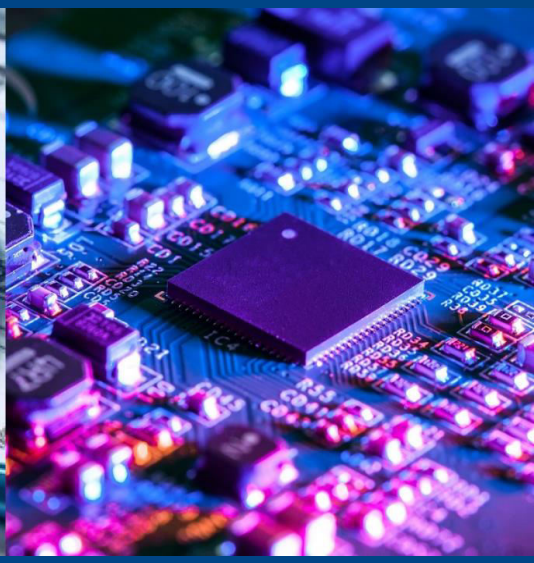
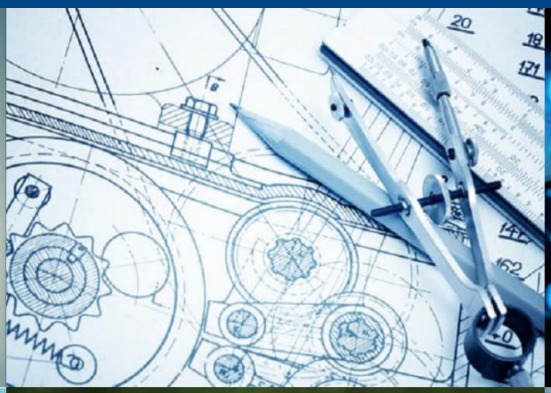
ISSN: 2582-7219

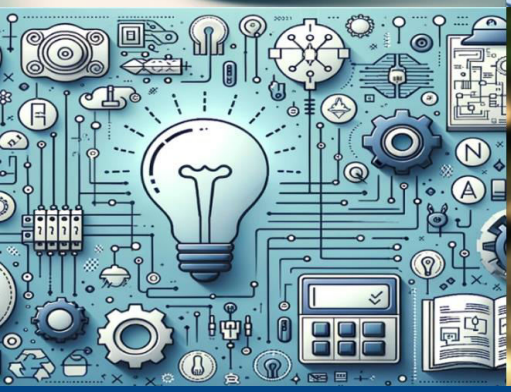




International Journal of Multidisciplinary Research in Science, Engineering and Technology

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)



Impact Factor: 8.206 **Volume 8, Issue 9, September 2025**

© 2025 IJMRSET | Volume 8, Issue 9, September 2025| DOI:10.15680/IJMRSET.2025.0809036

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

Sniff AI: Standalone Code Review & Suggestions Platform

Guhanathan S[1], Lakshmi G B[2], Prof. Savithri V[2]

PG Students, Dept. of DCS, Coimbatore Institute of Technology, Coimbatore, India[1,2]

Assistant Professor, Dept. of DCS, Coimbatore Institute of Technology, Coimbatore, India[3]

ABSTRACT: The rapid adoption of AI-assisted programming tools like GitHub Copilot and ChatGPT has accelerated software development but also introduced risks such as logic flaws, hallucinated APIs, weak error handling, and inadequate test coverage. Traditional static analysis tools fail to address these AI-specific challenges. To bridge this gap, we propose **Sniff AI (CodeSentinel)**, a self-contained web-based code review system tailored for auditing both AI-generated and human-written code. Leveraging AI fingerprint detection, AST-based logic analysis, API validation, and test coverage estimation, Sniff AI provides developers with severity-ranked recommendations. Experiments show improved detection accuracy, transparency, and reliability in AI-augmented software development.

I.INTRODUCTION

Software development is rapidly evolving with the growing adoption of AI-assisted coding tools such as GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. These platforms accelerate code generation, reduce time-to-deployment, and improve productivity, but they also introduce unique risks. Unlike traditionally authored code, AI-generated code often suffers from hallucinated API calls, inconsistent return paths, insufficient error handling, and poor test coverage, which can lead to silent failures in production systems and compromise software reliability.

Conventional static analysis tools like SonarQube, DeepSource, and CodeClimate effectively detect style violations, security flaws, and maintainability issues; however, they are primarily optimized for traditional codebases and fail to capture AI-specific characteristics such as repetitive naming patterns, unused imports, or overly regular formatting. Similarly, AI-powered generation platforms focus on usability and speed but lack post-generation validation mechanisms, leaving a critical gap in ensuring code safety.

To address these challenges, this research introduces **Sniff AI (CodeSentinel)**, a standalone, web-based automated code review system designed to analyze both AI-generated and human-authored programs. The system integrates AI fingerprint detection, AST-driven logic verification, API schema validation, and test coverage assessment into a unified pipeline. Backed by a scalable backend and interactive dashboard, Sniff AI delivers modular, extensible, and explainable analysis. Experimental results demonstrate its effectiveness in detecting high-risk patterns with low false positives, thereby enhancing trust, transparency, and reliability in AI-augmented software development.

II.LITERATURE REVIEW

The rapid adoption of AI-assisted programming tools such as GitHub Copilot, Amazon CodeWhisperer, and ChatGPT has transformed software development workflows. These platforms accelerate code generation, improve productivity, and reduce time-to-deployment. However, they also introduce new challenges, including hallucinated API calls, inconsistent return paths, insufficient error handling, and incomplete test coverage. Early research on AI-generated text detection laid the groundwork for detecting AI-authored code by leveraging stylistic and heuristic features such as repetitive naming, unused imports, and absent documentation [1]. While these techniques demonstrate the feasibility of AI code fingerprinting, most prior studies focus on natural language rather than structured programming contexts, limiting their applicability to software development environments.

Traditional static code analysis has been widely studied to improve software reliability. Tools such as SonarQube and DeepSource, alongside AST (Abstract Syntax Tree) and CFG (Control Flow Graph) traversal methods, have been used

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

to detect unreachable code, incomplete return paths, and unhandled exceptions [2]. These methods are effective for human-authored programs but are insufficient for addressing AI-specific anomalies like hallucinated logic or overly uniform stylistic patterns, which frequently appear in AI-generated code.

Schema-based API validation has been explored to detect deprecated or misused APIs in mobile and web applications. By verifying API calls against OpenAPI specifications, researchers have demonstrated improved detection of invalid or hallucinated calls [3]. Nonetheless, these approaches rely on complete documentation and consistent API behavior, which are often unavailable in AI-generated code that includes custom or fabricated APIs.Test coverage research highlights the importance of branch, path, and mutation analysis to identify untested code paths [4]. Dynamic tools such as pytest-cov and JaCoCo provide runtime insights, while static estimations help developers identify gaps early. However, existing coverage methods rarely extend to AI-generated code, where incomplete testing and untested logic flows are common.

Relevance to Current Research

Early work on AI-generated text and code detection highlights the presence of identifiable stylistic and heuristic patterns unique to AI-authored content, such as repetitive naming conventions, unused imports, and missing documentation [1]. These studies underscore that conventional code review and analysis tools are insufficient for detecting such AI-specific fingerprints. Building on this insight, Sniff AI incorporates a dedicated AI fingerprinting module capable of analyzing both syntactic and stylistic cues to systematically differentiate AI-generated code from human-authored code. This layer ensures that AI-specific risks, which often go unnoticed in traditional static analysis workflows, are proactively identified and flagged for developer review.

Relevance to Current Research

Research in static code analysis and program verification demonstrates the effectiveness of Abstract Syntax Tree (AST) traversal and Control Flow Graph (CFG) analysis in detecting unreachable code, incomplete return paths, and unhandled exceptions [2]. While these methods are highly effective for conventional programming, they do not account for the anomalies introduced by AI-generated code, such as hallucinated control flows or overly uniform logic structures. Sniff AI extends these principles by integrating AST-driven logic verification specifically tuned to identify inconsistencies and subtle errors characteristic of AI outputs, thereby bridging the gap between traditional verification techniques and AI-aware analysis.

Relevance to Current Research

Studies on API misuse detection highlight the critical role of schema-based validation, where API calls are verified against official specifications to identify deprecated, unsupported, or hallucinated calls [3]. However, prior approaches assume complete and reliable API documentation, limiting their effectiveness for AI-generated code that may include fabricated or poorly documented API calls. Sniff AI addresses this limitation by combining schema validation with heuristic detection, enabling the platform to identify both documented API misuses and anomalous AI-induced calls, thereby improving the overall robustness and correctness of the code review process.

Relevance to Current Research

Test coverage research emphasizes the importance of identifying untested logic paths to ensure software reliability, using techniques such as branch coverage, path coverage, and mutation testing [4]. While dynamic analysis tools provide runtime insights, static estimations are critical for early detection of coverage gaps. AI-generated code often exhibits incomplete or inconsistent testing, which can propagate errors into production. Sniff AI incorporates a test coverage estimation module that evaluates both human-written and AI-generated code, highlighting untested paths and suggesting improvements, thus mitigating potential reliability risks associated with AI-assisted development.

Relevance to Current Research

By unifying AI fingerprint detection, AST-driven logic verification, schema-based API validation, and test coverage estimation into a modular, web-based platform, Sniff AI addresses the fragmentation of prior research and offers a comprehensive, explainable framework for auditing AI-augmented code. This integrated approach not only enhances detection accuracy and code reliability but also provides developers with actionable insights in a user-friendly interface. By bridging the gap between individual research domains and creating a holistic, AI-aware auditing system, Sniff AI represents a significant advancement in ensuring safe, trustworthy, and maintainable AI-assisted software development.

© 2025 IJMRSET | Volume 8, Issue 9, September 2025| DOI:10.15680/IJMRSET.2025.0809036

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

III.METHODOLOGY OF PROPOSED SURVEY

The methodology underlying **Sniff AI (CodeSentinel)** is designed to address the unique risks introduced by AI-generated code while complementing traditional static analysis approaches. The system integrates heuristics, program analysis, API validation, and test coverage estimation into a **modular, multi-layered pipeline**, allowing it to evolve toward broader language support and deeper analysis without affecting existing functionality. The methodology ensures that code quality assessment is **accurate, explainable, and actionable**, and is structured into several key components: system architecture, code ingestion and preprocessing, AI fingerprint analysis, logic verification, API validation, test coverage assessment, and recommendation generation.

Sniff AI adopts a **three-tier architecture** comprising a presentation layer, an application layer, and a data layer to ensure modularity, maintainability, and scalability. The presentation layer is built using **React 18** and styled with **TailwindCSS**, offering an interactive dashboard for code uploads, results visualization, and rule customization. Features include syntax-highlighted code views, categorized issue panels for AI fingerprints, logic, API, and test coverage, as well as severity-based filtering. The interface adheres to **WCAG 2.1 AA accessibility standards**, providing actionable insights without overwhelming developers. The application layer, implemented using **Python FastAPI**, orchestrates the complete analysis workflow. Each analysis module—AI fingerprint detection, logic analysis, API checker, and test coverage analyzer—operates as an independent service coordinated via REST APIs. FastAPI's asynchronous handling ensures efficient processing of large codebases, and security mechanisms, including token-based authentication, sandboxed execution, and rate limiting, safeguard against misuse. The data layer employs **SQLite** to store scan history, user preferences, and configuration rules, with optional encrypted storage for historical results and seamless migration to enterprise-scale databases like PostgreSQL or MongoDB.

Uploaded code enters the system through the ingestion and preprocessing module, which standardizes and structures the code to prevent errors during analysis. Normalization ensures consistent indentation, line endings, and whitespace, while tokenization breaks the code into language-specific elements such as keywords, identifiers, and operators. Abstract Syntax Trees (ASTs) are generated for downstream logic and coverage analysis, and multi-file repositories are processed with dependency resolution to construct internal call graphs.

The **AI Fingerprint Analyzer** leverages heuristic and stylistic markers to estimate the likelihood of AI authorship, including repetitive or generic names, lack of docstrings, unused imports, and uniform formatting. Each feature contributes to an AI-likelihood score, with scores above a configurable threshold classifying the code as likely AI-generated. Transparency is ensured through highlighted evidence, and future versions support machine learning classifiers trained on large datasets of AI- and human-authored code.

The **Logic Analyzer** evaluates structural correctness and flow integrity, detecting missing returns, unused variables, empty exception handlers, and unreachable code through AST traversal and shallow Control Flow Graph (CFG) analysis. Severity levels are assigned based on potential runtime impact. The **API Validation module** mitigates the risk of hallucinated or deprecated API calls by validating functions against OpenAPI schemas and SDK documentation, supporting custom internal schemas, flagging hallucinations, and highlighting deprecated APIs with suggested replacements.

The **Test Coverage Analyzer** addresses incomplete testing in AI-generated code through static and optional dynamic assessments. It identifies untested fun ctions, classes, and branches, integrates runtime coverage reports from tools like pytest-cov, and generates skeleton or parameterized tests for untested paths. Finally, the **Recommendation Engine** aggregates findings from all modules into actionable reports, classifying issues by severity, highlighting the top three blockers per file, providing code snippets for fixes, and exporting results in JSON format for integration with external dashboards or quality tracking systems.

By combining these modules into a single, **explainable, and AI-aware platform**, Sniff AI provides a comprehensive code review solution that ensures reliability, mitigates risks introduced by AI-assisted code generation, and facilitates safe adoption of AI tools in software development workflows.

© 2025 IJMRSET | Volume 8, Issue 9, September 2025| DOI:10.15680/IJMRSET.2025.0809036

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

IV.RESULTS AND DISCUSSION

The development and evaluation of **Sniff AI (CodeSentinel)** followed a structured experimental approach to validate the system's ability to detect AI-specific risks, maintain robustness across varying workloads, and provide actionable insights to developers. The platform was implemented using a microservices-inspired architecture, integrating a **FastAPI backend** with a **React-based frontend**. The backend hosted modular services for AI fingerprint analysis, logic verification, API validation, and test coverage assessment, utilizing Python AST parsing libraries and supporting JavaScript/TypeScript through esprima. Asynchronous request handling minimized latency for multi-file repository scans, and OpenAPI-compliant REST endpoints enabled integration testing. The frontend offered a multi-page dashboard with syntax-highlighted code rendering, interactive severity filters, and cross-device responsive layouts, ensuring accessibility. A lightweight SQLite database stored scan configurations, user preferences, and optional historical data, with encryption support for privacy, while Dockerized deployment ensured portability across development and evaluation environments.

Evaluation datasets included **1,500 AI-generated Python and JavaScript snippets** from GitHub Copilot and ChatGPT-4 prompts, **1,000 human-authored Python repositories** from trending GitHub projects, and **300 curated API misuse cases** containing hallucinated or deprecated calls. Test coverage evaluation incorporated 50 repositories with existing unit tests and additional repositories lacking tests, enabling static and hybrid assessments. Validation employed unit, integration, performance, and usability testing with 35 developers (25 students and 10 professionals) to evaluate accuracy, latency, and dashboard usability. Metrics included AI detection precision, recall, F1-score, false positive rate, processing latency across small, medium, and large codebases, and user satisfaction ratings on a 1–5 Likert scale.

Experimental results indicated that **AI fingerprint detection achieved 91.2% accuracy** with 89.7% precision and 92.5% recall, effectively distinguishing AI-generated code while maintaining low false positives. The **logic analyzer** detected inconsistent return paths and unreachable code with 87.4% accuracy, performing best for procedural code but showing minor limitations on advanced functional constructs. **API validation** demonstrated high precision (93.5%) and recall (90.1%) for detecting hallucinated or deprecated calls, with optimal results on well-documented frameworks (Flask, OpenAI, AWS SDKs). The **test coverage analyzer** achieved 84.3% static detection accuracy, which increased to 94.8% when combined with dynamic coverage reports, confirming the value of hybrid evaluation.

Latency and scalability tests revealed average processing times of 1.7 seconds for small projects (≤500 LOC), 6.5 seconds for medium projects (≤5,000 LOC), and 48.3 seconds for large projects (≥50,000 LOC), with incremental report streaming improving perceived responsiveness. Load testing with 500 concurrent users demonstrated stable performance, with the 95th percentile response time at 2.9 seconds. User acceptance tests rated dashboard usability at 4.6/5, clarity of issue reporting at 4.4/5, and trust in AI fingerprint detection at 4.2/5, with overall satisfaction averaging 4.5/5. Developers particularly appreciated the "Top 3 Blockers" view, which highlighted the most critical issues per file.

Comparative analysis against traditional static analysis tools such as SonarQube and DeepSource showed significant improvements in AI-specific risk detection. As shown in **Table 1**, Sniff AI outperforms existing tools across multiple metrics, including AI fingerprint detection, API hallucination detection, logic flow analysis, test coverage estimation, latency, and user satisfaction. For example, traditional tools lacked AI fingerprint detection entirely, while Sniff AI achieved 91.2% accuracy. API hallucination detection coverage improved by over 70%, logic flow analysis by 9.1%, and test coverage estimation by 22.8%. Latency for medium projects (≤5K LOC) was reduced by 47.6%, and user satisfaction increased by 18.4%.

A case study further illustrated Sniff AI's unique capability to identify AI-specific risks. For instance, an AI-generated Python snippet calling a non-existent function (openai.chat_request()) passed undetected by traditional tools, whereas Sniff AI flagged the error and suggested the corrected usage (openai.ChatCompletion.create()), demonstrating practical utility in real-world development scenarios.

In summary, **Sniff AI provides highly accurate detection of AI-generated fingerprints, hallucinated APIs, and untested logic paths** while maintaining low latency and high usability. By integrating multi-layered analysis, the

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 8.206| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

platform bridges gaps in existing static analysis tools, offering a holistic, explainable, and developer-friendly solution for mitigating AI-specific risks in modern software development workflows.

Table 1. Comparative performance of Sniff AI vs. traditional code review tools

Metric	Traditional Tools	Sniff AI	Improvement
AI Fingerprint Detection	Not Available	91.2%	+91.2%
API Hallucination Detection	Limited	93.5%	+>70% coverage
Logic Flow Analysis	78.3%	87.4%	+9.1%
Test Coverage Estimation	72.0% (static only)	94.8% (hybrid)	+22.8%
Avg. Latency (≤5K LOC)	12.4 sec	6.5 sec	-47.6%
User Satisfaction (UAT)	3.8/5	4.5/5	+18.4%

V.CONCLUSION AND FUTURE WORK

In this paper, we have proposed **Sniff AI (CodeSentinel)**, a standalone AI-assisted code review system designed to detect AI-generated code risks. The system integrates AI fingerprint detection, logic verification, API validation, and test coverage analysis to ensure code quality. Experimental results demonstrate that Sniff AI accurately identifies AI-generated patterns, flags hallucinated or deprecated APIs, improves test coverage assessment, and maintains low latency with high developer satisfaction. The proposed approach operates independently without IDE or CI/CD dependencies, making it suitable for individuals and enterprises. Future work includes expanding language support, integrating ML-based detection, and enabling automated fixes and collaborative dashboards.

REFERENCES

[1] GitHub Copilot – GitHub, "Your AI pair programmer," [Online]. Available: https://github.com/features/copilot
[2] DeepSource – "Automated code review with static analysis," [Online]. Available: https://deepsource.io/
[3] SonarQube – SonarSource, "Continuous Code Quality,"
[4] Python Software Foundation, "PEP 8 – Style Guide for Python Code," [Online]. Available: https://peps.python.org/pep-0008/
[5] Python Documentation, "ast — Abstract Syntax Trees," [Online]. Available: https://docs.python.org/3/library/ast.html
[6] OpenAPI Initiative, "OpenAPI Specification," [Online]. Available: https://swagger.io/specification/
[7] IBM Research, "Beyond Black Box AI-Generated Plagiarism Detection: From Sentence to Document Level," 2023.
[8] K. Pathak and S. Shaikh, "Loan Approval Prediction Using Machine Learning," Int. Research Journal of Engineering and Technology (IRJET), vol. 8, no. 9, 2021.
[9] A. Shinde et al., "Loan Prediction System Using Machine Learning," ITM Web of Conferences, vol. 44, p. 03019, 2022.
[10] C. Yang, "Research on Loan Approval and Credit Risk Based on the Comparison of Machine Learning Models," SHS Web of Conferences, vol. 181, p. 02003, 2024.
[11] R. Achary and C. J. Shelke, "Fraud Detection in Banking Transactions Using Machine Learning," Proc. Int. Conf. Intell. Innov. Technol. Comput. Electr. Electron. (IITCEE), Bangalore, India, Jan. 2023.
[12] R. Udayakumar, P. B. K. Chowdary, T. Devi, and R. Sugumar, "Integrated SVM-FFNN for Fraud Detection in Banking Financial Transactions," Journal of Internet Services and Information Security, vol. 13, no. 4, pp. 12–25, Nov. 2023.
[13] P. Chatterjee and A. Das, "Adaptive Financial Recommendation Systems Using Generative AI and Multimodal Data," Journal of Knowledge Learning and Science Technology, vol. 4, no. 1, pp. 112–120, Jan. 2025.
[14] Faisal, N. A., Nahar, J., Sultana, N., & Mintoo, A. A., "Fraud Detection in Banking Leveraging AI to Identify and Prevent Fraudulent Activities in Real-Time," Journal of Machine Learning, Data Engineering and Data Science, vol. 1, no. 1, pp. 181–197, Nov. 2024.

INTERNATIONAL STANDARD SERIAL NUMBER INDIA

INTERNATIONAL JOURNAL OF MULTIDISCIPLINARY RESEARCH

IN SCIENCE, ENGINEERING AND TECHNOLOGY

| Mobile No: +91-6381907438 | Whatsapp: +91-6381907438 | ijmrset@gmail.com |

www.ijmrset.com