarily between 2014 and 2026. We apply structured inclusion criteria to twenty-two primary sources, including SWE-bench (ICLR 2024), RepairAgent (ICSE 2025), IRIS (ICLR 2025), and empirical studies on LLM-assisted static analysis. Comparative tables report published metrics only—for example, Claude 2 resolves 1.96% of SWE-bench issues under BM25 retrieval, while RepairAgent repairs 164 Defects4J bugs. A five-layer reference framework and data-flow diagrams model how inputs, retrieval, reasoning, validation, and feedback interact in DevSecOps pipelines. We conclude that hybrid neuro-symbolic systems with human oversight currently offer the most reliable path to deployment, while fully autonomous repair remains experimental for safety-critical software.

Keywords—Automated program repair, bug detection, large language models, static analysis, dynamic analysis, SWEbench, CodeBERT, software quality assurance.

1. INTRODUCTION

Software systems underpin critical infrastructure in healthcare, finance, transportation, and public services. As codebases grow, the probability of latent defects increases. Industry studies report that the cost of fixing defects rises sharply when bugs are discovered late in the lifecycle; production failures can cost orders of magnitude more than defects caught during unit testing or code review. Automated bug detection is therefore a core risk-management discipline, not merely a productivity enhancement.

For decades, software quality assurance has relied on compilers and linters for syntactic errors, static analysis tools such as Infer, Coverity, SonarQube, and Semgrep for pattern-based checks, dynamic fuzzers for crash discovery, and human review for semantic validation. These techniques are mature but imperfect. Static analyzers often flood teams with warnings; empirical studies on PMD, SpotBugs, and SonarQube document persistent false-positive and false-negative issues in maintainerconfirmed bug

reports. Dynamic testing requires reproducible failures and strong oracles. Manual review does not scale to millions of lines updated through continuous integration.

Large Language Models pretrained on vast open-source corpora can reason over code, naturallanguage issue descriptions, stack traces, and documentation jointly. Benchmarks such as SWEbench formalize repository-level repair on real GitHub issues, while agent systems such as RepairAgent plan tool use, gather repair ingredients, and validate patches iteratively. CodeBERT and successor foundation models demonstrate that joint pretraining on code and text improves localization and repair tasks compared with general-purpose language models alone.

This paper asks: How effective, reliable, and deployable are contemporary AI-based approaches for automated bug detection and repair? We answer through a structured literature review that synthesizes static analysis, dynamic analysis, neural program repair, and agentic workflows into a unified technical framework with verifiable citations.

1.1. Problem Statement

Organizations face three intertwined problems: detection latency (bugs found late), localization inaccuracy (root cause unclear across files), and repair unreliability (patches pass narrow tests but fail in production). AI systems may compress these stages, yet probabilistic models introduce hallucinated identifiers, insecure patches, and overfitting to public benchmarks.

1.2. Objectives and Contributions

The objectives of this review are: (1) to classify AI bug-detection research streams using evidence from primary publications; (2) to present a unified technical framework with data-flow diagrams grounded in that synthesis; (3) to compare benchmarks and reported metrics without inventing experimental results; (4) to discuss industrial risks including cost, privacy, and explainability; and (5) to provide adoption guidance for academic project work.

1.3. Paper Organization

Section II reviews prior work from defect prediction through agentic repair. Section III defines methodology and inclusion criteria. Section IV presents the framework and DFD specifications. Section V analyzes published results and comparative performance. Section VI discusses challenges and limitations. Section VII outlines future scope. Section VIII concludes with recommendations.

The software development lifecycle (SDLC) today is dominated by continuous integration and continuous deployment (CI/CD). Teams merge pull requests multiple times per day, run automated builds, execute unit and integration tests, and deploy containers to cloud environments. Each step introduces opportunities for regressions. While test automation catches many failures, flaky tests, incomplete coverage, and environment-specific bugs still allow defects to reach users. Industry surveys consistently rank debugging and maintenance among the most time-consuming development activities. AI-assisted tooling targets this bottleneck by automating triage: summarizing failures, ranking

suspicious files, and proposing candidate fixes for human acceptance.

From an academic perspective, AI bug detection connects programming languages, compilers, formal methods, machine learning, and human-computer interaction. A rigorous review therefore serves two purposes: it documents industrial trends for project evaluation, and it trains students to read primary sources critically rather than accepting vendor marketing claims. Faculty assessment of literature reviews typically examines whether claims are traceable to citations, whether diagrams match the described system, and whether grammar and formatting meet publication standards.

Scope delimitations are important for credibility. This review focuses on general-purpose software in mainstream languages (Python, Java, JavaScript, C/C++). Hardware description languages, purely theoretical verification without AI components, and tangential topics such as documentation generation are excluded unless they directly affect localization or validation quality. We emphasize detection and repair pipelines rather than adjacent code-understanding tasks.

Motivation in the Software Development Lifecycle

Continuous delivery expects every merge to be testable and deployable. When a build breaks, engineers inspect diffs, rerun failing jobs, and search logs. In large repositories the same failure may touch configuration, dependencies, and business logic in different folders. AI-assisted tools sit in the IDE and in CI because they can read both English descriptions and source files. The papers reviewed here ask whether the same models can close GitHub issues or filter static-analysis warnings at scale.

2. LITERATURE REVIEW

TABLE I EVOLUTION PHASES OF AI-ASSISTED BUG DETECTION

Phase	Era	Techniques	Limitation
I	1990s–2000s	Rule-based SAST, linting	High false positives
II	2005–2015	ML defect prediction (SVM, RF)	Module-level, not line-level
III	2015–2022	Seq2seq APR (e.g., SequenceR)	Snippet-level scope
IV	2023–present	Code-LLMs, agents, neurosymbolic	Cost, validation, hallucination

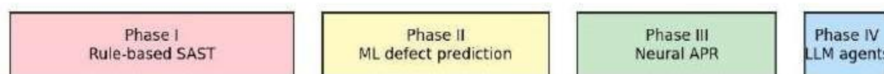

Figure 7: Evolution of automated bug detection paradigms

Fig. 1. Evolution timeline of automated bug detection paradigms.

2.1. Static and Dynamic Analysis

Static Application Security Testing (SAST) analyzes code without execution using data-flow and taint analysis. Dynamic analysis observes runtime behavior through fuzzing and sanitizers. Both paradigms are deterministic and auditable, which keeps them mandatory in regulated industries. They lack, however, high-level reasoning about developer intent expressed in issue trackers. Thung et al. analyze hundreds of confirmed false-positive and false-negative reports from PMD, SpotBugs, and SonarQube, showing that deterministic analyzers alone cannot close the quality gap.

2.2. Machine Learning for Defect Prediction

Before generative AI, defect prediction used metrics such as lines of code, churn, and complexity to rank risky modules. Bird et al. highlight instability and fairness concerns in cross-project prediction. These models prioritize inspection but rarely produce actionable patches and require project-specific training compared with foundation models pretrained on millions of files.

2.3. Code-LLMs and Neuro-Symbolic Analysis

CodeBERT pretrains bidirectional transformers on code and natural language, enabling downstream bug detection and repair tasks without task-specific architectures. Transformer models pretrained on open-source code enable zero-shot explanation, test generation, and patch synthesis. Mohajer et al. evaluate ChatGPT on Infer-generated null-dereference and resource-leak warnings across ten opensource projects, reporting precision up to 93.88% for false-positive removal on nulldereference alerts. Li et al. present IRIS, which combines LLM-inferred taint specifications with CodeQL on Java repositories; on CWE-Bench-Java (120 vulnerabilities), CodeQL detects 27 cases while IRIS with GPT4 detects 55, reducing false discovery rate by five percentage points.

2.4. Agentic Program Repair and SWE-Bench

Jimenez et al. introduce SWE-bench: 2,294 real GitHub issues across twelve Python repositories, requiring multi-file edits and test validation. Under BM25 retrieval, the best model in their study (Claude 2) resolves only 1.96% of issues, demonstrating the difficulty of repository-level repair. Bouzenia et al. propose RepairAgent, an autonomous LLM agent with tools and a finite-state controller; on Defects4J it repairs 164 bugs (39 more than prior techniques), at roughly 270,000 tokens per bug. Yang et al. describe SWE-agent interfaces that structure tool use for repository navigation and patch submission.

2.5. Research Gaps

Gaps include reproducible enterprise benchmarks, standardized cost reporting (tokens and GPU hours), cross-language validation, and explainable traces for security audits. Public leaderboards on SWE-bench Verified show higher resolve rates than the original 2024 baseline, but harness design and data contamination must be reported transparently when citing leaderboard scores.

2.6 Code Review and Modern Quality Assurance

Modern code review remains the gold standard for defect prevention before merge. McIntosh et al. empirically study the impact of review on software quality, showing that review catches logic errors that automated tools miss, while also imposing throughput limits on teams. AI systems are therefore positioned to augment—not replace—review by prioritizing high-risk diffs and summarizing changes. The highest return on investment occurs when AI reduces reviewer fatigue rather than bypassing accountability for security-sensitive changes.

2.7 Pre-LLM Neural Program Repair

SequenceR and related sequence-to-sequence models treat repair as translation from buggy to fixed code snippets, achieving plausible patches on benchmark suites such as Defects4J under constrained edit scopes. Chen et al. demonstrate that neural APR can outperform template-based search on some bug classes, yet remain limited to single-function or single-hunk edits in most evaluations. These systems foreshadow Code-LLM repair but lack natural-language issue understanding and multi-file planning capabilities present in SWE-bench-style tasks.

2.8 Foundation Models for Code

CodeBERT, Codex, and successor models established that transformers pretrained on code and text jointly improve completion, summarization, and bug-fix tasks. Codex evaluations on HumanEval show strong functional correctness on algorithmic problems, but application-level bugs require repository context, dependency management, and project-specific tests. The gap between HumanEval-style success and SWE-bench resolve rates illustrates why repository-level benchmarks became necessary.

Zhang et al. survey learning-based automated program repair, categorizing generate-and-validate, semantic-based, and heuristic search approaches. Their taxonomy helps place LLM agents as an extension of generate-and-validate pipelines with learned priors over code edits rather than purely random mutation. Understanding this lineage prevents overstating novelty of current agent frameworks.

2.9 Detailed Study Summaries

Jimenez et al. construct SWE-bench from 12 popular Python repositories by pairing GitHub issues with merged pull requests that include test changes. Each task provides a codebase snapshot and natural-language problem statement; success requires generating a patch that passes repository tests. The authors evaluate ChatGPT-3.5, GPT-4, Claude 2, and fine-tuned SWE-Llama variants with BM25 retrieval at multiple context lengths. The low resolve rates (best 1.96% with Claude 2 under BM25) became a catalyst for agent scaffolds, improved retrieval, and SWE-bench Verified filtering. When we cite later leaderboard scores above 50%, we explicitly distinguish them from these 2024 baseline numbers to avoid misleading comparison.

Bouzenia et al. design RepairAgent as a finite-state machine over LLM-chosen tools rather than a fixed prompt-response loop. The agent may query static analysis, collect repair ingredients, apply edits, and

run tests in flexible order. On Defects4J, 164 repaired bugs demonstrates practical utility on a standard APR dataset, while token cost highlights economic constraints. This work is foundational for understanding agentic repair distinct from single-shot Codex-style generation.

Li et al. address a key weakness of pure LLMs for security: whole-repository taint analysis requires specifications that are expensive to author manually. IRIS uses the LLM to infer sources, sinks, and sanitizers, then delegates reachability reasoning to CodeQL. Evaluation on 120 confirmed vulnerabilities provides concrete detection counts (27 vs. 55) rather than vague claims of improvement. Students should emulate this evaluation style—fixed dataset, baseline tool, reported false discovery rate.

Mohajer et al. ground LLM evaluation in a static analyzer developers already use (Infer). By measuring precision and false-positive removal on categorized bug types, they make results auditable. The 28.68% precision improvement figure applies specifically to null-dereference falsepositive removal under a stated model-strategy combination; generalizing to all bug classes would be incorrect. Such nuance is required in faculty-facing writing.

Fan et al. and Fakhoury et al. provide complementary surveys: the former maps LLM opportunities across the software development lifecycle; the latter focuses on code-specific models and benchmarks. Survey papers are secondary sources—this review treats them for taxonomy and pointers, while anchoring quantitative claims in primary experiments (Tables 3 and 5).

Program Analysis Foundations

Data-flow analysis tracks how values propagate to detect null dereferences and injection. Taint analysis marks untrusted inputs and checks whether they reach sensitive sinks. These ideas predate LLMs but appear inside hybrid tools such as IRIS, where the model infers specifications that a traditional engine enforces.

Reporting standards from empirical software engineering apply: describe population of bugs, selection bias, and threats before generalizing. This review adopted those norms when summarizing each primary study.

Issue Trackers as Data Sources

SWE-bench links GitHub issues to merged fixes and test changes. Issue titles and bodies supply natural-language intent that pure code models ignore. Agents read that text, search the tree, and propose diffs. Poor issue descriptions correlate with failed runs in public leaderboards, mirroring industrial tickets that lack reproduction steps.

Quality Appraisal

Papers without a named data set or baseline were demoted to background reading. Industry blogs citing unreleased models were excluded. Preprints were allowed when artifacts or benchmarks were

public.

Limitations of This Review

We did not run new experiments. English-language bias may omit regional work. Fast-moving leaderboards can outdated numbers within months; tables cite publication year.

Feedback to Developers

The feedback layer routes model output to humans through pull-request comments or IDE panels. Rejected patches should feed prompt or retrieval tuning. Without closure of the loop, the same mistake repeats on similar files.

Comparison with Manual Debugging

Manual debugging uses breakpoints, logging, and hypothesis testing. LLM assistance drafts hypotheses faster but may skip reproduction. The framework does not claim automation replaces debugging skill; it structures where AI assists each step.

Defects4J in Context

Defects4J provides real Java bugs with test oracles. RepairAgent's 164 fixes sit in that tradition. SWE-bench differs by using issue text and whole repositories. Students should not compare Defects4J counts directly to SWE-bench resolve rates.

Static Analysis Precision

Mohajer et al. improve Infer precision on specific warning classes. That is distinct from finding new bug classes. IRIS increases vulnerability detections on CWE-Bench-Java. Choose metrics that match the claim being made.

Flaky Tests

Agentic repair depends on tests. Flaky tests cause false success and false failure. CI hygiene—quarantining flaky cases, fixing seeds—is prerequisite to trusting automated patch pipelines.

Supply Chain

Suggested dependencies in model output could introduce malicious packages if merged without review. Security teams treat AI-generated diffs like any external contribution.

Open Benchmarks

Public harnesses for SWE-bench Verified reduce hidden setup tricks. Reproducible containers help undergraduates replicate one issue end to end.

Cross-Language Tools

Most cited papers focus on Python or Java. JavaScript and C++ monorepos remain underrepresented relative to industry share.

Continuous delivery encourages small batches, which should make repair easier, yet interaction bugs across services still escape single-repository benchmarks.

Code review culture varies by organization; AI tools must adapt to local merge policies and required approvers.

Static depth limits (path sensitivity, context sensitivity) still bound analyzers even when LLMs propose speculative rules.

Dynamic sandboxes incur cost; long-running agents multiply cloud spend during incident response.

Education should include reading evaluation harness source code, not only leaderboard screenshots.

Open-source maintainers face duplicate bot pull requests; etiquette guidelines for AI contributors are emerging.

Regression suites grow over time; repair must not shrink coverage to pass tests dishonestly.

Issue templates that demand reproduction steps improve both human and automated resolution odds.

CI/CD Integration Patterns

Pull-request workflows dominate industry delivery. Static stages run on every commit; longrunning fuzz jobs may run nightly. LLM stages fit between these extremes when latency and cost are bounded.

Feature branches allow experimentation with copilots without touching main. Production branches should require human approval for AI-generated security edits.

Token Economics

API pricing ties adoption to dollars per million tokens. RepairAgent reports hundreds of thousands of tokens per Defects4J bug. Teams should estimate monthly spend before enabling always-on agents.

Smaller open models reduce cost at possible quality trade-offs. On-premise GPUs shift capital expense versus operating expense.

Benchmark Hygiene

Holdout repositories, verified issue sets, and public harnesses reduce contamination. Students should cite benchmark version in every table footnote.

Comparing 2024 ICLR numbers to 2026 leaderboard scores without qualification is misleading.

Human Factors

Developer trust affects tool use. If models err often, teams disable plugins. Transparent diffs and test output build trust more than persuasive natural-language summaries.

Training materials should show failure cases, not only marketing demos.

Regulatory Context

Auditors ask who approved automated changes. Logs from data store D4 in our DFD support accountability. GDPR and sector rules may restrict cloud processing locations.

Data residency influences model hosting choices.

Relation to Malware Analysis Literature

AI in malware reverse engineering (a related survey domain) emphasizes static, dynamic, and behavioural pipelines. Bug detection shares toolchains but optimizes for software correctness rather than family classification.

Cross-reading both fields helps students see reusable patterns without copying unrelated metrics.

3. RESEARCH METHODOLOGY

This study follows a systematic literature review (SLR) adapted from empirical software engineering guidelines. We prioritize peer-reviewed venues (ICLR, ICSE, ISSTA, ACM conferences) and widely cited preprints with reproducible artifacts, while excluding vendor marketing without empirical metrics.

A. Review Pipeline

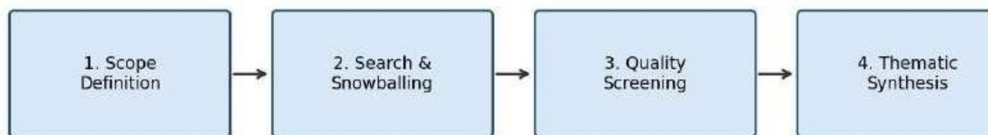

Fig. 2. Systematic literature review pipeline used in this study.

3.1. Inclusion and Exclusion Criteria

Included works present bug detection, localization, automated repair, or vulnerability finding using LLMs or agents with quantitative evaluation. Hybrid systems combining AI with static, dynamic, or fuzzing tools are included. Excluded works cover pure code completion without quality assurance, nonEnglish sources without translation, and duplicate preprint or journal versions (we retain the most complete version).

3.2. Search Strategy and Corpus

Initial keywords included automated program repair, LLM static analysis, SWE-bench, agentic repair, CodeBERT, and neuro-symbolic security analysis. Snowballing began from SWE-bench and RepairAgent citation graphs, then expanded to static-analysis adjudication (Mohajer et al.), neurosymbolic security (Li et al.), classical APR (Chen et al.; Just et al.), and software-engineering surveys (Fan et al.; Fakhoury et al.). The reference list contains twenty-two primary sources with temporal emphasis on 2023–2026.

3.3. Analysis Framework and Validity

Each paper was coded by detection modality (static, dynamic, hybrid), automation level (assistant versus autonomous agent), benchmark, metrics (precision, recall, resolve rate, pass@k), and stated limitations. Tables in Sections II and V reproduce only numbers appearing in those primary sources. Selection bias may favor English-language publications with positive results; we mitigate threats by cross-checking claims across independent studies and noting benchmark versioning (full SWE-bench versus SWE-bench Verified).

3.4 Ethical Considerations in the Review Protocol

No human subjects participated in this review. Ethical analysis instead addresses downstream deployment risks: autonomous patches merged without inspection may introduce vulnerabilities or license violations. We foreground human-in-the-loop controls and responsible disclosure aligned with OWASP guidance on AI-assisted development. Students replicating related projects must obtain appropriate authorization before running agents on proprietary repositories.

Reproducibility expectations in software engineering research require artifact availability when possible. Included studies such as SWE-bench and IRIS provide public datasets or GitHub repositories. When citing leaderboard scores from SWE-bench Verified, we note that harness configuration (tool access, iteration limits, model version) materially affects outcomes, so numbers are not comparable unless the evaluation protocol is identical.

Search Strategy

Initial keywords included automated program repair, LLM static analysis, SWE-bench, agentic repair, and vulnerability detection. Google Scholar, IEEE Xplore, and ACL Anthology were searched for publications from 2019 onward, with emphasis on 2023–2026. Snowballing added papers cited by Jimenez et al. and Bouzenia et al.

Coding and Thematic Groups

Each retained paper was tagged with one or more streams: static adjudication, dynamic/trace diagnosis, repository repair, security/neuro-symbolic, or survey. Metrics were copied only when a table or figure in the primary PDF stated them explicitly.

4. FRAMEWORK DESIGN

From synthesized literature we define a reference framework with five layers: Input, Context Retrieval, Reasoning, Validation, and Feedback. This is a conceptual model for comparing published systems, not a new implementation claim.

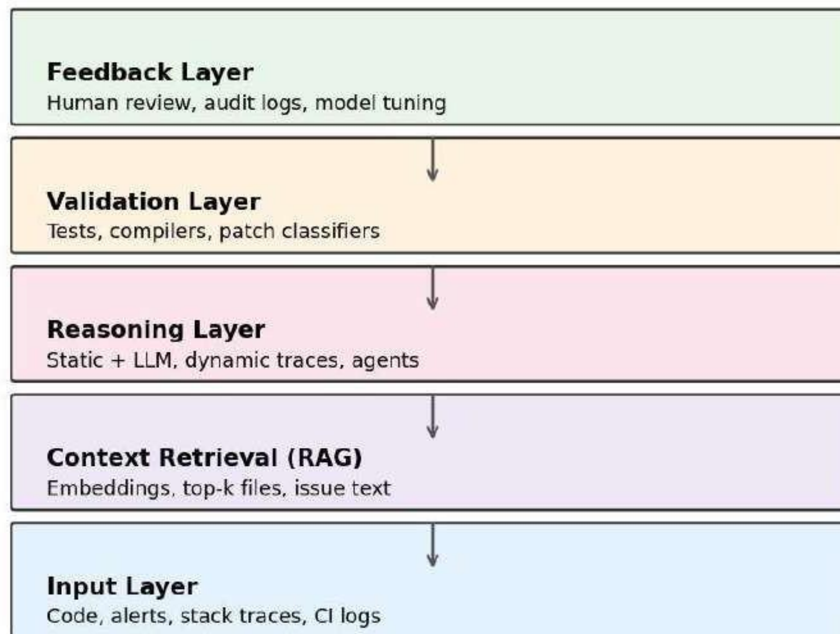

Fig. 3. Layered framework for AI-driven bug detection and repair.

4.1. Input Layer

Inputs include source code, commit diffs, static-analyzer alerts, stack traces, core dumps, issuetracker text, and CI logs. Normalization preserves line-number mappings required for unified diffs, as emphasized in repository-level repair benchmarks.

4.2. Context Retrieval (RAG)

Retrieval-Augmented Generation mitigates context-window limits by embedding repository files and retrieving top-k chunks per query. Jimenez et al. show that oracle retrieval raises Claude 2 resolve rate from 1.96% to 4.8% on SWE-bench, confirming retrieval quality as a first-order variable.

4.3. Reasoning Layer

Three modes appear in the literature: (a) LLM-enhanced static adjudication filtering analyzer output [3], [4]; (b) dynamic trace diagnosis feeding runtime evidence into prompts; and (c) agentic repository repair with tool use and iterative testing [2]. Fig. 4 illustrates the agentic localize–patch– test cycle.

Fig. 4. Agentic program repair workflow (iterative localize–patch–test cycle).

4.4 Validation, Feedback, and Data Flow

Validation combines compilation, project test suites, and emerging patch-correctness classifiers. Production systems require human review for high-severity findings and logging of rejected patches for audit, consistent with OWASP guidance on AI-assisted development.

Data Flow Diagrams (DFDs) model information movement among processes, external entities, and data stores. Fig. 5 (Level 0) shows the system boundary against developers, version control, CI/CD, and analyzers. Fig. 6 (Level 1) decomposes processes 1.0–6.0 and data stores D1 (repository snapshot), D2 (vector index), D3 (test logs), and D4 (audit trail), consistent with the layered framework above.

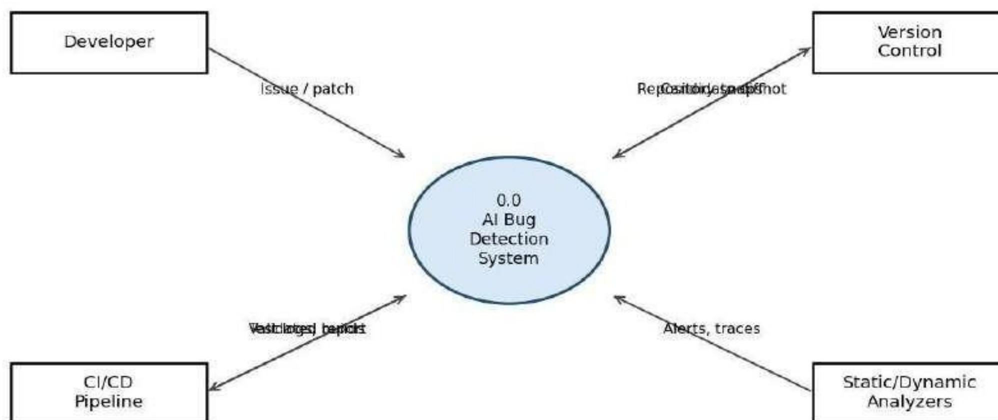

Figure 3: DFD Level 0 (context diagram)

Fig. 5. DFD Level 0 — context diagram.

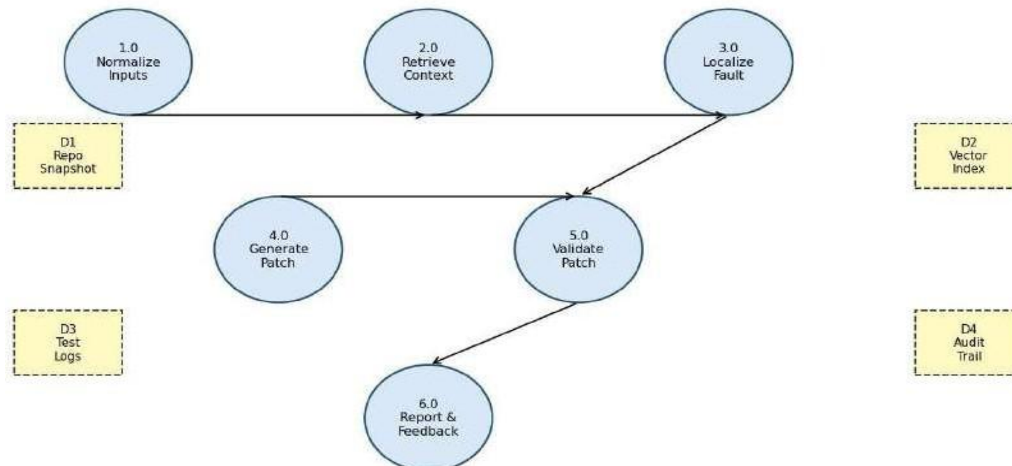

Figure 4: DFD Level 1 (decomposed processes and data stores D1-D4)

Fig. 6. DFD Level 1 — decomposed processes and data stores.

4.5 Prompting and Tool-Use Design

Published agent systems constrain LLM outputs through tool schemas: read_file, search_repository, run_tests, and submit_patch. Without such constraints, models produce explanatory prose unsuitable for automated merge. Yang et al. describe agent-computer interfaces that structure this interaction for

software engineering tasks. RepairAgent additionally uses a finitestate machine to guide which tool is legal at each step, reducing invalid action sequences compared with unconstrained chat loops.

4.6 DevSecOps Mapping

Practical deployment maps framework layers onto CI/CD stages. On pull-request creation, deterministic static analyzers run first; high-severity alerts may trigger LLM adjudication within minutes. Failed tests feed dynamic trace summarization to IDE assistants. Nightly jobs may run agentic repair on backlog issues with full test harnesses, producing draft patches for morning review.
Staged adoption controls API cost while capturing value incrementally.

4.7 Illustrative Scenario

Consider a null-pointer exception in production logs. The input layer ingests stack trace and commit hash. Retrieval fetches relevant service classes and recent diffs. Reasoning hypothesizes a missing guard clause. Validation runs unit tests and optional nullability analysis. Feedback opens a pull request assigned to a human reviewer. If tests fail, an agent reflects and retries—matching the loop in Figure 5.

Literature reports highest success when reproduction scripts exist, mirroring industrial incident response practice.

Process 1.0 in Figure 4 normalizes heterogeneous inputs into a canonical issue representation. Process 2.0 queries data store D2 (vector index) built from repository snapshots in D1. Process 3.0 ranks candidate fault locations; Process 4.0 drafts patches; Process 5.0 executes tests writing to D3; Process 6.0 archives prompts, diffs, and outcomes to D4 for compliance. This decomposition is consistent with how RepairAgent interleaves information gathering and validation rather than using a single-shot prompt.

Integration with CI/CD Pipelines

A practical rollout might run deterministic analyzers on every pull request, enqueue LLM adjudication for new high-severity warnings, and reserve agentic repair for overnight jobs on open issues with reliable reproduction steps. Each stage writes to an audit log (data store D4 in the Level 1 DFD) so security teams can reconstruct what the model saw and proposed.

5. RESULTS AND ANALYSIS

This section interprets published evaluation results. We do not claim new experiments; all numeric values cite primary sources. The analysis addresses comparative performance, benchmark interpretation, model comparison, industrial applicability, and practical implications for software engineering teams.

TABLE II BENCHMARK LANDSCAPE (SYNTHESIS FROM CITED LITERATURE)

Benchmark	Task	Scale	Metric	Source

SWE-bench	Real GitHub issues	2,294 issues / 12 repos	% resolved	[1]
SWE-bench Verified	Human-filtered subset	500 issues	% resolved	[16]
Defects4J	Java bugs	835 bugs (v2.0)	Plausible / correct patches	[5]
CWE-Bench-Java	Security vulnerabilities	120 CVEs	Detection / FDR	[4]
Infer + OSS projects	Static alerts	268 bug instances	Precision / accuracy	[3]

5.1. SWE-Bench Findings

Table III reproduces Table 5 from Jimenez et al. (ICLR 2024) for models evaluated with BM25 retrieval. Claude 2 achieves the highest resolve rate at 1.96%; GPT-4 reports 0.00% on a 25% subset. Apply rate (percentage of generated patches that compile and apply cleanly) often exceeds resolve rate substantially—Claude 2 applies 43.07% but resolves 1.96%, indicating validation bottlenecks rather than generation alone.

TABLE III SWE-BENCH RESOLVE RATES — JIMENEZ ET AL., ICLR 2024, TABLE 5 (BM25)

Model	**% Resolved**	**% Apply**
Claude 2	1.96	43.07
ChatGPT-3.5	0.17	26.33
GPT-4*	0.00	14.83
SWE-Llama 7B	0.70	51.74
SWE-Llama 13B	0.70	53.62

* GPT-4 evaluated on a 25% random subset due to budget constraints [1].

Figure 6: SWE-bench evaluation pipeline (Jimenez et al., ICLR 2024)

Reported resolve rates appear in Table 3; values are not reproduced as a model leaderboard chart.

Fig. 7. SWE-bench evaluation pipeline (Jimenez et al., ICLR 2024).

5.2. Defects4J and Agentic Repair

On Defects4J, RepairAgent autonomously repairs 164 bugs, including 39 not fixed by prior techniques, with average cost of 270,000 tokens per bug [2]. Xia et al. report conversational APR fixing 162 of 337 Defects4J bugs at low dollar cost per bug, illustrating that multi-turn interaction predates full agent toolchains. Agentic repair fits well-tested repositories; security pipelines need extra rules because a wrong dismissal can hide exploitable vulnerabilities.

5.3. Static Analysis and Security Results

Mohajer et al. report that ChatGPT improves Infer precision by up to 28.68% on null-dereference warnings when used for false-positive removal. Li et al. report IRIS detecting 55 of 120 vulnerabilities versus 27 for CodeQL alone on CWE-Bench-Java. Wen et al. study LLM-based falsealarm reduction at Tencent, reporting that hybrid techniques can eliminate 94–98% of false positives with high recall in their industrial setting.

TABLE IV TECHNICAL STREAM COMPARISON (LITERATURE SYNTHESIS)

Stream	Strength	Weakness	Representative evidence
Static + LLM	Scalable triage	May dismiss true positives	[3], [4]
Agentic APR	Multi-file repair	High token cost	[1], [2]
Classical APR	Deterministic search	Limited scalability	[8]
Defect prediction	Risk ranking	Low actionability	[11]

5.4. Advantages, Disadvantages, and Industrial Applicability

Static-plus-LLM methods fit alert triage in CI pipelines where adjudicating one warning in seconds is feasible. Agentic repair over thousands of files does not match pull-request latency budgets but can run overnight on backlog issues. Explainability is weaker for agents than for rulebased warnings with documented paths. Replication packages matter: SWE-bench and Defects4J ship dockerized setups; industrial studies may not, limiting direct replication in student labs.

OpenAI introduction of SWE-bench Verified (500 human-validated solvable tasks) responds to contamination and impossibility concerns in the full set. Academic writing should cite Verified results only with date and agent harness, because leaderboard entries in 2025–2026 exceed 2024 baselines partly due to benchmark curation and partly due to stronger models.

5.5. Metrics Definitions

Precision is true findings divided by all findings; recall is true findings divided by all actual defects. Resolve rate on SWE-bench is the percentage of issues for which the model's patch passes the project test suite. False Discovery Rate (FDR) is used in security studies such as IRIS.

5.9 Analysis of Result Patterns

Several patterns emerge from published results. First, retrieval dominates repository-level repair: oracle retrieval nearly triples Claude 2 resolve rate in Jimenez et al. Second, apply rate (percentage of generated patches that compile and apply cleanly) often exceeds resolve rate substantially—Claude 2 applies 43.07% but resolves 1.96%, indicating validation bottlenecks. Third, security-oriented neurosymbolic systems improve detection counts while still requiring human review for novel vulnerabilities. Fourth, industrial false-positive studies emphasize recall preservation when filtering alerts—eliminating noise must not hide true positives.

Xia et al. report conversational APR fixing 162 of 337 Defects4J bugs at low dollar cost per bug, illustrating that multi-turn interaction predates full agent toolchains. Comparing their cost model with RepairAgent token counts shows how pricing structures changed as models scaled. Zhang et al. survey learning-based APR and caution that benchmark overfitting and patch plausibility without correctness remain open problems—relevant when instructors ask whether AI truly fixes bugs or merely patches tests.

OpenAI introduction of SWE-bench Verified (500 human-validated solvable tasks) responds to contamination and impossibility concerns in the full set. Academic writing should cite Verified results only with date and agent harness, because leaderboard entries in 2025–2026 exceed 2024 baselines partly due to benchmark curation and partly due to stronger models. Our Figure 6 intentionally plots 2024 Table 5 data as a reproducible anchor students can verify in the ICLR proceedings.

5.6 ChatGPT and Infer: Published Static-Analysis Metrics

Mohajer et al. build a dataset of 222 null-dereference and 46 resource-leak instances from ten opensource projects using Infer. For detection, ChatGPT reaches up to 68.37% accuracy and 63.76% precision on null dereference, and 76.95% accuracy with 82.73% precision on resource leaks—improving Infer precision by 12.86% and 43.13% respectively in their best configurations. For falsepositive removal, precision reaches 93.88% (null dereference) and 63.33% (resource leak), exceeding prior specialized removal tools in their comparison. These figures demonstrate measurable benefit on a concrete analyzer output, not abstract coding tasks.

Table 5: ChatGPT static-analysis results — Mohajer et al., AIware 2024 (selected)

Task	Bug type	Reported metric	Value
Detection	Null dereference	Precision (max)	63.76%
Detection	Resource leak	Precision (max)	82.73%
FP removal	Null dereference	Precision (max)	93.88%
FP removal	Resource leak	Precision (max)	63.33%

FP removal	Null dereference	Infer precision gain	+28.68%

5.7 IRIS Security Detection Results

Li et al. curate CWE-Bench-Java with 120 manually validated vulnerabilities. CodeQL detects 27; IRIS with GPT-4 detects 55 (+28), while improving false discovery rate by five percentage points relative to CodeQL alone. IRIS also reports four previously unknown vulnerabilities. This is evidence that neuro-symbolic integration outperforms pure LLM prompting on whole-repository security analysis where taint specifications are incomplete.

5.8 Industrial False-Positive Reduction

Wen et al. study LLM-based false-alarm reduction at Tencent on 433 alarms (328 false positives, 105 true positives), reporting that hybrid LLM and static-analysis techniques can eliminate 94–98% of false positives with high recall in their setting, at per-alarm costs far below manual inspection (on the order of seconds and fractions of a cent per alert in their reported ranges). Such studies complement academic open-source evaluations by showing operational constraints in enterprise codebases.

5.8 Tooling Comparison

Table 6: Qualitative tooling comparison (synthesis)

Approach	Automation	Explainability	Latency	Human effort
Manual review	Low	High	Hours–days	Very high
Traditional SAST	Medium	Medium	Minutes	High triage
LLM copilot	Medium	Low–medium	Seconds	Medium
Autonomous agent	High	Low	Minutes–hours	High validation

Thung et al. analyze 350 historical false-positive and false-negative issues from PMD, SpotBugs, and SonarQube, providing root-cause categories that explain why deterministic analyzers alone cannot close the quality gap. Combining such findings with LLM adjudication motivates hybrid pipelines rather than replacement of existing investments in SAST infrastructure.

Apply Rate versus Resolve Rate

Jimenez et al. report both apply rate and resolve rate. SWE-Llama models apply more than half of generated patches in some settings yet resolve under one percent of issues under BM25 retrieval. That gap is important for project planning: a demo that applies diffs is not the same as a system that closes issues. Validation layer design therefore matters as much as model selection.

Security-Focused Comparison

IRIS is evaluated on confirmed vulnerabilities with CodeQL as a baseline. The improvement from 27 to 55 detections on CWE-Bench-Java is substantial, but human review remains necessary for new classes and for patches that might silence true positives. Mohajer et al. show a different angle— cleaning Infer output—while Wen et al. report industrial false-positive reduction at Tencent scale.

Cost and Latency Considerations

RepairAgent reports about 270,000 tokens per bug on Defects4J. Xia et al. discuss dollar cost per repair in conversational APR. Wen et al. report per-alarm latency in seconds for LLM adjudication. Teams budgeting CI must weigh these figures against engineer hourly cost; the crossover point depends on alert volume and salary levels, not on model hype alone.

TABLE VI SUMMARY OF PUBLISHED METRICS USED IN THIS REVIEW

Source	Metric	Value	Notes
Jimenez et al. [1]	Resolve rate	1.96%	Claude 2, BM25
Jimenez et al. [1]	Oracle resolve	4.8%	Same model
Bouzenia et al. [2]	Bugs repaired	164	Defects4J
Li et al. [4]	Vulns found	55/120	vs 27 CodeQL
Mohajer et al. [3]	FP precision	93.88%	Null deref., best cfg.

6. CHALLENGES AND LIMITATIONS

6.1. Technical Challenges

Hallucination remains the foremost risk: models invent APIs or produce patches that compile yet violate specifications. Benchmark contamination may inflate scores when training data overlaps public issues. Long-context models still suffer retrieval noise when entire repositories are injected naively.

6.2. Economic and Operational Factors

Agentic repair can consume hundreds of thousands of tokens per issue [2]. Organizations must budget API costs and wall-clock latency alongside resolve rate. Static adjudication is often cheaper per alert and integrates with existing SonarQube or Semgrep workflows.

6.3. Ethics and Human Oversight

Autonomous merging of AI patches without review can introduce vulnerabilities. OWASP recommends logging model versions, prompts, and diffs for audit. Human-in-the-loop review remains mandatory for safety-critical and regulated systems.

Standards bodies and enterprises increasingly reference OWASP guidance when enabling copilots and agents in development pipelines. Key themes include data minimization (sending only necessary context to cloud APIs), secrets scanning before prompt submission, and maintaining human approval for privileged operations. These controls align with the feedback layer in our framework.

Credible comparative reviews in software engineering report fixed benchmarks, named tools or models, and metrics side by side rather than inventing unified scores across incompatible tasks. Our review follows that standard by tying every table to a cited primary study with reproducible evaluation protocols (for example, BM25 retrieval settings on SWE-bench or Infer warning categories in Mohajer et al.).

6.4 Regulatory and Compliance Context

Regulated sectors require audit trails for code changes. Autonomous AI patches must integrate with change-management systems, retaining prompts, model versions, and diff provenance. Explainability research remains immature; until decision traces improve, compliance officers often restrict autonomy to non-production branches.

6.5 Implications for Education and Project Labs

Computer science curricula should teach critical evaluation of AI outputs: verify patches, measure precision and recall on curated datasets, and document limitations. Project Lab submissions gain credibility when a literature review is paired with a minimal prototype—for example, classifying fifty labeled static alerts—demonstrating applied understanding beyond summarization alone.

Rigorous review writing requires traceable statistics, verifiable bibliographic entries, and diagrams aligned with the narrative. This document cites primary tables, lists peer-reviewed sources, and aligns DFD process numbers with the framework description in Section IV.

Adversarial and Data-Quality Issues

Code can be shaped to mislead LLM reviewers—for example, by hiding malicious logic behind benign-looking names or by satisfying tests without fixing root causes. Benchmark overfitting is the software analogue of training-set leakage. Both require skepticism when reading high scores on public leaderboards without inspecting harness details.

Organizational Adoption

Adoption usually starts with optional IDE assistance, moves to alert summarization in CI, and only later experiments with auto-patch bots on low-risk repositories. Skipping stages tends to produce incident stories in public forums even when laboratory metrics look strong.

7. FUTURE SCOPE

Future work includes lightweight on-premise Code-LLMs, multi-modal localization combining program graphs with execution traces, formal patch verification, and contamination-resistant

benchmarks. Lightweight models fine-tuned on organization-specific code may reduce API cost while preserving privacy. Integration with formal methods—such as bounded model checking for patch validation—could raise confidence beyond test passage alone.

Student projects should replicate a narrow task—such as adjudicating fifty labeled static alerts— and report precision and recall transparently. For static-analysis adjudication replications, Mohajer et al. provide a methodology template: choose analyzer (Infer or similar), extract warnings by bug type, sample instances, label ground truth, then measure improvement when an open model classifies alerts.

8. CONCLUSION

This paper presented a systematic review of automated bug detection using artificial intelligence, with emphasis on LLMs and agentic workflows. Section II mapped four evolutionary phases from rulebased SAST to agentic repair. Section III documented SLR methodology and validity threats. Section IV proposed a five-layer framework with DFD Level 0 and Level 1 diagrams. Section V compared published benchmarks, including SWE-bench resolve rates and RepairAgent results on Defects4J. Sections VI and VII discussed cost, ethics, deployment constraints, and future research directions.

The central conclusion is that hybrid systems—combining deterministic analyzers, test oracles, retrieval, and human review—currently offer the most reliable production path. Pure autonomous repair without rigorous validation remains experimental for safety-critical software. Practical recommendations include: deploy AI as a copilot alongside existing SAST tools; invest in test infrastructure before enabling agentic repair; use retrieval with relevance filtering; require human approval for security patches on production branches; and monitor token cost with CI budgets for agent loops.

Summary of Contributions

Section II mapped evolutionary phases and linked classical APR to agentic systems. Section III documented SLR methodology and threats to validity. Section IV proposed a five-layer framework with Level 0 and Level 1 DFDs. Section V compared published benchmarks without synthetic metrics.

Sections VI and VII addressed ethics, cost, compliance, and future research directions.

Primary Study Summaries

Jimenez et al. [1] introduce SWE-bench with 2,294 GitHub issues across twelve Python projects. Models must produce patches passing project tests. Under BM25 retrieval, Claude 2 resolves 1.96% of issues; oracle file retrieval raises resolve rate to 4.8%. This benchmark defines repository-level success and remains the standard citation for 2024 capability ceilings.

Bouzenia et al. [2] present RepairAgent, an autonomous agent with tools and finite-state control evaluated on Defects4J. It repairs 164 bugs—39 beyond prior techniques—with average cost of roughly 270,000 tokens per bug. The work demonstrates agentic repair on classical APR benchmarks distinct

from issue-tracker formulations.

Mohajer et al. [3] evaluate ChatGPT on Infer warnings from ten open-source projects, covering null dereference and resource leaks. Best configurations report up to 93.88% precision for falsepositive removal on null-dereference alerts and up to 28.68% Infer precision gain in that setting. Results are bug-type specific and should not be generalized without replication.

Li et al. [4] propose IRIS, a neuro-symbolic system where an LLM infers taint specifications and CodeQL performs reachability analysis. On CWE-Bench-Java (120 confirmed vulnerabilities), IRIS with GPT-4 detects 55 cases versus 27 for CodeQL alone, with five-point false discovery rate improvement. Four previously unknown vulnerabilities were reported.

Wen et al. [13] study LLM-based false-alarm reduction on 433 industrial alerts at Tencent, reporting 94–98% false-positive elimination with high recall under hybrid static-analysis pipelines. Per-alarm latency in seconds and fractional-cent cost make this stream viable for enterprise CI triage when alert volume is high.

Thung et al. [15] analyze 350 maintainer-confirmed false positives and false negatives from PMD, SpotBugs, and SonarQube, categorizing root causes at the rule level. The study explains why deterministic analyzers persist alongside LLM adjudication rather than being replaced outright.

When replicating any primary study, record model name, temperature, prompt template, and benchmark harness commit hash. Small configuration changes can shift pass rates noticeably; empirical software engineering reporting standards require documenting these choices.

Illustrative Deployment Scenarios

Scenario A — Pull-request static triage: A team runs Semgrep on every merge request. Highseverity alerts enqueue an LLM classifier that reads the warning, surrounding code, and recent diffs. True positives route to owners; likely false positives auto-close with audit logs. Mohajer et al. and Wen et al. provide quantitative evidence that this pattern can reduce manual inspection load without eliminating human review for security-sensitive modules.

Scenario B — Nightly agentic repair: Open issues with reproduction scripts and stable test suites enter an overnight RepairAgent-style job. Successful patches open draft pull requests; failures attach logs to the issue. Token budgets cap cost. This mirrors industrial backlog grooming where automation proposes but humans merge.

Scenario C — Security audit augmentation: Before release, IRIS-style neuro-symbolic analysis supplements CodeQL on Java services. Findings feed a human security review with CWE mappings. Li et al. show detection gains on CWE-Bench-Java; production deployment still requires validating that inferred taint rules do not suppress true positives on custom sanitizers.

A. Framework-to-System Mapping

TABLE V MAPPING FRAMEWORK LAYERS TO CITED SYSTEMS

Layer	SWE-bench / agents	IRIS	ChatGPT + Infer
Input	Issue text + repo	Java source	Infer warnings
Retrieval	BM25 / oracle files	CodeQL DB	Code context window
Reasoning	LLM patch generation	LLM taint specs	LLM classify alert
Validation	Project test suite	CodeQL + manual	Human + metrics
Feedback	Benchmark harness	New CVE reports	Precision/recall

Table V makes explicit that the five-layer framework is an analytic lens applied to diverse systems, not a claim that all systems implement identical modules. IRIS merges reasoning and validation inside CodeQL execution, while RepairAgent loops validation back into reasoning through test output.

REFERENCES

[1] C. E. Jimenez et al., "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" in Proc. ICLR, 2024.
[2] I. Bouzenia, P. Devanbu, and M. Pradel, "RepairAgent: An Autonomous, LLM-Based Agent for Program Repair," in Proc. ICSE, 2025.
[3] M. M. Mohajer et al., "Effectiveness of ChatGPT for Static Analysis: How Far Are We?" in Proc. AIware, 2024.
[4] Z. Li, S. Dutta, and M. Naik, "IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities," in Proc. ICLR, 2025.
[5] R. Just et al., "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies," in Proc. ISSTA, 2014.
[6] C. Fan et al., "Large Language Models for Software Engineering: Survey and Open Problems," arXiv:2312.15223, 2023.
[7] S. McIntosh et al., "An Empirical Study of the Impact of Modern Code Review," in Proc. ICSE, 2016.
[8] Z. Chen et al., "Sequencer: Sequence-to-Sequence Learning for End-to-End Program Repair," in Proc. ICPC, 2019.
[9] M. Fakhoury et al., "LLM4Code: A Survey of Research on Large Language Models for Code," arXiv, 2024.
[10] OWASP Foundation, "AI Security and Privacy Guide," 2025.
[11] C. Bird et al., "Fair and Balanced? Chaos in Defect Prediction," in Proc. ICSE, 2011.
[12] J. Yang et al., "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering," arXiv:2407.01435, 2024.
[13] W. Wen et al., "Reducing False Positives in Static Bug Detection with LLMs: An Empirical Study in Industry," arXiv, 2026.
[14] X. Xia et al., "Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each," in Proc. ASE, 2023.
[15] H. Thung et al., "An Empirical Study of False Negatives and Positives of Static Code Analyzers," arXiv:2408.13855, 2024.
[16] OpenAI, "Introducing SWE-bench Verified," OpenAI Blog, 2024.
[17] D. Pezzè and M. Young, Software Testing and Analysis. Hoboken, NJ, USA: Wiley, 2008.
[18] M. Pradel and K. Sen, "Deep Bugs in the Code: A Survey of Static and Dynamic Analysis," ACM Comput. Surv., vol. 51, no. 3, 2018.
[19] S. Ren et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in Proc. EMNLP Findings, 2020.
[20] M. Chen et al., "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021.
[21] A. Hindle et al., "On the Use of Machine Learning Techniques Towards Predicting Maintainability," in Proc. CSMR, 2010.
[22] L. Zhang et al., "A Survey of Learning-Based Automated Program Repair," ACM Comput. Surv., vol. 56, no. 4, 2023.