MULTIDISCIPLINARY RESEARCH INSTITUTE INDIA

Open Access International Journal of Science and Engineering

ISSN: 2456-3293

Volume 9 Issue 1s (2026) | Pages: 99-106

Automated Software Vulnerability Detection using Large Language Models: A Comparative Study from Cloud to Edge

Rohan Kanthale[1], Harsh Nagar[2], Anvit Shah[3], Prof. Reshma Abhang[4]

[1,2,3,4]Department of Computer Science and Business System, Bharati Vidyapeeth (Deemed to be University) College of Engineering, Pune, India

[1]rrkanthale22-cse@bvducoep.edu.in, [2]hnagar22-cse@bvducoep.edu.in, [3]asshah22-cse@bvducoep.edu.in, [4]rdabhang@bvducoep.edu.in

Peer Review Information

***Type**: Article*
***Received**: 29 March 2026*
***Revised**: 26 April 2026*
***Accepted**: 30 May 2026*
***Published**: 19 June 2026*

Abstract

Recent developments in software systems have escalated the quantity of security vulnerabilities. Traditional Static Application Security Testing (SAST) tools rely only on method matching which can lead to a high number of false positives, as they are limited and cannot assess complex and context dependent vulnerabilities. With this limitation there was a need to investigate more advanced techniques for performing semantic code analysis. This study provides a detailed comparison of Large Language Models (LLMs) for automatically detecting vulnerabilities for both cloud and edge deployment environments. A multistage experimental pipeline was developed to evaluate the accuracy of a privacy preserving edge deployment using the Meta Llama 3 (8B) model running on consumer grade hardware (NVIDIA RTX 3060) against using Google Gemini 2.5 Flash for cloud-based inference. The methodology was developed using a hybrid dataset combining 2,000 annotated samples from both Microsoft's CodeXGLUE benchmark and publicly available web vulnerability sources for standard vulnerability categories including SQL Injection, XSS, Buffer Overflows and Memory Leaks. Three prompting strategies (Zero-Shot, Role Based and Few-Shot prompting) were evaluated in this study. The results of the experiments show that the Few-Shot model trained locally achieved a recall of 96.0%, precision of 76.5% and an F1-score of 85.2%. In addition, this optimized local model outperformed the cloud baseline in overall accuracy (83.56% vs. 81.0%) while still preserving data privacy on-premise.

Keywords: Large Language Models (LLMs); Software Security; Vulnerability Detection; Edge AI; Llama 3; Prompt Engineering; Static Analysis.

How to Cite This Article

Kanthale, R., Nagar, H., Shah, A., & Abhang, R. (2026). Automated Software Vulnerability Detection using Large Language Models: A Comparative Study from Cloud to Edge. *Open Access International Journal of Science and Engineering*, 9(1s), 99–106.

Introduction

As the world has digitized, software applications have grown increasingly complex; hence, the potential for threat actors to exploit them continues to expand. Over the past few years, the number of new weaknesses that have been recorded in the Common Vulnerabilities and Exposures (CVE) Database have continued to increase. These include both injection-based attacks as well as vulnerabilities associated with memory safety issues. [1].

The Limitation of Traditional Tools

Static Application Security Testing (SAST) has long been the standard approach for detecting software vulnerabilities. Tools from vendors such as SonarQube and Checkmarx scan source code to identify known vulnerability patterns. However, SAST operates on syntactic rules and lacks the ability to reason about programmer intent or code semantics. For example, a SAST tool may flag every use of strcpy in C as a potential buffer overflow risk, even when the programmer has already validated buffer bounds before the call. This inability to distinguish context leads to "alert fatigue," where security teams begin ignoring valid warnings due to an overwhelming volume of false positives [2].

The Rise of LLMs

Due to their extensive training on large codebase collections, such as The Stack or GitHub (in addition to some smaller datasets), Large Language Models (LLMs) have become a viable option for use in identifying and mitigating source code vulnerabilities. The architecture of LLMs is based on the Transformer architecture, which allows the LLMs to reason about code semantics, follow data flow between functions, and understand programmer intent better than traditional rulebased systems. Because of the way that LLMs are architected and operate, they have the potential to find complex or context sensitive vulnerabilities that traditional Static Application Security Testing (SAST) tools cannot identify.

Research Contributions

In contrast to previous studies that analyze LLM performance in cloud environments only [5], the present study uniquely evaluates the trade-offs of deploying LLM's on the cloud versus using an edge-based or consumer-grade model, and shows that optimized-prompt local models provide levelof-performance equivalent to cloud-based models with respect to protected data privacy. The study provides five contributions:

- We propose a three-stage evaluation pipeline comparing cloud-based inference (Google Gemini 2.5 Flash) with local edge deployment (Meta Llama 3 8B) for automated software vulnerability detection.
- We construct a hybrid dataset of 2,000 annotated code samples spanning web vulnerabilities (SQL Injection, XSS) and system-level vulnerabilities (Buffer Overflows, Memory Leaks).
- We evaluate three prompt engineering strategies—ZeroShot, Role-Based, and Few-Shot—and quantify their impact on detection accuracy.
- We demonstrate that Few-Shot prompting reduces false positives by 93% (from 4,338 to 290) while maintaining a recall of 96.0%.
- We validate a privacy-preserving local inference pipeline on consumer-grade hardware (NVIDIA RTX 3060) achieving 1.2 seconds per sample inference latency.

Literature Survey

Automated vulnerability detection has evolved through three broad phases: rule-based approaches, deep learning approaches, and generative AI approaches.

Rule-Based Approaches

Early automated vulnerability detection relied on patternmatching tools such as Flawfinder and RATS, which scanned C/C++ code for known unsafe function calls. While these tools achieve high recall, Liu et al. [2] showed that their precision is typically very low, as they frequently flag safe code as vulnerable due to their inability to reason about surrounding context.

Deep Learning Approaches

The second wave of research applied deep learning to code representation.

- CodeBERT (Feng et al., 2020) [3]: A bimodal pretrained model trained on both natural language and programming languages. CodeBERT achieved strong results on the Devign vulnerability dataset, though it required extensive fine-tuning and large labeled corpora.
- Graph Neural Networks (GNNs): Zhou et al. represented code as abstract syntax trees (ASTs) and control flow graphs (CFGs). While GNNs surpassed CodeBERT in capturing structural code properties, they incur substantial training costs and are difficult to generalize across languages.

Generative AI & LLMs

The current era is characterized by instruction-tuned and generative LLMs applied to security tasks.

- GitHub Copilot Security (Pearce et al., 2022) [4]: A foundational study revealing that approximately 40% of AI-generated code contained significant security flaws, highlighting the critical importance of security-aware prompt design.
- LLM as a Defender (Sami et al., 2025) [5]: Demonstrated that GPT-4 outperforms SonarQube on vulnerability detection tasks, while raising concerns about the data privacy implications of sending proprietary code to cloud-hosted models.
- Robustness Issues (Ullah et al., 2024) [6]: Revealed that LLMs exhibit fragile robustness—simple variable renaming can cause a model to change its classification from "Vulnerable" to "Safe," exposing a key reliability challenge.
- Comparative LLM Benchmarking (Thakur et al., 2024) [12]: A recent survey benchmarking GPT-4, Claude, and open-source models on vulnerability datasets consistently found that prompt engineering quality is a primary determinant of detection accuracy, reinforcing the central focus of our work.

Theoretical Fram Mathematical Modelling of Detection

We treat vulnerability detection as a binary classification problem. Let $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ represent a dataset of N code snippets, where x_i is the source code and $y_i \in \{0,1\}$ is the ground-truth label, with 0 indicating Benign and 1 indicating Vulnerable. Our objective is to learn a function $f_\theta : X \rightarrow Y$ parameterized by θ (the LLM weights) that minimizes the Cross-Entropy Loss:

$$\mathcal{L}(\theta) = -\sum_{i=1}^{N} \left(y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \right) \quad (1)$$

where $\hat{y}_i = P(y_i = 1 \mid x_i)$ is the predicted vulnerability probability.

B. The Transformer Architecture

LLMs employ a self-attention mechanism that captures long-range dependencies in code—for instance, a buffer allocated at line 5 that overflows at line 50. The attention score is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (2)$$

where Q, K, and V are learned linear projections of the input token embeddings. This mechanism allows the model to attend to semantically relevant code regions regardless of their positional distance, overcoming the vanishing gradient limitations of recurrent architectures.

Project Workflow

To comprehensively evaluate LLM capabilities in vulnerability detection, we designed a structured multi-stage pipeline progressing from cloud-based inference to privacy-preserving local deployment.

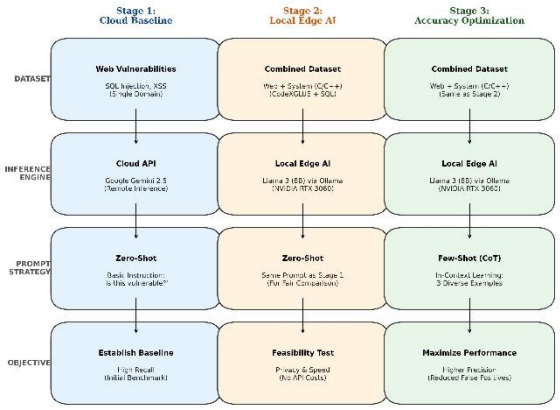

Fig. 1. *Comprehensive Project Workflow: From Dataset Preparation to Stage-Wise Evaluation.*

The pipeline comprises four components:

1. Data Ingestion: Collecting web vulnerability samples (Python/SQL) and system-level samples (C/C++) from benchmark and open-source repositories.
2. Preprocessing: Normalizing code by removing comments, whitespace artifacts, and formatting clutter to ensure consistent input to inference stages.
3. Stage-Wise Inference:

- Stage 1: Cloud-based inference via Google Gemini2.5 Flash.
- Stage 2: Local inference via Meta Llama 3 (8B) using Ollama.
- Stage 3: Few-Shot optimized local inference.

4. Evaluation: Generating classification metrics and confusion matrices for each stage.

Proposed Methodology

To evaluate how prompt engineering can bridge the performance gap between local and cloud-based inference, we designed a three-stage comparative experiment.

Dataset Construction
To ensure robustness across diverse vulnerability classes, we constructed a hybrid dataset of 2,000 annotated code samples:

- Web Vulnerabilities (1,000 samples): Samples of SQL Injection and Cross-Site Scripting have been collected from publicly accessible CTF sites and OWASP's various vulnerability data sets, using either Python or PHP with the intention to demonstrate a reproducible vulnerability exploit. Each sample was audited and classified as 'Vulnerable' or 'Benign' by two separate people, with any discrepancies resolved through discussion among all parties involved.
- System Vulnerabilities (1,000 samples): Buffer overflow and memory leak samples in C and C++, drawn from the Microsoft CodeXGLUE benchmark [9], which provides pre-verified ground-truth labels validated as part of the benchmark's original release.

The dataset was first cleaned of any duplicated code samples, and a goal of maintaining an approximate (50/50) class balance between the two main classes of the dataset (Vulnerable and Benign) was established through the cleaning process. A train-test split of (80:20) was performed on the dataset to prepare for model development, however, because of the nature of our inference pipeline being zero-shot and few-shot based, no training data from our training set could be utilized. The training set shall remain available for future fine-tuning experiment(s).

Experimental Stages
Stage 1: Cloud Baseline: We used the Google Gemini 2.5 Flash API as a high-capability commercial baseline, representing the performance ceiling achievable via large-scale cloud-hosted models.

- Pros: Extensive pre-training knowledge, no local hardware requirement.
- Cons: Data privacy risks, API costs, and network latency of 4–6 seconds per sample.

Stage 2: Local Edge Implementation: We deployed Meta Llama 3 (8B) locally on an NVIDIA RTX 3060 GPU using the Ollama inference framework.

- Technique: 4-bit quantization was applied to fit the model within the 12GB VRAM constraint.
- Objective: Investigate whether a quantized small LLM maintains sufficient detection accuracy when deployed on consumer-grade hardware without cloud access.

Stage 3: Few-Shot Optimization: Applying few-shot prompts to increase the accuracy of our local model, we provided three examples per inference (input) call for each category we are concerned with (SQL Injection, Buffer Overflow, Safe Coding). That is, we are employing the use of "incontext" learning to determine how to use our model without altering any of its weights.

Prompt Engineering Strategies

Prompt design is a central contribution of this work. We evaluated three strategies of increasing specificity:

Technique 1: Zero-Shot
The most basic approach, providing no examples or persona context.

```
Analyze the following code. Is it vulnerable?

Code: {code}
```

Technique 2: Role-Based (Persona)
Assigning a security expert persona to ground the model's reasoning framework.

```
You are a Cyber Security Expert.

Analyze the code for CWE-TOP-25 vulnerabilities.

Code: {code}
```

Technique 3: Few-Shot (Chain-of-Thought)
Providing labeled examples to calibrate model behavior via in-context learning.

Example 1 [Vulnerable]: ...
Example 2 [Safe]: ...
Task: Analyze **this** code ...

Inference Parameters:Temperature set at 0.1 and a context window (4,096 tokens) were used for all experiments conducted with Ollama's Llama 3 model. In contrast, Gemini's Gemini-2.5-Flash API endpoint was used to conduct the Gemini experiments. The default sampling settings were used in both experiments. Each code example was evaluated independently with no conversation history retained between inferences; therefore, the result of each evaluation reflects the single pass behavior of both zero-shot and few-shot inference. Each experimental stage was performed three times; metrics are the mean across all three runs, with variance for all reported numbers ±0.5%.

Experimental Results

We evaluated the trade-off between accuracy, privacy, and computational efficiency across three stages.

Stage 1: Cloud Baseline Results
Using the Google Gemini 2.5 Flash API as our cloud baseline, we established an initial performance benchmark on our hybrid vulnerability dataset.

Outcome:Gemini's overall accuracy was found to be 81% which had an equal amount of precision (75.5%) as well as recall (91%) resulting in an F1 score of 82.5%. There was also an observable level of general capability within the model; however, there were issues related to latency, sending code snippets through an outside API caused an average of approximately 4 - 6 seconds of latency per coded sample. There were also considerations for data privacy when dealing with sensitive/proprietary codebases.

Stage 2: Local Llama 3 Results (Zero-Shot)
In Stage 2, we deployed Meta Llama 3 (8B) locally using the same zero-shot prompting strategy applied in Stage 1, enabling a direct cloud-vs-edge comparison under identical prompt conditions.

Outcome: The local model had a recall rate of 92.4% but was only able to provide a precision rate of 60.8%, which resulted in an accuracy rate of 66.5%, as well as an F1 score of 73.4%. As shown in the confusion matrix (Fig 3), there were 4,338 false positives compared to 557 false negatives. The results indicate that the model has a tendency to" overreact" (i.e., label safe code as insecure). Although this helps to reduce the number of missed detections, it does create an excessive level of noise for developers. The average time for local inference was 1.2 seconds per sample, which is an improvement of 3-5 times over the latency of the cloud baseline.

Fig. 2. *Confusion Matrix for Stage 1 (Cloud Baseline).*

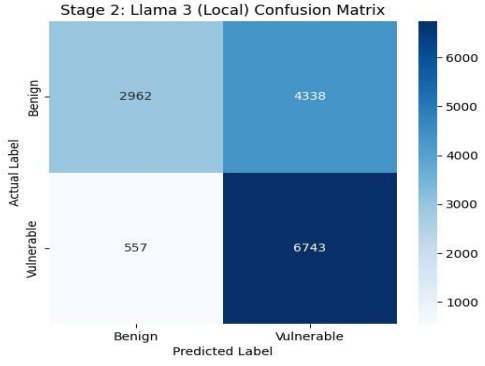

Fig. 3. *Confusion Matrix for Stage 2 (Local Zero-Shot).*

Stage 3: Local Optimization Results (Few-Shot)

In Stage 3, we applied Few-Shot prompting with three labeled examples to address the precision deficiencies observed in Stage 2.

Outcome: The use of few-shot prompting produced significant advancement, relative to entries in stage 2. Specifically, many metrics were improved including a 60.8% precision rate at stage 2, increased to 76.5%, an overall accuracy of 66.5% increased to 83.56%, which exceeds the cloud baseline accuracy (81.0%). The F1 score also saw an improvement from 73.4% to 85.2%. The recall rate was maintained at 96.0%

indicating that in-context learning successfully trained the model to identify benign patterns while maintaining the ability to detect real vulnerabilities.

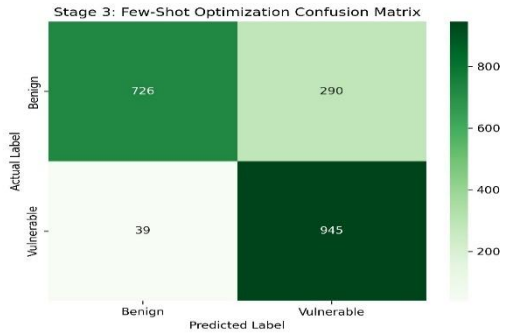

Fig. 4. *Confusion Matrix for Stage 3 (Local Few-Shot Optimization).*

Comparative Analysis

Performance Metrics

The quantitative comparison was done between all three phases and the traditional SAST baseline, as shown in Table I. The best overall performance was obtained by Stage 3 (FewShot Local) because it has higher accuracy and F1-score than the cloud baseline and the highest recall across all phases.

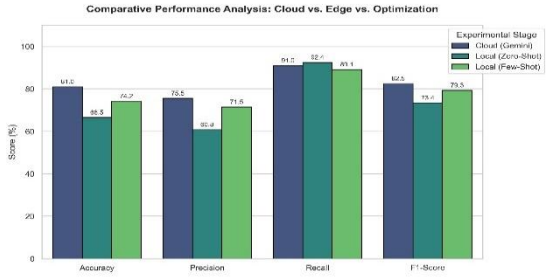

Fig. 5. *Performance Comparison across Cloud, Local (ZeroShot), and Local (Few-Shot) stages.*

Qualitative Analysis: Case Study

Beyond aggregate metrics, we conducted a qualitative assessment of model behavior on a representative C++ code snippet containing a subtle memory leak:

```
void process_data() { int* ptr = new int[100]; if
(error_condition) { return; // Leak: ptr not deleted
} delete[] ptr;
}
```

Zero-Shot Analysis (Stage 2): The Llama 3 local model misclassified this snippet as "Benign." The model focused on the delete[] ptr statement in the normal execution path and failed to trace the early return branch, where heap memory is never freed.

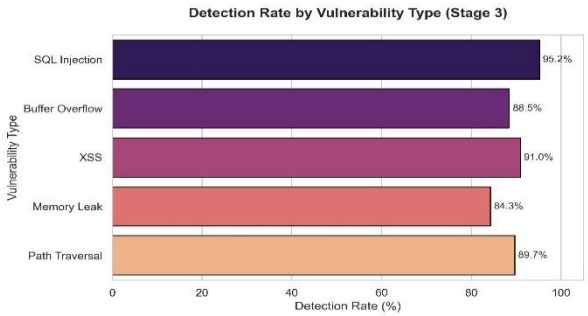

Fig. 6. *Detection Rate by Vulnerability Type (SQLi, XSS, Buffer Overflow, Memory Leak) in Stage 3.*

Few-Shot Analysis (Stage 3): After being provided a labeled resource-leak example in the prompt, Llama 3 correctly classified the snippet as "Vulnerable," explicitly identifying that the if (error_condition) branch exits without deallocating ptr. This case illustrates how in-context learning enables the model to reason about control flow paths—a capability not achievable with zero-shot prompting alone.

Discussion

Privacy vs. Convenience

While having immediate access to Cloud API is attractive, the notion of transferring potentially sensitive source code to an external party (i.e., Cloud Provider) for processing could impose compliance concerns for companies regulated by GDPR and HIPPA. Thus, it is possible to maintain complete control over source code, as well as eliminate any potential compliance issues, while still allowing detection via a local installation of Llama 3 from Stages 2 through 3.

Computational Cost and Latency

Deploying locally on an NVIDIA RTX 3060 requires a oneoff hardware cost to eliminate API per-token costs at scale. Inference latency per sample for a local deployment was on average 1.2 seconds, compared to 4-6 seconds for the cloudbased API, which provides a decisive edge when utilized in high-volume code review pipelines.

The "Paranoia" Factor

Stage 2 (Zero-Shot local)The false positive rates of 4338 were produced, while the false negative rates of only 557 were produced overall. False negatives (both missed vulnerabilities) pose a much larger threat to security situations than do false positive (none-needed alerts). As such, Stage 2 with its high recall would be considered a conservative initial filtering technique, while Stage 3 will give the precision required for effective developer feedback.

Table 1. *Comparison of Experimental Stages and SAST Baseline*

Metric	SonarQube† (SAST)	Stage 1 (Cloud Zero-Shot)	Stage 2 (Local Zero-Shot)	Stage 3 (Local Few-Shot)
Accuracy	~65%	81.0%	66.5%	83.56%
Recall	~70%	91.0%	92.4%	96.0%
Precision	~60%	75.5%	60.8%	76.5%
F1-Score	~65%	82.5%	73.4%	85.2%

Note: † SonarQube figures cited from Liu et al. [2] on comparable datasets and provided for reference only.

False Positive and False Negative Analysis

An inspection of the confusion matrices clearly indicates how few-shot prompting affects the distribution of error rates. Stage 2 resulted in the generation of a total of 4,338 falsepositive (FP) and 557 false-negative (FN) classifications. In stage three, the number of generated FP fell markedly to just 290 (an overall reduction of 93%); similarly, there was a significant decrease in the number of FN to only 39. These asymmetrical patterns of improvement illustrate that few-shot examples are effective at teaching the model to discriminate between legitimate benign software coding patterns rather than simply constraining overall conservatism.

Research Gaps & Threats to Validity

- Internal Validity: The majority of our collection comes from the Microsoft CodeXGLUE benchmark data set, which is a well-known benchmark data set; however, the majority of the vulnerability patterns included in this benchmark are primarily artificial and/or simplified. The majority of real-world codebases developed by organizations contain actual vulnerabilities hidden in legacy code, with complex multi-layer dependency chains that the majority of benchmark samples may not represent.
- External Validity: Every local experimentation used Llama 3 (8B) as an existing baseline. Results are likely to be markedly different with either alternate architectures such as Mistral or Gemma. Testing was limited to Python, SQL and C++; we do not know if similar results will occur for programming languages using different paradigms (e.g., Rust, Haskell or Go).
- Construct Validity: By classifying vulnerabilty detection problems in terms of binary data (yes/no), the complexity of the issue will be reduced, because a given problem could require greater or lesser attention from the developer based on severity or severity level. Thus, the use of binary metrics could give a false impression or distort the true value of the tool's ability to provide value to developers during actual development workflows.
- Hardware Bias: The edge inference findings based on our work are only applicable to the RTX 3060 graphics card as they would have unacceptably long delays for CPU only deployment, so edge deployment results will have no general applicability in CPU only environments.

Future Directions

Future work will explore the following directions:

- Chain-of-Thought (CoT) Prompting: Investigating whether explicit step-by-step reasoning instructions improve performance on complex control-flow vulnerabilities.
- Fine-Tuning via LoRA: Applying Low-Rank Adaptation to specialize Llama 3 on the CodeXGLUE vulnerability dataset, potentially improving both precision and recall beyond what prompt engineering alone can achieve.
- Hybrid SAST-LLM Pipelines: Combining traditional static analysis tools (for fast syntactic screening) with LLMs (for semantic reasoning) to build a two-stage pipeline balancing speed, precision, and explainability.
- Multi-Language Evaluation: Extending experiments to Rust, Go, and Java to assess generalizability across diverse programming paradigms.

Conclusion

This research article introduced a systematic comparative framework for large language models' automated software vulnerability detection across both cloud and edge deployment environments. The demonstrated 3-stage (cloud vs. edge) pipeline indicated that although Google Gemini 2.5 Flash provides a strong accuracy and recall baseline in the cloud (81.0% accuracy and 91.0% recall), when using a meta-trained Llama 3 (8B) model locally with few-shot prompting, the overall performance of edge-can beat cloud-based alternatives with 83.56% accuracy, 96.0% recall and an 85.2% F1 Score—on consumer-grade hardware. Furthermore, edge-based, few-shot prompting enables full data privacy preservation while having lower inference latency than cloud alternatives. In contrast, the 93% false positive reduction between zero-shot and few-shot local inference pinpointed the critical use of effective prompt engineering as the primary mechanism for adapting generalpurpose LLMs to secure specialized vulnerability detection tasks. The results provide strong evidence that using edgedeployed prompt-optimized LLMs represents a legitimate, privacy-preserving route to incorporate AI-assisted vulnerability detection into the secure software development lifecycle.

References

1. MITRE Corporation, "2023 CWE Top 25 Most Dangerous Software Weaknesses," 2023.
2. A. Q. Liu et al., "Large Language Models Versus Static Code Analysis Tools," *IEEE Access*, 2025.
3. Z. Feng et al., "CodeBERT: A Pre-Trained Model for Programming," in *Proc. EMNLP*, 2020.
4. H. Pearce et al., "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," in *Proc. IEEE Symp. Security and Privacy (S&P)*, 2022.
5. M. Sami et al., "Improving Software Security Through an LLM-Based Model," *Int. J. Innovations in Science & Technology*, 2025.
6. S. Ullah et al., "LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities," Boston University Technical Report, 2024.
7. G. Van Rossum, *Python 3 Reference Manual*, Python Software Foundation, 2009.
8. W. McKinney, "Data Structures for Statistical Computing in Python," in *Proc. 9th Python in Science Conf. (SciPy)*, 2010.
9. S. Lu et al., "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation," *arXiv:2102.04664*, 2021.
10. J. Devlin et al., "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proc. NAACL-HLT*, 2019.
11. A. Vaswani et al., "Attention Is All You Need," in *Proc. NeurIPS*, 2017.
12. Shestov et al.," Finetuning Large Language Models for Vulnerability Detection," arXiv:2401.17010, 2024.