 ISSN:2302-8149
e-ISSN:2540-9719

AI-Based Detection of Hallucinated and Vulnerable Code Generated by LLMs

[1]Shayma Mustafa Mohi-Aldeen*
[1] College of Computer Science And Mathematics, University of Mosul, Mosul-Iraq
*e-mail: shaymamustafa@uomosul.edu.iq

(***received*:** 23 October 2026, ***revised*:** 11 November 2026, ***accepted*:** 17 November 2026)

Abstract

The adoption of Large Language Models (LLMs) for code generation has grown, but there are two risks associated with the outputs of these models: code hallucination, which occurs when a model generates code for packages, modules, methods, or attributes that do not exist, and insecure code generation, which happens when a model produces insecure programming practices. This study introduces a lightweight, dependency-free desktop application named LLM Code Auditor to detect hallucinated software references and security vulnerabilities within the generated Python code by LLMs. It takes into account code written with LLM-based coding assistants, such as ChatGPT, Claude, and GitHub Copilot. A set of five representative Python code snippets were gathered for authentication, data retrieval operations, loading of configuration files, backup operations, and utility functions. The prompts were crafted to match realistic programming tasks and include references to API usage, dependency, and security-sensitive operations related to these application scenarios. A set of collected samples has been assessed with two complementary validation mechanisms. A first application of runtime introspection was used to check whether modules are imported or attributes are referenced in the target python environment. Second, a source code parser to find insecure constructs was used to create an AST (Abstract Syntax Tree) which was then passed to a rule analyzer that mapped the found vulnerabilities to Common Weakness Enumeration (CWE) identifiers and the severity level. The evaluation used quantitative measures such as the number and distribution of the vulnerabilities detected, the hallucination findings, CWE classification, level of confidence and a composite risk score, ranging from 0 to 100. The experimental findings were shown to be able to detect 14 vulnerability findings in 5 categories of CWE and 2 hallucination findings: One Nonexistent Package and One Unsupported Method. The mean risk score for the five samples was 54.4. The results show that runtime verification and explainable static analysis are a useful tool to investigate the reliability and security issues within LLM-generated Python code prior to deployment, and that they are complementary.

Keywords: AI code generation, code hallucination, large language models, secure coding, vulnerability detection

1 Introduction

Today, large language models (LLMs), such as GPT-4, Claude, and Code Llama have quickly become a part of the software development landscape. The coding assistant based on AI is now being used to create functions, translate programs from one programming language to another, detect and fix software bugs, generate test cases, and even build entire software systems from natural-language specifications. These systems can significantly decrease programming effort and increase the speed of software functionality implementation by automating repetitive programming tasks and allowing for quick access to code suggestions [1]. But the growing number of automatically generated codes has also created new challenges in the area of correctness, reliability, and security. Whereas traditional tools for software development involve explicit compilation, testing, documentation, and verification steps, LLMs produce code based on statistical plausibility of token sequences they have seen in massive amounts of text and source code. Thus, the resulting code can look perfect in terms of syntax and be well written and still have no software artifacts, use the wrong APIs, have logic bugs, or security flaws. There are therefore two significant and related risks: code hallucination and insecure code generation [2].

 ISSN:2302-8149
e-ISSN:2540-9719

A code hallucination occurs when an LLM produces code or software-related content that doesn't align with the actual software context. A model can actually reference packages, modules, classes, functions, methods, and attributes that do not exist in the model. The names produced might seem very reasonable, as the names of legitimate software libraries are commonly used. One particular problem with dependency hallucination is that developers might try to install a package that the AI coding assistant told them to install, but which does not exist [3]. This practice has led to the development of what has been dubbed package hallucination or “slop squatting” risks, where an attacker creates a malicious package under a name that an LLM is likely to mispronounce. Therefore, relying on AI-generated dependency without further verification may lead to the introduction of unwanted or malicious code into a software development project. In addition to package names, some API calls may be hallucinated that result in runtime or compilation errors if the API doesn't exist for the desired library [4].

The other big worry is the creation of poor-quality source code. LLMs can be trained on programming patterns from massive and diverse sets of openly available code that can include outdated, insecure or-contextualized code that should not be learned. Consequently, it is possible for programs to be created that contain known vulnerabilities despite the legal and valid functionality. SQL statement construction using unsensitized string concatenation, using untrusted input passed through functions like (eval), insecure serialization methods, weak cryptographic algorithms for security critical operations, in-source code credential inclusion, and TLS certificate verification disabling [5]. These vulnerabilities can enable an attacker to mount an attack on the application, including any of the following: injection, unauthorized access, disclosure of data, or arbitrary code execution. The issue is exacerbated by the fluency and confidence in LLM responses, which can cause insecure implementations to appear to be secure if a security analysis is not done by the developer [6].

While hallucination and vulnerability are failure modes, they do have one common trait: generated code is not presumed correct just because it is syntactically correct, or because it may seem plausible. However, with hallucinated dependencies and APIs, verification is needed to ensure they are available in the real software environment, and with security issues, there is a need for systematic analysis of the source code created and the potentially unsafe operations it contains. Hence, auditing of AI generated code should take into account both the factual correctness of software references and the security properties of the code generated by the AI [7]. Many software quality and security tools are available to assist in detecting common software patterns and known software flaws. But, general-purpose static analyzers are not specifically geared toward detecting whether a dependency, module, function, method, or attribute actually exists on the target Python environment. Likewise, it is difficult to manually read all the AI-generated code snippets if the developers are increasingly leveraging generative AI all across the software-development cycle. This demands for lightweight and transparent auditing mechanisms that can verify AI-generated code before being used in a software project [8].

In this work, we tackle this gap by developing a lightweight auditing system for identifying two orthogonal problems of AI-generated Python code: software hallucination and security breach. The framework is implemented with an environment-oriented module verification and referenced attribute verification system, and an abstract syntax tree (AST) based vulnerability analyzer. Rather than using only a set of package names from a database, the proposed method checks for dependencies by attempting to resolve the modules referenced in the package, and checking for objects in the modules by using Python introspection features. This allows the system to make inferences about nonexistent or unrepentant software references, directly from the execution environment.

Explicit, rule-based patterns are used to analyses security over the Python AST in isolation. The analyzer analyzes source code structures that are related to operations that may be problematic, including dynamic code execution, construction of unsafe SQL, insecure deserialization, weak cryptographic usage, hardcoded sensitive information, and disabled security verification. The detected weaknesses are related to related CWE categories, if a suitable mapping exists. This rule-based design gives evidence for each warning and enables the developer to check the source-code construct that caused the warning, instead of checking an opaque machine-learning prediction.

The framework is meant to be a practical first-line auditing tool for AI-driven Python coding. By focusing on direct dependency verification, program-structure analysis, security rules that are easy to understand to security and human-readable explanations, it allows developers to verify generated

 ISSN:2302-8149
e-ISSN:2540-9719

code without needing deep security skills or large-scale machine learning systems. The study will integrate vulnerability analysis with hallucination detection within the same workflow to offer a more holistic view of the reliability and security of the LLM-generated Python code, before it is executed, integrated, or deployed.

1.1 Research Gap

There has been some work on security vulnerabilities and hallucinations in LLM-generated code recently, but there are still a number of limitations. In the first place, the study of security vulnerabilities and/or package/package API hallucination is usually conducted separately. For this reason, the tools and/or procedures used to determine the correctness of software references and insecure programming practices may differ. Second, previous research has generally concentrated on testing the security and quality of the code generated by the LLM, rather than creating a lightweight and practice audit tool for developers to audit the generated code before deployment. Third, most approaches for package-hallucination detection focus on package names and repository metadata, while other kinds of software references such as imported modules, methods and attributes are underrepresented. Lastly, some machine-learning techniques for vulnerability detection need labeled data, and training the model requires more computational power, and they may be unsuitable for a light-weight local code audit. To overcome these drawbacks, this study suggests an integrated approach to Python code auditing, which is a runtime introspection and Abstract Syntax Tree (AST) based security analysis. The approach not only checks for the availability of software components referenced in the target Python environment, but also analyzes the source code structure for known vulnerable programming patterns in an independent manner. The integration offers the developers a single and comprehensible way to detect hallucinated software references and potential security attacks.

1.2 Problem Formulation

The main challenge that is tackled in this research is the need for a lightweight and integrated approach toward identifying reliability and security risks in LLM-generated Python code. Reference to non-existent packages, modules, methods, attributes, and programming constructs that create security vulnerabilities are possible in generated programs. These issues can be missed during first code review and cause a failure during execution, or security issues if integrated in software projects.
The problem of the research is formulated as follows:
What is the easiest way to audit automatically LLM generated Python source code for hallucinated software references and security vulnerabilities with a lightweight, explainable and locally executable auditing mechanism?

The proposed solution is based on the approach where the input source code is an object of two complementary analyses:

1. Verify if modules and attributes are imported or referenced and can be resolved during runtime in the target Python environment.
2. Security vulnerability detection: Analyze the source code structure with AST-based rules to detect the pre-defined insecure programming patterns and link the detected patterns with the relevant Common Weakness Enumeration (CWE) categories and severity.

The result of both analysis paths is then merged to produce an overall risk assessment and report on the issues found, the parts of the source code where issues were found, explanations of the issues and the severity of the issues.

1.3 Research Questions

To investigate the identified problem, this study addresses the following research questions:

- **RQ1:** How can runtime introspection be used to identify hallucinated packages, modules, methods, and attributes in LLM-generated Python code?
- **RQ2:** How effectively can AST-based rule analysis identify predefined security vulnerabilities in LLM-generated Python programs and associate them with relevant CWE categories?
- **RQ3:** How can hallucination findings and security vulnerability findings be integrated into a unified risk-assessment mechanism?

 ISSN:2302-8149
e-ISSN:2540-9719

- **RQ4:** To what extent can a lightweight, locally executable auditing tool provide understandable and actionable results for developers without relying on external AI services or trained machine-learning models?

1.4 Research Objectives

The primary goal of this work is to create, develop and test a lightweight auditing tool to detect hallucinations and security flaws in LLM-generated Python code. The specific goals are:

1. To study software references and programming constructs in LLM-generated Python code, and their reliability and security concerns.
2. To create a mechanism for runtime introspection to check if imported modules and referenced attributes exist in the target Python environment.
3. Implement a security analyzer based on AST which detects pre-defined vulnerable programming patterns.
4. To correlate identified security problems with CWE identifiers and set severity levels.
5. To incorporate outcome of hallucination verification and vulnerability analysis into a composite risk-assessment mechanism.
6. To create a desktop graphical interface showing detected issues, source code locations of affected code, explanations, severity levels and overall risk scores.
7. To analyse the proposed application with representative Python code samples created by the LLMs and check for hallucinated references and security vulnerabilities.

1.5 Research Novelty

The novelty of this work is using two complementary auditing mechanisms in a single lightweight and explainable application of LLM-generated Python code. The application proposed in this work is different from those focusing on package hallucination or the classic security vulnerability detection for various reasons: First, it takes both software references and insecure source-code patterns into consideration in the same auditing process; second, it provides a more comprehensive solution for both software references and insecure source-code patterns. Some of the main new features are:

- Hallucination verification and security vulnerability detection are integrated with each other in one auditing application.
- Environment-based reference verification: Running-time introspection for imported modules and referred attributes to check against the real target Python environment.
- Explainable AST-based security analysis: Using simple rules on source code and linking security patterns found in the code with CWE identifiers, severity levels and explanations.
- UNA: United risk assessment — a combination of the results from the two analysis mechanisms to create an overall risk score for prioritization.
- No reliance on outside AI services, paid API access, or trained machine-learning inference models: Offering a desktop-based auditing solution without the need for outside AI services, paid API access, or trained machine-learning inference models.

1.6 Research Contributions

The major findings of this study are summarized below:

1. The concept is to provide an integrated auditing view that considers software-reference hallucinations and security vulnerabilities as complementary risks in LLM-generated Python code.
2. Technical contribution: Providing a runtime-introspection mechanism to ensure imported modules and referenced attributes are correct in the target execution environment.
3. Security analysis contribution: Design and implement an AST based rule analyzer that can identify the insecure programming constructs defined in the "rules" file and correlate the results with the CWE categories and severity.
4. Risk-assessment contribution: Proposing a composite risk-scoring mechanism that integrates risk-scores from hallucination and vulnerability analysis to enable code-risk prioritization.
5. Software implementation contribution: Providing developers with local, transparent and understandable code-auditing results in the form of a lightweight desktop application.
6. Experimental contribution: Application evaluation via five representative Python code samples and reporting 14 vulnerabilities (in five categories) and two hallucination findings.

 ISSN:2302-8149
e-ISSN:2540-9719

The study provides a practical post generation auditing approach that can be used in conjunction with current code generation and security assessment methods in LLM. Unlike proposed efforts to stop LLMs from producing malicious or insecure Python code, this suggests that the application is designed to analyze the code produced by the LLM and ensure that it is safe to use in a software project.

2 Related Works

One of the earlier and systematic attempts at assessing the security of AI-generated code using GitHub Copilot was conducted by Pearce et al. [9]. The researchers developed 89 programming problems relevant to cybersecurity based on high-risk weakness from the MITRE CWE Top 25 and created 1,689 codes through various prompts and in different domains. The code was then manually inspected for the presence of security vulnerabilities due to Copilot. It was found that roughly 40 percent of the generated codes contained security vulnerabilities, indicating that code generators could reproduce vulnerable programming practices. The research provided a valuable empirical base for evaluating security of AI-generated software. Nonetheless, the researchers were mainly concerned about prevalence assessment, but did not develop a lightweight auditing framework that would be useful for developers. This limitation is addressed by this research, as it integrates automated Python-code inspection with real dependency and API analysis and CWE-based security rules.

Perry et al. [10] examine if access to AI programming assistants impacts the security of the software code developed by developers. The authors conducted a user study with 47 users that performed programming tasks in the Python, JavaScript, and C languages with the help of an AI assistant created on the basis of Codex. The methodology includes the comparison of participants with and without access to the AI assistant along with the consideration of user trust and prompting. The findings have shown that participants using the AI assistant created less secure software and were more confident about the security of their code. In addition, the researchers noted that low trust in AI and more prompting reduced the number of vulnerabilities. However, the study focuses on the relationship between humans and AI assistants, not on the automation of the process of code review. The study neither verifies non-existent packages or APIs nor provides line-by-line explanation of vulnerabilities.

The proposed user experiment by Sandoval et al. [11] investigates the security implications of AI coding assistants on the performance of programmers using C code. They involved 58 participants who were tasked with implementing a singly linked list data structure with pointers and arrays in C programming language. The experimental method consisted of comparing AI-assisted programmers with a control group and evaluating the resulting programs for any critical security flaws. From their findings, the security effect for the specific programming task in C programming language was found to be relatively small with critical security errors appearing not more than about 10% more frequent than in the control group. This research is important since it shows how the security effect of AI coding assistants can heavily depend on the programming context. However, it tests a particular user situation instead of building an automated tool for generated code detection. The current research advances this direction through automated auditing of the generated Python code for software references and security vulnerabilities.

Package hallucination is studied by Spracklen et al., [12] whereby LLMs used to generate code may suggest or refer to packages which don't exist. In the research, 16 LLMs, two programming languages, and two prompt datasets were used to generate about 576,000 code samples. Package names used in generated responses were compared against existing package databases. High rates of hallucinations were detected, ranging from an average of 5.2% among commercial models and 21.7% among open-source models, along with over 205,000 distinct hallucinated package names. They examined mitigation strategies such as retrieval-augmented generation, fine-tuning, and self-correction. Nevertheless, the emphasis of their study was package hallucination prevalence and mitigation instead of developer code audit. Their study required package datasets and model-based code generation experimentations. In contrast to this, this paper provides verification of modules and attributes imported at runtime/environment alongside security vulnerability analysis.

A two-tiered model has been proposed by Raj and Sahu [13] for detecting package hallucinations which could be susceptible to slopsquatting attacks. In their method, the first tier

 ISSN:2302-8149
e-ISSN:2540-9719

involves a deterministic process that verifies the existence of a package in PyPI, while the second tier involves Random Forest classification based on package-name and metadata characteristics. An import-name resolution process has also been proposed to separate the Python import name from the package distribution name. These researchers have used 300 fine-tuned prompts in order to evaluate their technique and found that the pipeline was capable of generating error-free code in 76% of cases. The major limitation of this technique was its focus only on package-name hallucination and PyPI metadata rather than API, attribute, semantic, or any other security vulnerabilities within generated Python code. The current research tries to remove this limitation.

Morkonda et al. [14] carry out the comparative empirical evaluation of the security of code generated by seven widely-used LLMs. The methodology seeks to simulate realistic developer actions through the creation of source code by several models for various programming tasks, followed by checking the resulting programs for potential vulnerabilities. According to the authors, all models used for the testing generated programs that had some vulnerabilities; many of which could be considered high or critical. This study thus proves that the problem of insecure generation of code is not inherent only to one model or vendor. Still, the research itself only demonstrates the presence and prevalence of the issue, without offering a light-weighted approach to vulnerability detection from the developers' perspective. Additionally, it does not deal with the hallucination of APIs or non-existent dependencies. This work fills in that gap by regarding the problems of hallucination and vulnerability as complementary and developing a combined Python auditing tool.

The research by Aldosari and Aldawsari [15] examines the possibility to enhance the security of LLMs' output via prompt engineering. This method involves comparison of the code produced by various open-source LLMs in various programming languages, analysis of the impact of prompt creation and its degree of preciseness on security. According to the authors, there is a great deal of difference between models and languages, as well as significant increase in the safety of programs (up to 77%) through security-aware prompt engineering. Therefore, this research indicates that generation time intervention influences security outcomes. Nonetheless, the aforementioned approach is preventive and, therefore, cannot ensure safety of the particular code being generated. Neither it verifies non-existent API and dependencies post generation. The current research complements prompt engineering by analyzing code generation post generation, i.e. comparing it to real runtime environment using explicit security rules.

Al-Hawawreh et al. [16] undertake a systematic literature review of the security issues associated with AI-generated codes. The authors analyze existing works related to vulnerabilities that may arise due to the use of AI coding tools, focusing on known vulnerabilities like those included in the MITRE CWE Top 25. Thus, the literature review categorizes and organizes available information on security issues, potential threats and ways to mitigate them showing that the security issues posed by AI-generated code is a relatively new problem in software security. The value of this study is that it brings together scattered studies and highlights research priorities. However, as a literature review, this work does not offer any practical implementation of an auditing technique for vulnerability detection and hallucination verification. This study aims to fill in this gap by developing an auditing tool based on environment and AST approaches and including CWE explanations.

VulDeePecker, introduced by Li et al., [17] provides one of the earliest methods for automatically detecting software vulnerabilities based on deep learning. This technique involves encoding semantically similar statements as code gadgets, which then get mapped to vectors that can be fed to neural networks. The researchers created their own vulnerability dataset and showed that their proposed system had fewer false negatives than multiple systems already in use. VulDeePecker was also tested against Xen, Seamonkey, and Libav, and discovered vulnerabilities that did not exist in the National Vulnerability Database, but had since been fixed by the developers. However, VulDeePecker is only concerned with classical vulnerabilities in C/C++ code and relies on learned models and dedicated datasets. The current research aims to address AI-generated code and dependencies.

Li et al. [18] introduce SySeVR, which is a deep learning approach for detecting vulnerabilities in C/C++ source code by taking into account both syntactic and semantic aspects of code. This methodology makes use of program representations related to vulnerabilities with the help of syntax-based and semantic-based analyses and represents these program representations as vectors. Tests performed on four software products showed the viability of the approach, including the identification

 ISSN:2302-8149
e-ISSN:2540-9719

of 15 vulnerabilities not reported in NVD; seven out of those were unknown and eight had already been fixed silently. SySeVR is a significant enhancement of manually engineered security rules since it learns vulnerability-related representations automatically. However, this approach requires more computational efforts and applies only to C/C++ code and traditional source code in general rather than LLM-produced Python code. The current research overcomes the limitations of the previous approach by means of using Python AST analysis and environment introspection.

In addition, Zhou et al. [19] introduce Devign, a GNN-based method for detecting vulnerabilities through learning the complete program semantics through graph representations. The approach first transforms the source code into program representations and utilizes a customized convolutive module to learn the features for graph-level vulnerability classification. In their work, the researchers trained the model by manually labeling datasets taken from four different open-source C projects, rather than solely synthetic code. They found significant improvements compared to the current methods, with an average increase in accuracy of 10.51% and in F1-score of 8.68%. The research demonstrates the potential of the structural representation of programs for vulnerability detection. Nevertheless, Devign needs model training, labeled datasets, and graph construction, and it is tailored specifically for detecting vulnerabilities in C-based source codes. Moreover, the method does not verify the existence of the dependencies or APIs created by AI. The current research introduces an auxiliary solution that utilizes Python AST inspection and verification in the software environment.

LineVul is an approach for fine-grained vulnerability prediction based on Transformers proposed by Fu et al. [20] This technique uses Transformer representations for source code and tries to find vulnerabilities in line level instead of only function/file level. Authors perform evaluation on a large-scale dataset with more than 188,000 C/C++ functions. According to their findings, there are significant advances compared to baseline methods, such as up to 160-379% improvement in F1 score for function-level prediction, up to 12-25% improvement in Top-10 accuracy for line-level localization, as well as the reduction of efforts during vulnerability detection. It proves that Transformer-based models allow fine-grained vulnerability localization. Still, LineVul targets traditional C/C++ programs, needs trained model and a large-scale labeled dataset. Also, it cannot recognize any hallucinated packages and incorrect APIs. These gaps are addressed in the current research due to a combination of source-level security analysis with validation of Python dependencies and attributes.

According to Khanmohammadi et al., [21] WildCode is an empirical study of code created by ChatGPT during actual use cases and not just generated code from artificially created research prompts. In this study, the researchers test the quality of generated code based on correctness and security. Moreover, they assess the motivations behind user requests. This approach is crucial because many previous studies use artificially generated research prompts which do not always depict actual user behavior. The findings of their study offer additional proof of the poor security of ChatGPT code as well as the poor focus of users on security features in their requests for the generated programs. Nevertheless, WildCode focuses only on describing the problem and does not develop a dedicated lightweight tool to automatically audit dependencies, APIs, and potential security vulnerabilities of generated Python code. The current paper aims to fill this practical gap by transforming the identified risks into a process of automatic auditing through runtime introspection and AST security rules.

Belozerov et al. [22] examines the security of GPT-generated code through the use of the DevGPT dataset in the real world rather than the artificial prompts used in previous research. In their study, the authors analyze 2,315 C, C++, and C# code snippets through the use of static security scanners followed by manual inspection to find 56 confirmed vulnerabilities in 48 files. The authors then evaluate GPT-4o, GPT-4.1, GPT-5, and Claude Opus 4.1 for vulnerability detection and repair. From the findings of the study, it is evident that current LLMs have the capability to detect and repair a considerable number of vulnerabilities although there are other vulnerabilities which they cannot address. For instance, GPT-4o detected 18 out of 32 confirmed vulnerabilities while repairing 17. Nevertheless, the paper mainly centers on using LLMs to detect and repair vulnerabilities rather than examining hallucinated dependencies and deterministic API verification.

Cotroneo et al. [23] focuses on the distinctions between human-written code and AI-generated code using LLMs. Researchers analyze code written by ChatGPT, DeepSeek-Coder, and Qwen-Coder in regard to various dimensions of software quality including bugs, vulnerabilities, and complexity. In

 ISSN:2302-8149
e-ISSN:2540-9719

this research, the methodology is extended by taking a look at security from the point of view of software engineering along with maintainability and code structure instead of analyzing vulnerability as a standalone property. This is important to see if there is a qualitative difference in the risks presented by AI-generated code compared to the usual programming. Nevertheless, comparative evaluation itself is not enough for the developers to assess the generated program before its implementation. Furthermore, it does not focus on imaginary dependencies and APIs. This paper aims to fill the gap by offering a practical approach that will allow auditing every generated program with regard to software references and security problems.

This section provides a summary of recent studies investigating the security vulnerabilities and hallucinations in Large Language Model (LLM) generated code, focusing on detection methods, analysis approaches, and current challenges. Rather than just describing the results of previous studies, it critically evaluates the study and provides insight into what it has contributed, strengths and weaknesses. Focus is on the verification of software packages, modules, methods and attributes, and the identification of unsafe programming practices. In addition, the review investigates the properties of the current methodologies in terms of their ability to detect, computational load, interpretability, and usefulness. Recent references are also checked for completeness and accuracy for the reliability of literature. Last, the limitations found are incorporated to outline the research gap and give justification for the proposed lightweight auditing framework that combines runtime introspection and analysis of the AST to identify hallucinated software references and possible vulnerabilities in Python code produced by LLM.

- Enhance critical analysis: Make comparisons with previous studies in terms of methodology, detection techniques, evaluation strategies, and results, and not just descriptions.
- Highlight gaps in research: Discuss the research gaps and in particular the lack of addressing code hallucination and security at the same time.
- Present gaps in the research: Describe the need for lightweight, integrated and explainable tools for auditing LLM-generated Python code.
- Enhance comparative discussion: Make the proposed approach different from the current package-hallucination and vulnerability-detection approaches.
- Check recent references: Confirm authors' names, publication titles, year, journal, volume, page numbers, and dois.
- Enrich references: Add relevant recent studies (2025-2026) if available and make sure that in-text citations are present in the reference list.
- Get rid of unsupported claims: Make sure that reported findings and comparisons are the result of the original cited studies.

3 Research Methodology

The design and implementation process for the proposed LLM Code Auditor, a lightweight desktop application that can be used to analyze Python source code generated by Large Language Models (LLMs) such as ChatGPT, Claude, and GitHub Copilot, is presented. The proposed framework targets two complimentary categories of risks related to AI-generated software, namely: code hallucination and security vulnerabilities. Security vulnerabilities are patterns that could be identified as a security flaw in the Python software environment, while code hallucination involves the inclusion of non-existent packages, modules, methods and attributes. The main method will be automated, transparent, and local, giving a developer a tool to use to audit the code that is generated prior to it being added to a software project. The proposed approach comprises four major steps: (1) preprocessing and parsing of the source code, (2) hallucination detection at runtime using introspection, (3) AST-driven vulnerability detection using rules, and (4) risk assessment and result visualization. Hallucination detection is used to verify modules imported from a source and attributes referenced in a source against the actual module and attribute at the time of runtime. Security analysis works separately from code analysis to discover insecure coding patterns that are predefined in the program and to match the detected insecure patterns with the corresponding Common Weakness

 ISSN:2302-8149
e-ISSN:2540-9719

Enumeration (CWE) categories. The results of each of the two analysis components are then combined into an easily understood risk-scoring process to provide a general estimate of the severity of the analyzed code.

The process starts with LLM-generated Python source code as input, as illustrated in the methodology block diagram. The code is then analyzed and goes through two distinct paths of analysis, both of which complement each other. The first path is a runtime checker for imported modules and referenced attributes to look for possible hallucinations, and the second path is an AST-based security analyzer to check for vulnerable programming constructs. The results from both of these approaches are categorized by type and level of severity and correlated with the CWE ID's if applicable, to give an overall risk assessment. Lastly, the results are displayed via the graphical interface, such as detected issue, affected source-code line, explanation, severity level, and composite risk score.

The study adopts a Design Science Research (DSR) approach that aims to answer a practical problem by systematically designing, developing, and validating a software artifact. The LLM Code Auditor is the resulting tool and the evaluation focuses on its capability of detecting hallucinated software references and security vulnerabilities in representative Python programs generated by LLMs. A central design principle is explainability and reproducibility: Every reported finding is accompanied by an observable verification result or an explicit analysis rule. As a result, the findings for hallucinations are based on repeatable import or attribute-resolution tests, while the findings for vulnerability are based on real AST pattern matches that are linked to known security issues. This way, the decision-making process is not opaque, and the auditor's findings can be audited by the developers themselves.

3.1 Research Design and Framework Overview

The study follows a Design Science Research (DSR) methodology to design, develop and evaluate a lightweight desktop application called LLM Code Auditor. The proposed application aims at detecting two types of risks in LLM-generated Python code: software-reference hallucinations and security vulnerabilities. There are 4 major phases of the framework:

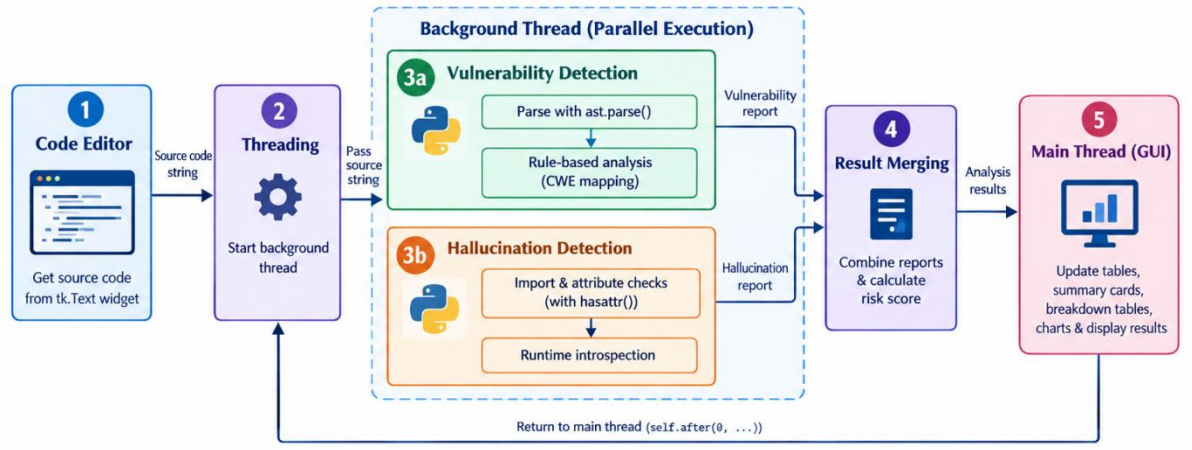
1. Source code is retrieved by the application editor and preprocessed and parsed to prepare for analysis. The Python Abstract Syntax Tree (AST) is a data structure that represents the structure of the program.
2. Hallucination detection: Runtime introspection is used to check if the imported modules and referenced attributes exist in the available Python execution environment.
3. Security vulnerability detection: An AST-based rule engine is used to scan the source-code structure for predefined insecure programming patterns, as well as mapping the detected vulnerabilities with the relevant CWE identifiers and severity levels.
4. The results obtained from both analysis mechanisms are then consolidated into a risk score between 0 and 100 which is reported. The application displays detected issues, source-code locations, explanations, severity classifications and overall risk assessment.

Both detection mechanisms are independent, and they give complementary information. They both pay attention to different aspects of the code: the hallucination detector examines for the existence of software references, while the vulnerability detector examines for any insecure programming constructs. No dependence on external AI inference services or a trained machine-learning model, no matter how complex; only Python standard library functionality and a graphical interface built with the Tkinter library.

3.2 Data Flow

 ISSN:2302-8149
e-ISSN:2540-9719

Figure (1) depicts the pipeline of analysis that is always followed when the user performs code analysis. The source code is first retrieved from the editor and is processed by a background thread to



preserve the reactivity of the GUI. The code is analyzed separately for any security flaws and code hallucination. Both analyzers produce a report of the findings detected during the analysis process. Once the analysis process is over, the findings are safely sent back to the main GUI where the application updates the tables of the findings, the risk score, the summary of statistics, and the graphs.

Figure 1. Dataflow Pipeline

One of the major methodological choices made was to code all the functionality in the application based solely on the Python standard library, without any third-party runtime dependencies. This includes GUI, code analysis, runtime inspection, multithreading, and data visualization through chart generation with native Tkinter drawing functions. Such an approach ensures greater internal validity, as it removes any additional dependencies on packages that may be included in the audit due to the supply chain or compatibility issues. It also increases portability and reproducibility of the application, as it can be run on any regular CPython 3.9+ installation without additional packages, internet, or virtual environments.

3.3 Dataset Description and Sample Selection

The experimental dataset used for the evaluation is comprised of 5 representative Python code samples selected to assess the ability of the proposed LLM Code Auditor in detecting them. The samples include various categories of intentionally injected and/or documented defects such as insecure use of APIs, hardcoded credentials, unsafe deserialization, weak cryptographic operations, and hallucinated software references, and cover common programming scenarios. The data has been structured into the following 5 application scenarios:

1. Backup and server-startup utility (S1): Contains backup operations, command execution, password hashing, server configuration and exception handling.
2. API data-fetch helper (S2): Considers communicating with APIs, using third-party packages, method availability and TLS configuration.
3. Authentication helper (S3): Authentications operations and credential handling, SQL query construction and access-control checks.
4. Configuration and cache operations (S4): Analyze loading of configurations, handling of data, and possibly insecure programming.
5. Utility operations (S5): These are generic Python operations and there is a relatively low severity case that is used to explore the analyzer's action with a limited number of vulnerabilities.

The five samples are tested for the hallucination detector and vulnerability detector one by one. The outcome of this is captured by category of detection, location in source code, CWE classification if applicable, severity, and risk score. The set of data is supposed to serve as a small representative set for functional evaluation. It is not meant to be a representative sample of all Python code written by LLM. Based on the sample selection and defect descriptions, the proposed detection mechanisms are analyzed to determine if the issues being detected are issues that are targeted.

 ISSN:2302-8149
e-ISSN:2540-9719

3.4 Mathematical Model

The mathematical model represents the proposed system for analyzing Python code formally. Where the above sections present a description of the framework from a software engineering perspective, in this section the mathematical objects that form the basis of the framework are outlined and the procedure by which source code is converted into module classifications, attribute verification results, vulnerabilities and the final risk score is presented. The proposed model uses operations that are directly executable through the Python interpreter. In this context, module verification is based on import lib import module whereas attribute verification is based on hasattr. The approach makes no assumptions about the availability of manually created documentation for Python packages and allows for the reproduction of classification under a predetermined Python execution environment. The proposed model analyzes Python source code to identify invalid packages, unsupported attributes, and security vulnerabilities. The source code S is first converted into an Abstract Syntax Tree (AST):

$$T = \text{parse}(S). \quad (1)$$

A package is considered valid when it can be successfully imported. Otherwise, it is classified as an invalid or potentially hallucinated package:

$$H_{\text{pkg}}(m) = \{0, \quad \text{import}(m) \text{ succeeds}, \ 1, \quad \text{import}(m) \text{ fails}. \quad (2)$$

Similarly, an attribute is verified using the corresponding Python object. A failed attribute check indicates an unsupported API:

$$H_{\text{attr}}(a) = \{0, \quad \text{hasattr}(o,p) = 1, \ 1, \quad \text{hasattr}(o,p) = 0. \quad (3)$$

The detected vulnerabilities and invalid APIs are combined into a raw risk value r. The final risk score is normalized to a range from 0 to 100:

$$R = 100\left(1 - e^{-r/12}\right). \quad (4)$$

A higher value of R indicates a higher overall risk in the analyzed Python source code.

3.5 LLM Models and Code Generation

The proposed application aims to audit Python source code created by Large Language Models (LLMs), such as code that is generated with the help of AI-powered programming tools. For the experimental evaluations, five typical Python programs are used, which include backup and start-up operations of a server, API data retrieval, authentication, configuration loading, and utility operations. Experimental samples are never used to train a machine-learning model, but simply used as input artifacts to evaluate the auditing application. The proposed application does not need to access the LLM that originally generated the source code as the analysis is done after the code is already generated.

If available, the exact LLM provider, model name, model version, generation date, prompt and generation settings should be documented for each sample to ensure reproducibility. These data should be captured on an experimental configuration table. Where ChatGPT, Claude or GitHub Copilot are mentioned, this is a general reference to the available sources of code generation, and does not imply that all three were used in the experiments reported here. The auditing framework itself is model-independent: it takes the Python code, and does the same things to detect it no matter which LLM is used to generate the code.

3.6 Hallucination Detection Methodology

A static allow list or deny list is not sufficient for hallucinated packages since the list of Python distributions and APIs changes over time. The proposed methodology therefore, does not depend on lists of packages in advance but uses real running-time introspection of the code. The detector checks if referenced modules and attributes exist in the current Python environment or not. This offers a direct and repeatable way to determine potentially hallucinated packages and APIs.

3.6.1 Hallucination Detection Pipeline

The hallucination detector goes through four stages of analysis on the given Python source. The source code is first parsed and all import statements gathered in order to make a mapping between local names used in the source code and their module paths. Resolution of the relative imports is dependent on the package structure context and so they are out of scope. Second, the imported

 ISSN:2302-8149
e-ISSN:2540-9719

materials are checked with current environmental conditions. Standard-library modules are identified in real time and other modules are tested by performing real runtime import operations. Valid imports are those which are successful. If an entire module path isn't found, but the top-level package is, the called package/API is considered a likely hallucinated API. If these cannot be solved, the result is considered as possibly missing and/or fake package. Thirdly, an optional PyPI verification step is added to separate an uninstalled package from a fabricated package. A successful PyPI response is a sign of the distribution being available, but it could just be that it is not available locally; in contrast, a not-found response is a more compelling sign of fabrication. If there are network failures, they do not affect the original confidence evaluation. A small mapping of common import and distribution names is used only for the purpose of improving the accuracy and clarity of the external lookup messages.

Lastly, the detector scans attribute chains in the source code. For every attribute referenced, the system backtraces the expression to the referenced root object and compares the referenced attribute with the runtime object. If an attribute is not found and it is not possible to determine the value, a high-confidence hallucinated API finding is created. Findings are duplicated to eliminate redundant findings and to enhance the clarity of the final analysis.

3.6.2 Confidence Levels

Table (1) summarizes how confidence levels are assigned; this three-tier scheme is designed to communicate the *strength of evidence* behind each finding rather than a subjective severity judgment.

Table 1. Confidence levels used by the hallucination detector.

Confidence	Evidentiary basis
High	A concrete, reproducible failure was observed: either an attribute genuinely does not exist on a successfully imported object (`hasattr()` = False), a submodule does not exist inside a package that itself imports fine, or a PyPI lookup returned HTTP 404 for the package name.
Medium	The import failed locally and no network check was performed (or none was requested); the package may be real but uninstalled, or entirely fabricated — both remain possible.
Low	The import failed locally, but a PyPI lookup confirmed the distribution genuinely exists; the most likely explanation is a missing local dependency rather than a hallucination.

3.7 Vulnerability Detection Methodology

The vulnerability detection module implements a static analysis tool that helps detect vulnerable code patterns in the input Python code. Static analysis is done by analyzing the Abstract Syntax Tree without running the code and thus safely and consistently detecting any vulnerabilities in the code. The static analysis technique uses AST-based rules along with auxiliary checks of the code, taking into account such problems as code injection, command injection, unsafe deserialization, cryptography, network settings, and hard-coded credentials. Every found problem is marked with a CWE id and severity, while the source location and explanation are stored for further review.

3.7.1 Rule-Based Static Analysis

The vulnerability scan leverages a one-time pass abstract syntax tree (AST) analysis with additional line-level checks. The analysis engine does not run or load the code sample, but rather discovers the specified insecure code constructs via syntax analysis. Each of the discovered vulnerabilities is linked to its CWE ID, thus enabling consistent classification of the security issue discovered. Furthermore, each detected vulnerability is categorized into one of the following five severity classes: Critical, High, Medium, Low, or Info.

3.7.2 Detection Rules

Rule set for vulnerability detection in Table (2) shows the rules that have been used in the proposed analyzer and classifies them based on the detection methodology behind them. These rules

 ISSN:2302-8149
e-ISSN:2540-9719

cover various types of vulnerabilities commonly found in Python applications, such as unsafe code execution, command and SQL injections, insecure deserialization, weak cryptography, hard-coded credentials, and insecure networking. For each rule, this table indicates the CWE ID, severity level, the code pattern vulnerable to that problem, and the detection method that can be applied.

Table 2. Vulnerability detection rules and their CWE mappings.

CWE	Severity	Pattern	Detection mechanism
CWE-95	Critical	eval() / exec()	Direct ast.Call name match; an additional rule specifically flags eval(input()) as a distinct, maximum-severity "direct code injection" pattern.
CWE-78	High	os.system, subprocess.(shell= True)	Dotted-call-name match, plus a keyword-argument scan for shell=True on any of subprocess.call/run/Popen/check_call/check_output.
CWE-89	Critical/High	SQL built via + concatenation, f-strings, or .format() before .execute()/. executemany()	Inspects the first argument node of any .execute/.executemany call: ast.BinOp(Add) and ast.JoinedStr are rated Critical; a .format() call is rated High.
CWE-502	Critical/High	pickle.load(s), marshal.loads, unsafe yaml.load	Direct call-name match; yaml.load is suppressed only when a keyword argument's value AST contains the substring Safe (i.e., an explicit SafeLoader).
CWE-327	Medium	hashlib.md5, hashlib.sha1	Direct call-name match.
CWE-338	Low	random.random used where a secure PRNG is implied	Direct call-name match.
CWE-377	Medium	tempfile.mktemp	Direct call-name match (race-condition-prone API).
CWE-798	High	Hardcoded credentials	ast.Assign nodes whose target name matches a case-insensitive regular expression for password|secret|api_key|access_token|auth_token|private_key and whose assigned value is a non-empty string literal.
CWE-295	High	requests.get/post/request(..., verify=False)	Dedicated AST walk for calls with attribute name get/post/request carrying a verify=False keyword constant.
CWE-489	Medium	debug=True left enabled	Line-based substring scan (excluding comment lines).
CWE-703	Low	Bare except:; assert for validation	ast.ExceptHandler with type is None; every ast.Assert node (since assertions are stripped under python -O).
CWE-547 / CWE-668	Info/Low	Hardcoded IP addresses; binding to 0.0.0.0	Regular-expression IPv4 matching per source line (excluding loopback and the wildcard bind address itself for the IP rule); a separate substring check for host="0.0.0.0".
CWE-1188	Info	Mutable default function arguments	ast.FunctionDef nodes whose positional or keyword defaults include an ast.List, ast.Dict, or ast.Set literal.

 ISSN:2302-8149
e-ISSN:2540-9719

3.7.3 Traceability of Findings

The vulnerability discovery includes a list of the following elements: source code line number, CWE ID, the level of severity of the issue, title and description of the vulnerability. Source code fragments related to discovered vulnerabilities are kept when possible. It allows associating the discovered vulnerability with its place of origin and verifying it independently from the results of the analysis.

3.7.4 Ordering of Findings

Multiple findings can be created for one source file, and these findings have different severity ratings. Hence, findings are arranged based on the severity rating of each finding, starting with the Critical findings, followed by the High, Medium, Low, and Info findings. Within the same severity level of findings, they are sorted based on their line number in the source code.

3.8 Security of the Analysis Process

Security of the auditing process is of importance as the submitted python programs might be full of malicious and unsafe instructions. The proposed framework thus involves two steps: source-code analysis and software reference verification. The AST based vulnerability analyzer uses only the source-code representation that has been parsed. Does not run, test or store the program submitted. Rather, it detects potentially dangerous operations based on their syntactic structures and compares them to a set of security rules. This design is created to ensure that the vulnerability analysis phase cannot send instructions that are malicious to the machine.

The procedure is different for the hallucination detector since module and attribute verification necessitates interaction with the Python environment. It tries to resolve imported modules and check attributes referred to. This process should, therefore, be undertaken in a controlled Python environment, for instance a virtual environment set up on a trusted network with limited access to other systems, and permissions. The application's GUI, parsing, analysis and reporting are done using the standard library modules of Python. To maximise the application responsiveness, Analysis tasks are executed separately from the graphical interface. The results are not run against the program analysed, but returned to the interface to display.

Static analysis should not be confused with runtime reference verification – while the target source code is not run by the AST analyzer, an external module that is imported during introspection may run its initialization code. So, the current implementation is not a completely sandboxed execution environment. Additional isolation e.g. in the form of restricted containers or disposable virtual machines are recommended for auditing untrusted third party dependencies.

3.9 Risk-Scoring Methodology

The risk-scoring framework outlined herein is a compact way of expressing the security and reliability problems found via code analysis through a numerical scale. Instead of listing problems in an unsorted way, this framework incorporates the severity of vulnerabilities and the confidence in hallucinations into a unified numerical score between 0 and 100. Weighted scores give higher priority to serious vulnerabilities and confident hallucinations, while exponential saturation ensures the numerical score remains reasonable. This score can then be categorized according to predefined risk levels for easy comparison between different analyzed files and different instances of analysis.

3.9.1 Motivation

A raw, unordered list of dozens of findings does not, on its own, tell the reader whether a particular file or an analyzed file is more or less risky compared to other files or other analyzed files, or whether a change to the file from the first run to the second run improved the overall picture or not. In order to provide this easy-to-compare relative assessment, the methodology contains a single 0–100 risk score which is calculated clearly from the two risk objects reported.

 ISSN:2302-8149
e-ISSN:2540-9719

3.9.2 Weighting Scheme

The point weights assigned for the vulnerability and hallucination findings are summarized in Table (3) below. The weightings for the vulnerability findings take into account their severity, with critical vulnerabilities getting the highest penalty as they could have more of a security impact. The confidence level assigned by the hallucination-detection component is used to weigh the findings in terms of hallucination. These weights will offer a common denominator to combine individual findings to create a single risk score, ranging from 0 to 100.

Table 3. Point weights used in the risk-score heuristic.

Category	Level	Point Weight
Vulnerability Severity	Critical	10
	High	5
	Medium	2
	Low	1
	Info	0.5
Hallucination Confidence	High	3
	Medium	1.5
	Low	0.5

3.9.3 Saturating Transformation

The raw weighted sum, r, is unbounded and would allow a single very noisy file to produce an uninformative, off-scale number. To keep the score within a bounded, interpretable $[0,100]$ range while still preserving fine-grained separation between low-risk files, the raw sum is passed through an exponential saturation function:

$$\text{score} = 100\left(1 - e^{-r/12}\right) \quad (5)$$

where $r = \sum_{f \in \text{vuln}} w_{\text{sev}}(f) + \sum_{f \in \text{hallu}} w_{\text{conf}}(f)$.

The divisor of 12 was chosen empirically so that a small number of *Critical*-severity findings (e.g., two or three, contributing $r \approx 20$–30) already pushes the score past the midpoint of the scale, reflecting the intuition that a handful of severe, exploitable issues should dominate the overall risk assessment even in an otherwise large and low-severity-heavy codebase. The final score is clamped to the integer range $[0,100]$ via $\max(0, \min(100, \text{round}(\text{score})))$.

3.9.4 Risk Bands

Table (4) shows the interpretation of the calculated 0–100 risk score using qualitative risk bands. The classification provides the user with an intuitive risk level that can be obtained from the numerical score, and thus the overall security status of the analyzed code can be quickly assessed. Every band has a specific colour representation which is applied throughout the summary banner, risk bar widget and exported report, creating a visual consistency and enabling quick interpretation of analysis results.

Table 4. Qualitative risk bands derived from the 0–100 score.

Score range	Band	Color
≥ 70	High Risk	Red (`#b71c1c`)
$[35,70)$	Moderate Risk	Amber (`#f9a825`)
$(0,35)$	Low Risk	Green (`#2e7d32`)
0	No Issues Found	Green (`#2e7d32`)

3.9.5 Explicit Scope Limitation

The proposed risk score is intended to be a relative risk comparison and prioritization indicator that is based on a documented but non-formal or non-standard heuristic for cybersecurity risk assessment, rather than an actual risk metric. The score is not designed to be used to replace existing vulnerability-scoring systems like the Common Vulnerability Scoring System (CVSS). The score will be intended to be a simple, easily understood number that would allow for easy comparing of analysis

 ISSN:2302-8149
e-ISSN:2540-9719

results to the score and to use to prioritize the files or findings for further analysis. This is because the numerical value and qualitative risk bands are specific to the heuristic defined in this study and are not necessarily directly interchangeable with the standardized vulnerability scores.

This restriction is built into the methodology, in the sense that the scope of the heuristic needs to be clearly expressed, in order to maintain transparency. The user interface and documentation of the system therefore clearly state that the score is a heuristic value, and that it is not an objectively calibrated probability of exploitation, a severity rating or a standardized security assessment. The explicit scope helps to avoid the perception that the risk score is more rigorous or authoritative than the methodology warrants.

3.10 Experimental Environment and Reproducibility

Using Python and its built-in libraries, the proposed application is implemented. The graphical interface is built on the Tkinter library and the source-code analysis is performed using Python AST parsing, module-import facilities and attribute inspection. The application is written for CPython 3.9+ and does not require external AI APIs, 3rd party runtime packages or a trained detection model. To ensure reproducibility, the same five samples of source code should be analyzed under the same python environment and application configuration. The hallucination and vulnerability analysis modules are applied independently to each sample. The reports created are kept for comparison and examination.

Evaluation is recorded in the number of vulnerabilities found, the categories of the CWE vulnerabilities, the severity distribution, the hallucination results, the level of confidence in the results of the hallucination, and the composite risk score. The source-code line numbers and explanation are also checked to ensure that the reported results are in the correct line of code. The results of the experiments are given as an indicator of the functionality of the proposed application, but not as a comprehensive statistical evaluation of detection accuracy, since the dataset used for evaluation is very small and consists only of five representative programs.

4 Graphical User Interface Design

The proposed LLM Code Auditor's graphical user interface (GUI) aims to create a user-friendly and centralized platform to load, analyze, review, and interpret Python code at the same time. The interface features a 2-panel design; one panel is the source-code workspace and the other panel is the analysis-results workspace, so that the user can view the input code and output results within a single pplication window. Figure (2) illustrates the main GUI of the developed system.

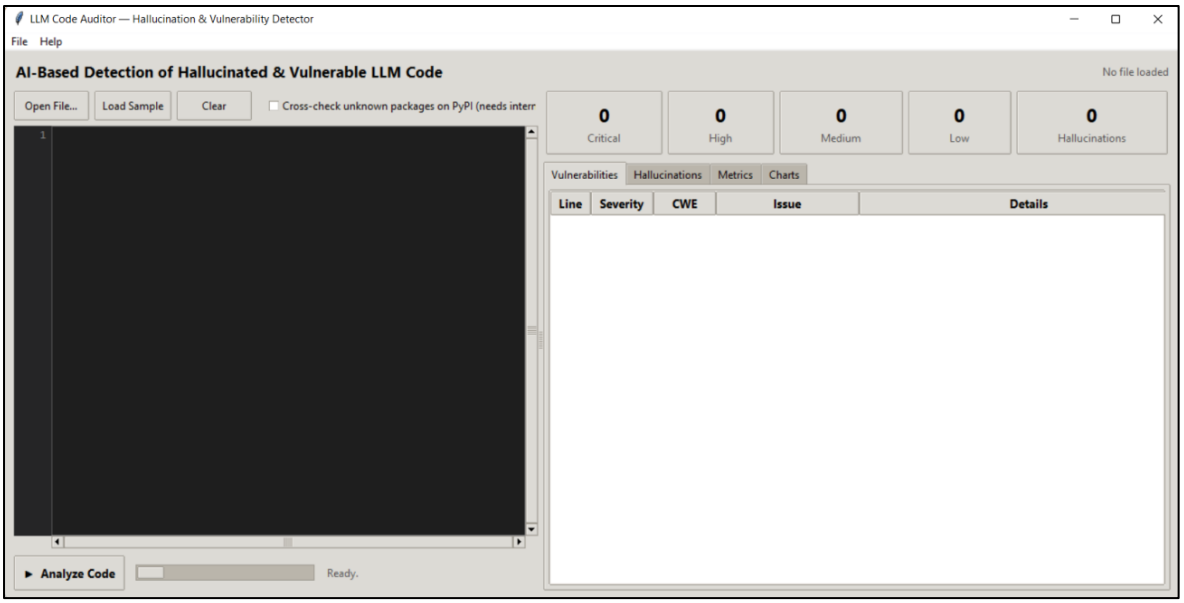


Figure 2. LLM Code Auditor's graphical user interface (GUI).

 ISSN:2302-8149
e-ISSN:2540-9719

The top left will have the application name and main file controls. The Open File… button lets the user choose a python source file from the local system, and the Load Sample button loads the built-in sample demonstration source code that includes intentionally buggy and hallucination-prone examples. The Clear button clears the current source code and resets the workspace. Additionally, an optional checkbox "Cross-check unknown packages on PyPI" is added to verify packages if necessary. Source-code editor occupies the left side of the main window. The editor displays a dark text box with line numbers so that users can see and analyze the Python source code before and as they are running their program. A vertical scrollbar provides scrolling for longer source files. This facilitates the ability of the user to determine the specific context of the reported findings in the user's source code.

In the top right corner, there are a series of indicators called summary indicators that give a quick look at the results that are detected. Each vulnerability finding is counted separately for Critical, High, Medium and Low, as well as in a special Hallucinations counter. These indicators are revised based on the analysis and do a quick assessment of findings found to determine its number and severity distribution. The results section is broken down into 4 notebook tabs below the summary indicators, which are: Vulnerabilities, Hallucinations, Metrics, and Charts. The Vulnerabilities tab displays discovered security vulnerabilities in a tabular format with the fields: Line, Severity, CWE, Issue and Details. Its Hallucinations tab offers a similar interface for results of the hallucination-detection component. The Ch charts tab displays data about the distribution of vulnerabilities and hallucinations graphically, and the Metrics tab displays the aggregated analysis statistics and the calculated risk score.

On the bottom left are the Analyze Code control and a progress indicator. Clicking this option starts the analysis pipeline. The status message next to the progress indicator indicates the current status of the application, e.g. "Analysis ready," "Analysis complete. Running the analysis on another thread outside of the GUI thread frees up the GUI so it is not processing the source code and therefore is less unresponsive. In general, the GUI provides a complete workflow of source-code entry, automated analysis, finding inspection, statistical summary and visualization. The design focuses on the direct access to evidence: loading source code, running the analysis, viewing specific results, viewing aggregate results, viewing a risk assessment and interpreting the results without the need to use separate software. Thus, this interface acts as the operating level where the methodology described in the previous sections is shown to the end user.

4.1 Vulnerability Distribution Analysis

The validation results give a quantitative perspective on the vulnerabilities that the proposed LLM Code Auditor identified. Figures (3) and (4) are complementary views of the same set of vulnerability findings. The results of this analysis are organized into the categories shown in Figure (3) by severity, and Figure (4) by the Standard Enumeration of Common Weaknesses (CWE) categories. The two graphs show the severity distribution and type distribution of the security issues detected.

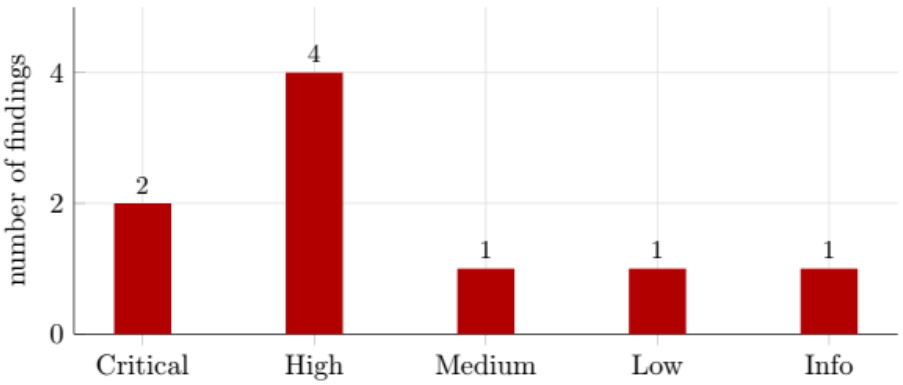


Figure 3. Vulnerability Findings by Severity.

 ISSN:2302-8149
e-ISSN:2540-9719

The severity distribution of the nine vulnerability findings found in the validation fixture are given in Figure (3). The results indicate that four vulnerabilities were marked as High severity, while two were marked as Critical. There are three levels of "severity": high, medium, and low, and each of these severity levels has one finding. Hence there is a considerable percentage of findings classified as High and Critical, with six of the nine findings in this category, a sign that this validation fixture has a high percentage of findings at the high and critical levels. The distribution also shows that not everything the detector detects is at the same severity level. Instead, the results cover the complete range from Critical to Informational findings. This variation is relevant to the risk-scoring portion of the proposed heuristic because the proposed heuristic assigns point values to findings based on the severity of the finding. Therefore, the severity distribution in Figure (3) directly affects the overall risk score obtained from the source code analyzed.

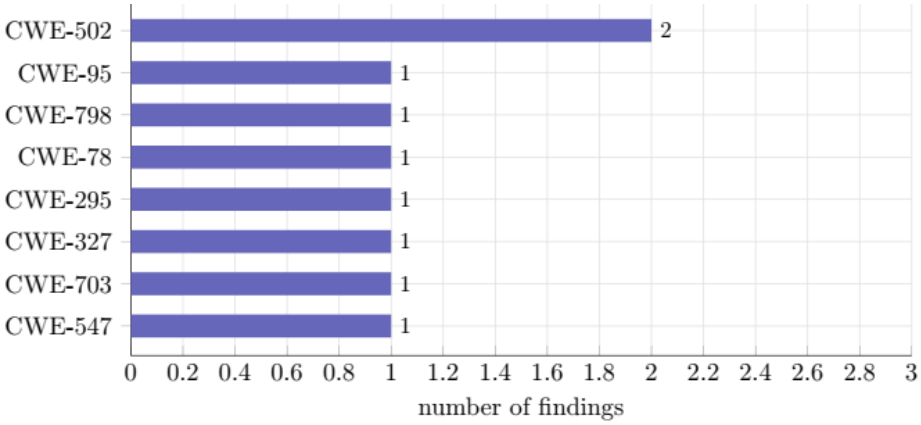


Figure 4. Vulnerability Findings by CWE Category.

The same nine vulnerability findings are presented in a categorized fashion on CWE identifiers in Figure (4). There is only one category with more than one finding, namely CWE-502 (Deserialization of Untrusted Data), which has two findings. The remaining categories—CWE-95, CWE-798, CWE-78, CWE-295, CWE-327, CWE-703, and CWE-547—each occur once. Both the CWE-502 findings are related to different insecure deserialization patterns in the validation fixture, one on Pickle.load and the other on insecure call to yaml.load. This shows that the detector is able to recognize multiple occurrences of the same general weakness class, even in different parts of the source code. The rest of the CWE categories cover a wider set of security vulnerabilities, and hence have a heterogeneous vulnerability profile. Following this severity-based analysis (shown in Figure 4), Figure (5) demonstrates that vulnerabilities in the validation fixture exist in more than one category of CWE, rather than being restricted to a single type of vulnerability.

5 Results and Discussions

The experimental results of the proposed LLM Code Auditor evaluation with a particular focus on its two complementary static-analysis components, namely the LLM hallucination detector and the LLM vulnerability detector are presented in this chapter. The evaluation was performed on five representative Python code examples that were chosen to closely resemble samples of real code that might be produced by a LLM from a typical set of programming queries. The samples contain various security and code-generation flaws, such as insecure API usage, hardcoded credentials, unsafe deserialization, weak cryptographic methods, hallucinated third-party packages, and hallucinated attributes attached to real Python modules. A sample of relatively low severity was also added to test if the tool does not over report where there is a limited security weakness. All five samples were processed in the two analysis entry points, `vulnerability_detector.analyze_source()` and `hallucination_detector.analyze_source()`. The resulting results were then combined together based on the same composite risk scoring heuristic (0–100) applied in the graphical application. The evaluation thus goes beyond the individual detection capability of the two analysis components, but also how the results of these components are combined into an overall risk assessment.

 ISSN:2302-8149
e-ISSN:2540-9719

The results are provided in a multi-layer format. An overview of the samples assessed and the intentionally introduced defect classes is given first. Second, the total number of vulnerabilities and hallucinations found along with the risk scores are compared between samples. Third, per-sample results are displayed in detail to understand the results found in each sample and their severity, CWE classification, and hallucination confidence level. Finally, the introspection statistics of the hallucination detector were analysed to measure the percentage of the hallucination's imported modules and attributes that were resolvable during introspection. These outcomes are then discussed with regard to what was observed and what limitations do you see in your evaluation.

5.1 Overview of Experimental Results

Five representative backup and server startup, API data retrieval, authentication, configuration and cache loading, and token generation Python programs were tested and evaluated for the proposed LLM Code Auditor. The hallucination detection and the vulnerability detection modules were used to examine each sample. The criteria evaluated were number of findings, severity of findings, CWE classification, hallucination confidence and composite risk score.

In total, the application detected 14 vulnerability findings and 2 hallucination findings, resulting in a total of 16 findings in five samples evaluated. The average composite risk score was 54.4/100. The results show that the proposed framework was able to detect various categories of security weaknesses and unresolved software references in typical Python programs. However, the results are not statistical with five samples in the dataset being a functional interpretation of the results of the detection process.

5.2 Sample Selection

Prior to elaborating on the results of the experiments in detail, it is important to outline the characteristics of the dataset that will be used to evaluate the performance of the auditing system under development as well as the nature of the vulnerabilities that have been deliberately injected into each test program. Table 1.2 lists the five Python programs employed for the evaluation and provides a brief description of their application scenarios as well as of the classes of vulnerabilities or hallucinations included into them.

Table 5. Summary of the five evaluated code samples.

ID	Scenario	Defect classes intentionally present
S1	Backup / server-startup utility	Command injection, weak hashing, debug mode, insecure bind, bare `except`, hardcoded IP
S2	API data-fetch helper	Hallucinated package, hallucinated method on a real module, disabled TLS verification
S3	Authentication helper	Hardcoded API key, SQL injection, arbitrary code execution, `assert`-based access control
S4	Config / cache loader	Insecure deserialization, unsafe YAML loading, mutable default argument
S5	Token / greeting utility	Weak PRNG for a security-sensitive purpose (comparatively benign sample)

5.3 Aggregate Results

Table (6) provides the results of the evaluation of the five Python scripts. In Table 1.3, the number of vulnerability findings and hallucination findings found by the tool, total number of findings, total risk score, and corresponding risk band are provided for each script. Overall, from the entire evaluation dataset, the tool was able to detect 14 vulnerability findings and 2 hallucination findings, amounting to 16 findings. The average risk score across all five scripts is 54.4.

Table 6. Aggregate findings and risk scores per sample.

Sample	Vuln. findings	Hallu. findings	Total	Risk score	Risk band
S1	6	0	6	62	Moderate Risk
S2	1	2	3	55	Moderate Risk
S3	3	0	3	74	High Risk

 ISSN:2302-8149
e-ISSN:2540-9719

S4	3	0	3	73	High Risk
S5	1	0	1	8	Low Risk
Total / Mean	14	2	16	54.4	–

The findings result also show that the number of the findings does not always dictate the risk score. S1 has the most findings, with 6 vulnerability findings, and got a risk score of 62 and was rated as Moderate Risk. By comparison, S3 only found 3 vulnerabilities, but scored the highest point at 74 (High Risk), whereas S4 found 3 vulnerabilities and scored 73 (High Risk). The difference can be attributed to the severity weighting included in the proposed risk-scoring heuristic. The findings found in S1 were spread across severity levels High, Medium, Low, and Info with no Critical severity level. While S3 had an arbitrary code-execution finding of Critical severity which exploits eval(), S4 had an insecure-deserialization finding of Critical severity which exploits pickle.load(). These Critical findings therefore have a significantly more impact on the final score than the higher number of lower severity findings in S1.

S2 has one vulnerability and two hallucination findings for a total of three findings with a score of 55 (Moderate Risk). This shows that the findings for hallucination can add to the composite score but not to a high-risk classification if the confidence weight for the finding and the severity of the vulnerability are not very high. The relatively benign S5 found only one Low-severity vulnerability finding, with the lowest score being 8 – and a Low-Risk classification. This outcome is significant in that it shows that the heuristic differentiates between a sample that has a relatively minor, less serious problem and one that has more serious security problems. The overall results of the aggregates suggest that the proposed scoring system is more closely linked with the severity of the findings than with the quantity of findings. A smaller number of high impact vulnerabilities may thus have a higher risk score than a larger number of lower severity vulnerabilities. This is similar to the heuristic's intended use as a prioritization and comparison tool and not a strict metric of finding count.

5.4 Per-Sample Results

In the per-sample analysis, there is an in-depth evaluation of the results that have been produced by the suggested LLM Code Auditor per each Python code sample under review. Every Python code sample is reviewed based on its specific usage case, discovered vulnerabilities, hallucinations, level of confidence, CWE type, and total risk score. The in-depth evaluation supplements the summary with the actual information about the discovered problems and helps understand how particular results contribute to the overall risk assessment. The results for Samples S1-S5 are provided below, both security-related and methodological.

5.4.1 Sample 1- Backup / Server-Startup Utility

Sample 1 is a sample of a backup and server-startup utility used to test the vulnerability detector against different vulnerabilities that may be found in otherwise plausible Python code. This sample includes operations such as backup creation, password hashing, and server start. Though the sample operations are plausible and look like operations performed often in programming, there are several decisions that make them vulnerable. Six vulnerability detection points were found; these are related to command injection, weak password hashing, debug mode activation, unrestricted network binding, general exception handling, and a hardcoded IP address. The calculated total vulnerability score is 62/100, or Moderate Risk.

```
import os
import hashlib

DB_HOST = "192.168.1.57"

def run_backup(user_folder):
    os.system("tar -czf backup.tar.gz" + user_folder)

def hash_password(password):
    return hashlib.md5(password.encode()).hexdigest()

def start_server():
    app.run(host="0.0.0.0", debug=True)

try:
    run_backup("/tmp/data")
except:
    pass
```

 ISSN:2302-8149
e-ISSN:2540-9719

The Table (7) shows the vulnerability detections found in Sample 1, along with the source-code line, severity level, CWE, and description of each discovered problem. This proves that the vulnerability scanner is capable of identifying several security vulnerabilities within a single code sample and differentiating between them based on their severity level. These include both the High-severity Command Injection problem and the Informational Hardcoded IP Address problem, among others.

Table 7. Findings for Sample 1 (risk score 62/100, moderate risk).

Line	Severity	CWE	Finding
6	High	CWE-78	OS command execution via string-concatenated *os.system()* call (command injection risk)
9	Medium	CWE-327	Weak hash algorithm (*hashlib.md5*) used for password hashing
12	Medium	CWE-489	Debug mode enabled (*debug=True*)
12	Low	CWE-668	Server bound to all network interfaces (*host="0.0.0.0"*)
16	Low	CWE-703	Bare *except:* clause silently swallows all errors
3	Info	CWE-547	Hardcoded IP address (*192.168.1.57*)

No hallucinated packages or APIs were reported for this sample; both imports (*os*, *hashlib*) are standard-library modules that resolved successfully, and all attribute accesses the detector could trace back to a concrete module existed as expected. The sample illustrates a common LLM failure mode: individually plausible, "textbook" function calls (backup via *tar*, hashing a password, starting a dev server) that are each insecure in a networked or production context.

5.4.2 Sample 2 — API Data-Fetch Helper

Sample 2 illustrates the API data fetching application where external package dependencies are used in combination with the HTTP protocol and number arrays handling. The code uses the requests, fastjsonhelper, and numpy libraries and creates functions that help to analyze data, fetch the content from the URL, and convert rows into the matrix representation. As per the analysis, there was found 1 vulnerability and 2 hallucinations related to packages and APIs, which gave the total risk score of 55/100 and corresponded to the moderate level of risk. The vulnerability relates to the disabled TLS certificate verification, while the hallucinations detector revealed a non-importable package and non-existent API method in the requests package.

```
import requests
import fastjsonhelper
import numpy as np

def analyze(data):
    result = requests.get_json_super_fast(data)
    return result

def fetch_safe(url):
    return requests.get(url, verify=False)

def to_matrix(rows):
    return np.array(rows).reshape_fast(-1, 1)
```

Table (8) illustrates the findings identified in Sample 2. This table provides information about the source code line number, finding type, severity/confidence, category, and description for all findings identified in the sample. From these findings, it is clear that Sample 2 includes a High CWE-295 finding due to TLS certificate verification being turned off in the sample along with a Medium Hallucination package and a High Hallucination API.

 ISSN:2302-8149
e-ISSN:2540-9719

Table 8. Findings for Sample 2 (risk score 55/100, moderate risk).

Line/Type	Severity/Conf.	Category	Finding
9 (vuln)	High	CWE-295	TLS certificate verification disabled (`verify=False`)
2 (hallu)	Medium conf.	Hallucinated Package	`fastjsonhelper` could not be imported and does not correspond to a real, installed distribution
5 (hallu)	High conf.	Hallucinated API	`requests.get_json_super_fast` does not exist on the real, successfully-imported `requests` module

Sample 2 shows the two types of hallucinations that can be targeted by the proposed hallucination detector, Hallucinated Package and Hallucinated API. The first is fastjsonhelper, a package that looks like a real package but is not installed and can't be imported. This is package-level hallucination, in which the generated code has some dependency it does not actually have. The second one is a little more nuanced since the module 'requests' is a valid and imported Python module. The code is "requests.get_json_super_fast(data)" but calls get_json_super_fast is not a function of the requests module. The detector turns the imported name into the actual module object and calls hasattr() to check whether the attribute it is looking for is present in the module object. As a result, it would correctly report a nonexistent package as a Nonexistent API, whereas a real module with a nonexistent API is correctly reported as a High-confidence Hallucinated API. The np.array(rows).reshape_fast(-1, 1) call was not detected as a hallucinatory API call. This is because the present attribute-chain analysis is limited. The resolver handles attribute chains right under imported names (like np.<attribute>), but does not make type inferences for return types of function calls (like np.array(rows>). Thus, it won't trace the returned object back to an ndarray and then check if reshape_fast exists. This restriction is the current area of the hallucination detector and is being taken into account in the limits discussion. In general, sample 2 shows that the detector can distinguish between fake dependencies, invalid API on valid modules, and expressions that need more runtime type information.

5.4.3 Sample 3 — Authentication Helper

Sample 3 is an example of a Python utility related to the authentication process that involves various security-critical actions. Hardcoded credentials, dynamic generation of SQL queries, running the user input via eval(), and an assert call to check for administrative privileges are among the actions performed by the code. As a result of the analysis, three vulnerabilities have been detected, yielding a risk rating of 74 out of 100 (High Risk). The most critical vulnerability is the usage of eval() on the user input as this action could potentially lead to arbitrary code execution. In addition, there is one High and one Low severity access control flaw in the sample.

```
API_KEY = "sk-live-bHdbXzy4kK0KPY4X6K7YzA5k3ttF4s8g"
SECURE_TOKEN = "mgp32rcectoyzL0i12127"

def get_ip_info(url, user_id):
    query = f"SELECT * FROM users WHERE id='{user_id}'"  # uses_f string
    return db_manager.query(query)

def create_user(user_input):
    return orm.create_user(user_input)

def check_admin(user):
    assert user.is_admin
    return True

def get_token_data(token):
    return jwt.decode(token, SECRET_KEY, algorithms=["HS256"])
```

Table (9) illustrates the security vulnerabilities present in Sample 3. The table highlights the source code line, risk score, CWE category, and finding description. It is evident that the risk score is highly dependent on the risk severity since, despite having only three findings, the presence of a CWE-95 (eval() function) vulnerability greatly enhances the risk score. The hard-coded API key belongs to CWE-798 category, whereas the use of assert for access control belongs to CWE-703.

 ISSN:2302-8149
e-ISSN:2540-9719

Table 9. Findings for Sample 3 (risk score 74/100, High Risk).

Line	Severity	CWE	Finding
9	Critical	CWE-95	Use of `eval()` on user-supplied input (arbitrary code execution)
1	High	CWE-798	Hardcoded credential in variable `API_KEY`
12	Low	CWE-703	`assert` used for an access-control check, which is stripped when Python runs with `-O`

With only three detected findings, Sample 3 scored the highest risk of all evaluated samples with a risk score of 74/100 and a High Risk classification. This finding is consistent with the desired behavior of the risk-scoring heuristic, where the severity of a finding is more important than the number of findings. In particular, it was determined that the usage of `eval()` with user supplied data is a **Critical-severity CWE-95 vulnerability**. The presence of Critical findings significantly contributes to the overall risk score as Critical findings receive the maximum point value and the heuristic algorithm can receive high scores very quickly when these findings are present. Two important flaws in the existing rules used to detect vulnerabilities are also highlighted in the results. The first one is that the SQL statement on line 5 which is built using string concatenation was not identified as a SQL injection vulnerability. The query is first stored in intermediate variable query and then sent to `db.execute()`. The current SQL-injection rule checks the argument passed directly to `.execute()` or `.executemany()`, and thus does not backtrack through arguments. This is a "false negative" due to the line-local analysis that this rule performs, not true data-flow analysis.

Secondly, no hardcoded credentials were identified for the variable SECRET_TOKEN. The naming-based secret detector uses a regular expression that matches identifiers which terminate with some keywords or phrases, like `secret` or `token`. The SECRET_TOKEN compound is not a perfect match to this pattern; that causes another coverage gap. The examples show that the detector is able to detect high impact exploits, like arbitrary code execution and hardcoded credentials, but that the detector's current rule-based approach is not sufficient to detect vulnerability where data-flow tracking or more flexible semantic interpretation of compound identifiers is required.

5.4.4 Sample 4 — Config / Cache Loader

Sample 4 depicts a Python utility for loading configurations and keeping a collection of items. This piece of code utilizes the pickle and yaml modules to unmartial configurations and provides a method for adding items to the collection. The security assessment has found three vulnerabilities, yielded a risk score of 73/100 and placed the sample into the category of High Risk. The most serious problem with this sample is represented by the usage of pickle.load() since this is a Critical-severity CWE-502 vulnerability due to the insecure deserialization process.

```
import pickle
import yaml

def load_config(path):
    with open(path, "rb") as f:
        return pickle.load(f)

def parse_yaml(text):
    return yaml.load(text)

def add_item(item, bucket=[]):
    bucket.append(item)
    return bucket
```

Table (10) shows the security issues found in Sample 4. The table contains information about the line of code where the problem was found, its severity, CWE code, and description. These findings contain one Critical CWE-502 issue due to pickle.load(), one High CWE-502 issue due to yaml.load() without SafeLoader, and one Informational CWE-1188 issue related to a mutable default

 ISSN:2302-8149
e-ISSN:2540-9719

argument bucket= []. As a result, the total risk score is 73/100, which corresponds to the High Risk class.

Table 10. Findings for Sample 4 (risk score 73/100, High Risk).

Line	Severity	CWE	Finding
6	Critical	CWE-502	Insecure deserialization via *pickle.load()*
9	High	CWE-502	Unsafe *yaml.load()* without an explicit *SafeLoader*
11	Info	CWE-1188	Mutable default argument (*bucket=[]*) in *add_item*

Both pickle and yaml are correctly resolved, real, installed modules, so no hallucinations are reported here; every finding is a genuine insecure-default pattern of the kind LLMs frequently reproduce because it mirrors common (but outdated or unsafe) tutorial code. The detector's rule for yaml.load correctly checks the call's keyword arguments for an explicit SafeLoader before flagging it, so a hypothetical yaml.load(text, Loader=yaml.SafeLoader) call would have been correctly suppressed had the sample used it.

5.4.5 Sample 5 — Token / Greeting Utility (Benign Baseline)

Sample 5 consists of a relatively simple utility written in Python, comprising of two functions – one that produces a token string and the other that formats a greeting message. The sample was deliberately chosen as an example of a benign program, which could be used to test whether the suggested auditor could differentiate between relatively normal programs and those having multiple vulnerabilities. In terms of vulnerabilities, only one vulnerability was found – Low-severity issue described by CWE-338, scoring 8 out of 100 and classified as Low Risk. The problem lies in the use of random.random() function for token production, which is not designed for producing cryptographically secure random numbers.

```
import random
import string

def generate_token(length=16):
    chars = string.ascii_letters + string.digits
    return "".join(random.random() > 0.5 and c or c for c in chars[:length])


def format_greeting(name):
    return f"Hello, {name}!"
```

Table (11) shows the sole vulnerability found in Sample 5. The table includes information about the source-code line, the severity level, the CWE category, and the explanation of the identified vulnerability. This vulnerability relates to the use of the random.random() function in the secure token generation process and is categorized as CWE-338 with Low severity. Thus, the score of 8/100 puts the sample in the Low-Risk category. The benign sample also serves as a good control for the potential problem of reporting too many vulnerabilities by the tool. In this case, there was only one actual vulnerability reported in relation to the insecure pseudo-random number generation and no hallucinations were reported at all. This means that the analysis does not categorize normal code as a very risky code simply due to the existence of a security-sensitive function in it.

Table 11. Findings for Sample 5 (risk score 8/100, Low Risk).

Line	Severity	CWE	Finding
6	Low	CWE-338	*random.random()* is not a cryptographically secure PRNG and should not be used to generate security-sensitive tokens

The purpose of including this sample is to establish a baseline; both functions in this example are mostly correct Python code, idiomatically written, except for the one error of generating a token using the random module as opposed to secrets. This program returns just one Low-severity detection

 ISSN:2302-8149
e-ISSN:2540-9719

and no hallucination, scoring a Low-Risk band. It is evident from this example that the tool is not an indiscriminate flagger of benign code. This is essential for practical usage.

5.5 Hallucination-Detector Introspection Statistics

The number of instances covered by the introspection mechanism of the hallucination detector in each sample is presented in Table (12). In contrast to the hallucinations detected at the end of the analysis process that are limited to only those issues reported in the evaluation process, these numbers refer to the number of imported modules and attribute-access expressions where the detector tried to verify the presence of corresponding Python objects through real-import and hasattr() introspection.

Table 12. Hallucination-detector introspection coverage per sample.

Sample	Imports checked	Imports resolved	Attrs. checked	Attrs. resolved
S1	2	2	5	2
S2	3	2	4	3
S3	0	0	2	0
S4	2	2	3	2
S5	2	2	4	3
Total	9	8	18	10

The detector examined 9 module paths for imports, and resolved 8 of those (a 90% success rate), with the subset of the five imported samples accounting for the remaining 89% of the import-resolution success rate. No unknown imports were remaining after Sample 2 was reported as a Hallucinated Package, except for fastjsonhelper. The result shows that the real-import mechanism can directly differentiate an available dependency from a fabricated and/or unavailable package. The results for the attributes are less complete. Of the 18 attribute-access expressions, 10 could be resolved by the detector, resulting in an overall attribute-resolution rate of about 56%. These resolved expressions could be mapped to real objects that were imported from files as concrete objects, and so it was possible to verify them with hasattr(). The other attribute expressions were not directly resolvable to imported roots in the scope of the static analysis at runtime, for the following reasons:

This is further illustrated in the per-sample results. Sample 1 imported 2 out of 2 attributes and 2 out of 2 imports were resolved, with 2 out of 5 attributes resolved for Sample 2. Sample 2 successfully imported 2 attributes out of 3 (the unimported one was the fake fastjsonhelper package) and 3 attributes out of 4. Sample 3 did not have any imports to verify but had 2 attribute expressions which were not resolvable. Samples 4 and 5 had 2/3 and 3/4 of their attributes resolved, respectively, while 2/2 of their sample were resolved when it comes to import resolution. The results suggest that the detector performs well at the module-import level (89%) while attribute verification is more limited due to what can be statically verified on the object on which the attribute is accessed. For example, attributes of local variables, function parameters or function return values can not necessarily be traced back to a specific imported object. The limitation is why the introspection coverage is less for attributes, and an important source of potential false negatives in the current implementation.

5.6 Comparative Interpretation of Risk Scores

The risk scores for the five samples were 62, 55, 74, 73 and 8 respectively giving an average score of 54.4. The results show that the framework is able to discriminate samples based on the severity of the identified weaknesses.

S3 and S4 were the top two scores due to having a Critical-severity Vulnerability each. S1 was classified as a Moderate Risk because the vulnerabilities were spread across the different severity levels, whereas the other two were classified as a Very Low Risk due to having only a few vulnerabilities at different severity levels. The S2 was also classified into the Moderate Risk category, due to its combined findings of a High severity TLS configuration weakness and two hallucination findings. Whereas S5 got Low Risk since its only finding was a Low-severity finding.

The comparison is useful for the intended use of the composite score as a prioritization tool. Should not, however, be read as a risk-to-security-calibration or a statistically tested measurement of

 ISSN:2302-8149
e-ISSN:2540-9719

security risk. It should summarize the results of the detection rules that have been implemented, and help the developers prioritize code review.

5.7 Discussion and Limitations

From the five-sample evaluation, it can be seen that the proposed system can classify representative sample instances of the defect classes. Not only does the evaluation check for security vulnerabilities, like command injection, insecure deserialization, weak cryptographic usage, unsafe programming defaults, hardcoded credentials and disabled TLS certificate verification, it also tests for hallucinated packages and missing attributes on legitimate Python packages. The findings also confirm the desired properties of the composite risk-scoring heuristic. For Critical-severity findings (S3 and S4), samples with more S4 or S3 findings had a higher risk score than samples with more S1 findings. This confirms that the scoring is not just based on the number of issues found but also the severity of the issues. However, the score for S5, intentionally added as a benign baseline, was the lowest of all – 8/100 – demonstrating the capacity of the tool to correctly not earmark unnecessarily high risk to relatively routine code. In spite of these findings, the evaluation finds some drawbacks on the current static and rule-based implementation. The first restriction is the lack of data-flow tracking. Both detectors focus on the analysis of individual AST nodes (e.g., calls, assignments, and attribute accesses) rather than on following values from one statement to another. In SQL-injection (S3), the dynamically built query is stored in an intermediate variable and then passed to execute(), as is the format of results in the case of the SQL-return example in S5. The rule used is not designed to trace the variable back to the original concatenation operation and so gives a false negative as the current rule only looks at the expression passed directly to the Call to Database.

The second constraint is related to the attribute detection within the hallucination detector. The resolver is able to check an attribute chain if the root identifier can be directly linked to an imported module. You can check requests.get_json_super_fast, for instance, because requests is directly imported. But expressions like np.array(rows).reshape_fast(...) need to know the type of the object returned by the previous function call to check the subsequent attribute, which is what the detector does. The current implementation does not implement this kind of type inference and for this reason such expressions are not verified. The third restriction is about the naming-based detection of hardcoded secrets. The existing rule is based on a regular expression over variable names and is based on an exact match of keywords or suffixes. Therefore, a compound identifier (like SECRET_TOKEN) can appear without being identified by a human reviewer, even though it is easily understood in its meaning. It is a limitation of only using lexical naming patterns for detecting credentials. These restrictions offer definite guidance on future directions. The most useful extensions are those that perform lightweight intra-procedural data-flow tracking, which enable the detection of potentially dangerous values carried across statements; type-inference-assisted attribute resolution, which allows for the validation of chained expressions of function-returned objects; and a wider (or fuzzy-matching) oracles for detecting secrets. These improvements might make detection coverage higher and maintain the lightweight and dependency-free nature of the existing implementation.

5.8 Limitations and Threats to Validity

The experimental results illustrate the feasibility of the proposed framework, but it is worth noting that the framework has a number of limitations.

1. First, all the programs in the evaluation set are only representative python programs. As such, the results are not generalizable to other programming tasks, LLM providers, model versions, or programming environments.
2. Second, the rule-based vulnerability detector is based on the syntactic patterns that are pre-defined. It doesn't offer full data-flow, control-flow or interprocedural analysis. This is shown in the S3 example, where the vulnerable query was kept in a temporary variable before being executed.
3. Third, complex chains of attributes and expressions of function return values are not fully supported in the hallucination detector. The reshape_fast() example given below is not solved, showing the need for better type inference and object tracking.
4. Fourth, unexpectedly compounded identifiers or naming patterns could lead to a failure to identify hardcoded credentials by naming based detection.Fourth, when naming based-

 ISSN:2302-8149
e-ISSN:2540-9719

detection occurs, it may fail if the developer has used an unexpected compound identifier or naming pattern. This is just one of the drawbacks of the SECRET_TOKEN finding.

5. Last but not least, the installed dependencies and configuration of the Python environment where the script is running can impact runtime introspection. You can import a module that can't be imported locally, but it still can be a valid package. Thus, the interpretation of the findings related to hallucinations should be based on the confidence level and not in all cases taken as final proof of hallucinations.

Moving forward the evaluation should use a larger, independently labelled dataset, multiple versions of LLM, other Python projects, and manual verification of detected and missed findings. This assessment would allow the calculation of precision, recall, F1-score, false positive rate, and false negative rate and, thus, provide an more in-depth evaluation of detection effectiveness.

6 Conclusions

In this study, we proposed a light-weight Python code auditing tool named LLM Code Auditor to detect some selected security vulnerabilities and possible hallucinations in Python code generated by a large language model (LLM). The proposed approach is a combination of runtime introspection of the available imported modules and the available referenced attributes with analysis based on python abstract syntax trees (AST) to find pre-defined insecure coding patterns. The tool also provides severity and a combined risk value to facilitate the interpretation and prioritization of findings detected. The five controlled Python code samples corresponding to the different programming scenarios were used for experimental evaluation. The findings found 14 vulnerability findings and 2 potential hallucination findings giving a total of 16 findings. The mean of the five samples was 54.4 points. The authentication helper (S3) and configuration/cache loader (S4) were the two components with the highest risk scores (74 and 73 respectively) mainly due to critical issues involving the use of `eval()` and unsafe deserialisation were found. The score for the token and greeting utility (S5) was 8, which is relatively less severe in terms of security issues. The results show that the tool is able to detect several pre-defined categories of insecure coding practices and potential inconsistencies in code generation in the samples it has been evaluated on.

The runtime introspection code inspected 9 modules, and resolved 8 (88.9%). It also analyzed 18 referenced attributes, of which 10 could be resolved, for a resolution rate of 55.6%. According to these measurements, the component can serve as a valuable source of dependency and attribute availability information. However, they do not indicate overall hallucination detection accuracy, but resolution coverage. Likewise, in the identified-vulnerability findings there is no guarantee that all these vulnerabilities are detected, but only that the found ones are detected, and that the operation of the rules is demonstrated. These results must be viewed in the context of some limitations. The first is that the evaluation was done on just five controlled samples with specific defects, so the results do not measure performance on a large, varied set of naturally generated LLM code. Second, the study did not provide ground truth independent labels and did not report standard metrics used for classification like precision, recall, F1-scores, and false-positive rate. As a result, the accuracy and generalization ability of the developed tool cannot be determined. Third, the AST-based analyzer takes a set of rules for detecting vulnerabilities, and it may not be able to detect vulnerabilities if they are implemented in a way that does not match the AST-based rules. The evaluation identified, for instance, an undetected credential-like identifier as well as an undetected missing SQL injection detection when the query has been assigned to an intermediate variable before actual execution and an unresolved method pattern that wasn't covered by checks implemented. In addition, runtime module introspection can be impacted by variations in Python environments and module initialization.

Future work includes a larger sample of Python programs produced by varying LLM models to increase the evaluation sample, and a standard set of metrics for detection, with ground-truth annotations independently verified. The analysis engine will also be expanded to include support for data-flow and interprocedural analysis, enhanced identifier and API resolution, and more vulnerability patterns. More tests on different Python versions and execution environments will be required to confirm the repeatability and applicability to practice. Finally, our experiments show that using runtime introspection with a static analysis approach based on AST can be an initial step in discovering certain hallucinated code references and programming patterns in Python samples.

 ISSN:2302-8149
e-ISSN:2540-9719

However, the current implementation is limited in its scope of testing and rule-based so it is not a complete or validated security tool, but rather a preliminary code auditing approach.

References

[1] B. Radenkovic, S. Prokhorov, M. Radenkovic, and A. Labus, *Application of Large Language Models in Software Development: Review of the Current State and Development Perspectives*. 2025.

[2] A. Patel, K. Z. Sultana, and B. Samanthula, *A Comparative Analysis between AI Generated Code and Human Written Code: A Preliminary Study*. 2024.

[3] C. Gao *et al.*, *A Systematic Literature Review of Code Hallucinations in LLMs: Characterization, Mitigation Methods, Challenges, and Future Directions for Reliable AI*. 2025.

[4] Y. Lee, J. Song, D. Kim, J. Kim, M. Kim, and J. Nam, *Hallucination by Code Generation LLMs: Taxonomy, Benchmarks, Mitigation, and Challenges*. 2025.

[5] M. Kharma, S. Choi, M. .Alkhanafseh, and D. Mohaisen, *Security and Quality in LLM-Generated Code: A Multi-Language, Multi-Model Analysis*. 2025.

[6] S. Ben Yaala and R. Bouallegue, "*Vulnerability Detection in Large Language Models: Addressing Security Concerns*," *J. Cybersecurity Priv.*, Vol. 5, No. 3, 2025.

[7] F. Liu *et al.*, "*Beyond Functional Correctness: Exploring Hallucinations in LLM-Generated Code*," *IEEE Trans. Softw. Eng.*, Vol. 52, pp. 1037–1055, Jan. 2026.

[8] M. Iavich, T. Kuchukhidze, and A. Lopata, "*Static Analysis Techniques for Embedded, Cyber-Physical, and Electronic Software Systems: A Comprehensive Survey*," *Electronics*, Vol. 15, No. 5, 2026.

[9] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "*Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions*," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 754–768.

[10] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "*Do Users Write more Insecure Code with AI Assistants* ?," No. 8, 2023.

[11] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, "*Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants*," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 2205–2222.

[12] J. Spracklen, R. Wijewickrama, A. H. M. N. Sakib, A. Maiti, B. Viswanath, and M. Jadliwala, "*We have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating {LLMs}*," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 3687–3706.

[13] A. Raj and S. Sahu, "*Names can Hurt : Spotting Slopsquatting Risks caused by Package Name Hallucinations in Local Coding LLMs*," 2026.

[14] S. G. Morkonda, M. Selim, and H. Assal, "*Security of LLM-Generated Code : A Comparative Analysis .*," *arXiv:2605.23091*, 2025.

[15] S. S. Aldosari and L. S. Aldawsari, "*Securing LLM Code Generation: Leveraging Prompt Engineering to Mitigate Vulnerabilities Across Models and Languages*," *SCI. Comput. Program.*, Vol. 251, p. 103446, 2026.

[16] C. Negri-Ribalta, R. Geraud-Stewart, A. Sergeeva, and G. Lenzini, "*A Systematic Literature Review on the Impact of AI Models on the Security of Code Generation*," *Front. Big Data*, Vol. Volume 7-2024, 2024.

[17] Z. Li, D. Zou, S. Xu, X. Ou, and H. Jin, "*VulDeePecker : A Deep Learning-based System for Vulnerability Detection*," *arXiv:1801.01681*, No. February, 2018.

[18] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "*SySeVR: A Framework for using Deep Learning to Detect Software Vulnerabilities*," *IEEE Trans. Dependable Secur. Comput.*, Vol. 19, No. 4, pp. 2244–2258, 2022.

[19] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "*Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks*," in *Advances in Neural Information Processing Systems*, 2019, Vol. 32.

[20] M. Fu and C. Tantithamthavorn, "*LineVul: A Transformer-based Line-Level Vulnerability*

Prediction," in *Proceedings - The 2022 Mining Software Repositories Conference, MSR 2022*, 2022, pp. 608–620.

[21] W. P. Konan, A. Da, and S. College, "*WildCode Revisited : A Comprehensive Empirical Study on the Security of LLM-Generated Code*," *arXiv:2512.04259*, pp. 1–38, 2026.

[22] V. Belozerov, P. J. Barclay, and A. Sami, "*Secure Coding with AI – from Detection to Repair*," *Empir. Softw. Eng.*, Vol. 31, No. 4, p. 93, 2026.

[23] D. Cotroneo, C. Improta, and P. Liguori, "*Human-Written vs. AI-Generated Code: A Large-Scale Study of Defects, Vulnerabilities, and Complexity*," in *2025 IEEE 36th International Symposium on Software Reliability Engineering (ISSRE)*, 2025, pp. 252–263.