

# An AI-Driven Secure Interpreter for Adaptive and Security-Aware Program Execution

**Zeeshan Ahmed<sup>1</sup>, Shivangini Bihari<sup>2</sup>, Anu Priya<sup>3</sup>**

<sup>1</sup>Student, Department of CSE, ITER, Siksha 'O' Anusandhan University, Bhubaneswar, Odisha, India.

<sup>2</sup>Assistant Professor, Faculty of CSE & IT, Jharkhand Rai University, Ranchi, Jharkhand, India.

<sup>3</sup>Assistant Professor, Amity Institute of Information Technology, Amity University, Patna, Bihar, India.

**To Cite this Article:** Zeeshan Ahmed<sup>1</sup>, Shivangini Bihari<sup>2</sup>, Anu Priya<sup>3</sup>, "An AI-Driven Secure Interpreter for Adaptive and Security-Aware Program Execution", Indian Journal of Computer Science and Technology, Volume 05, Issue 02 (May-August 2026), PP: 1087-1096.



Copyright: ©2026 This is an open access journal, and articles are distributed under the terms of the [Creative Commons Attribution License](#); Which Permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

**Abstract:** Traditional programming interpreters are primarily designed to perform lexical analysis, syntax analysis, semantic processing, and program execution according to predefined programming-language rules. Although such interpreters provide reliable and deterministic execution, they generally have limited capabilities for contextual code understanding, adaptive optimization, intelligent assistance, and continuous security analysis. Recent advances in artificial intelligence (AI), large language models (LLMs), machine learning (ML), and cloud computing provide an opportunity to transform conventional interpreters into intelligent execution environments. However, integrating AI directly into an interpreter introduces additional concerns related to malicious code execution, vulnerable AI-generated code, prompt injection, data exposure, model manipulation, privacy, and computational overhead.

This paper proposes an AI-Powered Secure Intelligent Interpreter (AISII) that integrates conventional interpretation mechanisms with AI-based code understanding, vulnerability analysis, runtime behavioral monitoring, anomaly detection, secure sandboxing, adaptive policy enforcement, and cloud-assisted intelligence. The proposed architecture follows a modular, defense-in-depth approach in which source code is analyzed through lexical, syntactic, semantic, static-security, and AI-based assessment stages before execution. A runtime monitoring layer continuously evaluates execution behavior and dynamically updates a risk score. Depending on the estimated risk, the security policy engine can permit, restrict, isolate, or terminate execution. The framework further incorporates local, edge, and cloud intelligence to balance privacy, latency, and computational scalability.

A comparative analysis positions the proposed architecture against conventional interpreters, AI-assisted programming tools, static analysis systems, and autonomous software-engineering agents. An experimental methodology is also proposed using code-generation benchmarks, vulnerability datasets, benign/malicious execution workloads, and real-world software-engineering tasks. Evaluation metrics include execution latency, resource utilization, vulnerability-detection precision, recall, F1-score, false-positive rate, and containment effectiveness. The paper concludes with an implementation roadmap and discusses federated learning, edge AI, autonomous programming, and quantum computing as future research directions. The proposed framework provides a foundation for developing interpreters that are not merely execution engines but adaptive, intelligent, and security-aware computing environments.

**Key Word:** Artificial Intelligence, Intelligent Interpreter, Large Language Models, Secure Code Execution, Cybersecurity, Vulnerability Detection, Runtime Monitoring, Cloud Computing, Adaptive Learning.

## I. INTRODUCTION

Programming interpreters have traditionally provided a mechanism for executing high-level programming instructions without requiring the entire source program to be translated into machine code in advance. A conventional interpreter generally performs a sequence of operations involving lexical analysis, parsing, semantic analysis, intermediate representation, and runtime execution. These mechanisms have been fundamental to programming environments because they provide flexibility, portability, interactive execution, and rapid development.

However, the requirements of modern software environments are substantially broader than simple program execution. Contemporary programming environments increasingly incorporate artificial intelligence to support code generation, code completion, debugging, documentation, refactoring, vulnerability identification, and software maintenance. Large language models have demonstrated capabilities across multiple software-engineering tasks, including coding, program repair, documentation, refactoring, and analysis (Fan et al., 2023, Hou et al., 2024).

The development of code-oriented language models has further accelerated this transformation. Models such as Code BERT, CodeT5, GraphCodeBERT, Code Llama, StarCoder, and CodeT5+ demonstrate that source code can be represented and processed using architectures originally developed for natural-language processing. Code BERT, for example, provides joint representations of programming and natural languages, while GraphCodeBERT explicitly incorporates data-flow information into code representation (Feng et al., 2020; Guo et al., 2021).

More recent systems have moved beyond code completion toward repository-level and autonomous software engineering. Repo Coder incorporates repository-level retrieval into code completion, while SWE-bench evaluates language models on real software-engineering issues rather than isolated programming exercises (Zhang et al., 2023; Jimenez et al., 2024). Cloud security

mechanisms can be integrated into interpreters to detect potential threats and protect code execution in cloud-based environments (Alam & Singh, 2015).

A conventional interpreter generally determines whether a program is syntactically and semantically valid. It does not necessarily determine whether the program is malicious, whether its behavior is anomalous, whether an AI-generated instruction contains security vulnerability, or whether execution should be dynamically restricted according to contextual risk.

This limitation becomes particularly important when AI-generated code is executed automatically. Research has shown that LLM-generated code can contain bugs and security weaknesses, even when the code appears syntactically correct. Empirical studies have therefore emphasized the need for security assessment and verification of AI-generated programs (Black et al., 2024; Dozono et al., 2024; Li et al., 2024).

The problem is therefore not simply how to make an interpreter more intelligent. The more important question is how to make it intelligent without making execution less trust worthy. The proposed AI-Powered Secure Intelligent Interpreter (AISII) addresses this challenge by combining five capabilities:

- a) Conventional program interpretation
- b) AI-based code and intent understanding
- c) Static and dynamic security analysis
- d) Adaptive risk-based execution control
- e) Local/edge/cloud-assisted intelligence

The proposed framework follows a defense-in-depth approach rather than allowing an AI model to directly execute arbitrary commands, AI-generated or user-supplied instructions pass through multiple verification and security layers. The architecture is also motivated by the observation that AI should complement rather than replace traditional software-engineering mechanisms. Fan et al. (2023) emphasize the importance of hybrid approaches combining traditional software engineering and LLM-based techniques for reliable software development.

Accordingly, the primary contribution of this paper is not another code-generation model. Instead, it proposes an execution architecture in which AI becomes a controlled component of the interpreter pipeline.

### 1.1 Research Problem

The research problem addressed in this study is: How can artificial intelligence, secure execution, runtime behavioral analysis, adaptive learning, and cloud-assisted intelligence are integrated into a programming interpreter to provide secure, intelligent, scalable, and adaptive program execution?

### 1.2 Research Questions

The study addresses the following research questions:

- a) How can AI-based code understanding, static and dynamic analysis, and anomaly detection be integrated to enhance program interpretation and identify potentially malicious or vulnerable code?
- b) How can sandboxing and adaptive security policies be designed to safely execute AI-assisted programs while preventing unauthorized or malicious code from compromising the host environment?
- c) How can local, edge, and cloud computing be integrated into the proposed AI-assisted execution architecture to balance security, privacy, latency, and scalability, and how effectively does it perform compared with conventional and existing AI-assisted execution environments?

### 1.3 Research Objectives

The objectives are:

- a) To analyze the limitations of conventional interpreter architectures and investigate the role of Large Language Models (LLMs) in intelligent code understanding and program interpretation.
- b) To design and develop a secure AI-integrated interpreter architecture incorporating risk-based execution, AI-assisted threat detection, and sandbox-based isolation mechanisms.
- c) To integrate cloud, edge, and federated-learning technologies into the proposed architecture to enhance security, privacy, scalability, and adaptive execution.
- d) To develop an experimental evaluation methodology for assessing the proposed architecture against conventional and AI-assisted execution environments and to identify future directions toward adaptive and autonomous programming.

### 1.4 Contributions

The principal contributions of the paper are:

- a) A conceptual AI-powered secure intelligent interpreter architecture integrating conventional interpretation with AI-assisted code understanding and analysis.
- b) A multi-layer security and adaptive execution mechanism combining static, dynamic, and AI-based threat detection, mathematical risk scoring, and security-aware sandboxing.
- c) A local-edge-cloud intelligence framework for secure, scalable, privacy-aware, and efficient AI-assisted program execution, supported by an experimental methodology and benchmark strategy.
- d) A comprehensive evaluation and implementation roadmap comparing the proposed framework with existing conventional and AI-assisted approaches and identifying pathways toward adaptive and autonomous programming.

## II.BACKGROUND AND RELATED WORK

**2.1 Conventional Programming Interpreters:** A conventional interpreter typically transforms source code through several stages: Source Code → Lexical Analysis → Parsing → Semantic Analysis → Intermediate Representation → Execution. The lexical analyzer converts source characters into tokens. The parser determines the syntactic structure. Semantic analysis verifies contextual correctness, after which the runtime engine executes the program. This architecture is effective for deterministic program execution but is not inherently designed for learning or contextual reasoning. The proposed framework therefore retains the conventional interpretation pipeline instead of replacing it. AI is introduced as an additional intelligence and security layer.

**2.2 Large Language Models for Software Engineering:** LLMs have substantially expanded the capabilities of AI-based software engineering. Fan et al. (2023) identified applications including coding, design, requirements analysis, repair, refactoring, performance improvement, documentation, and analytics. Hou et al. (2024) conducted a systematic literature review covering the growing body of research on LLMs for software engineering and identified both significant opportunities and limitations. Jiang et al. (2024) reviewed the development of LLMs specifically for code generation and highlighted issues involving data curation, evaluation, practical deployment, ethics, and environmental impact. Code-oriented models provide important technological foundations for the proposed intelligent interpreter. CodeBERT uses a bimodal representation of programming and natural languages (Feng et al., 2020). CodeT5 introduces identifier-aware pretraining for code understanding and generation (Wang et al., 2021). GraphCodeBERT incorporates data-flow information to improve code representation (Guo et al., 2021). Code Llama and StarCoder demonstrate the increasing availability of specialized open models for programming tasks (Rozière et al., 2023; Li et al., 2023). These technologies can provide the intelligence layer of an AI-powered interpreter.

**2.3 Repository-Level and Autonomous Software Engineering:** Code generation from isolated prompts is insufficient for complex software development. Real software engineering requires understanding multiple files, dependencies, existing architecture, tests, and execution feedback. RepoCoder addresses this issue through iterative retrieval and generation using repository-level information (Zhang et al., 2023). SWE-bench extends evaluation to real GitHub issues and requires models to modify real repositories. Its initial evaluation demonstrated that complex software engineering remains challenging for LLMs (Jimenez et al., 2024). SWE-agent further explores agent-computer interfaces that enable language models to interact with software environments (Yang et al., 2024). Agentless demonstrates that simpler, structured approaches can sometimes compete effectively with more complex autonomous-agent architectures (Xia et al., 2024). These studies support an important design principle for AISII: AI should operate through constrained interfaces rather than receiving unrestricted control over the execution environment.

**2.4 AI-Generated Code and Security:** The use of LLMs for code generation creates a new security challenge because generated code can be functionally correct while remaining insecure. Black et al. (2024) experimentally examined the relationship between correctness and security in commercial LLM-generated code and reported that appropriate prompting can reduce vulnerabilities, although model selection remains important. Dozono et al. (2024) investigated LLM-assisted vulnerability detection across multiple programming languages and integrated vulnerability analysis into an IDE environment. Cloud security requires effective mechanisms to protect data confidentiality and mitigate security threats (Alam, 2020). Li et al. (2024) investigated fine-tuning code models using vulnerability-fixing data and reported improvements in secure code generation. Islam et al. (2024) identified data collection, system design, and evaluation as important challenges in LLM-based secure code analysis and proposed an LLM-based vulnerability identification and repair approach. These findings strongly support the integration of security analysis directly into an AI-assisted execution environment.

**2.5 Machine Learning for Vulnerability Detection :** Before LLM-based security analysis, deep-learning approaches were already being investigated for vulnerability detection. VulDeePecker demonstrated the use of deep learning and code gadgets for software vulnerability detection (Li et al., 2018). Transformer-based methods have subsequently improved vulnerability localization. LineVul, for example, investigated line-level vulnerability prediction using Transformer-based techniques. The important implication for AISII is that AI-based security analysis should complement, rather than replace, conventional static analysis. AI/ML can enhance interpreters through **code understanding, error detection, anomaly detection, execution optimization, intelligent debugging, and adaptive execution** (Bihari & Alam, 2025).

**2.6 Federated Learning for Cybersecurity:** Centralized collection of source code and security telemetry may introduce privacy risks. Federated learning provides an alternative in which models can be trained collaboratively without requiring all raw data to be transferred to a central location. Bi et al. (2024) examined federated learning for privacy-preserving cyber-threat detection and found that federated models can achieve performance comparable to centralized models under several experimental conditions. Federated learning is therefore considered in AISII as a mechanism for collaborative improvement of threat-detection models. Cloud security can be enhanced through robust security mechanisms that protect sensitive data and strengthen confidentiality against potential cyber threats (Alam & Singh, 2021).

## III.RESEARCH GAP

The reviewed literature reveals substantial progress in four independent areas:

- AI-assisted software development;
- LLM-based code understanding;
- AI-based vulnerability detection;
- secure execution environments.

However, these capabilities are generally implemented as separate systems. Existing code-generation systems primarily emphasize generation and developer assistance. Vulnerability-detection systems primarily emphasize identifying weaknesses. Sandboxing systems primarily emphasize execution isolation. Cloud AI systems primarily emphasize computational scalability. A unified architecture that integrates these capabilities directly into an interpreter remains insufficiently explored. The identified research gaps are summarized in Table 1.

| Approach                   | Code Understanding | AI Assistance | Security Analysis | Runtime Monitoring | Sand box | Adaptive Execution | Cloud/Edge Integration |
|----------------------------|--------------------|---------------|-------------------|--------------------|----------|--------------------|------------------------|
| Conventional Interpreter   | ✓                  | ✗             | Limited           | Limited            | ✗        | ✗                  | Limited                |
| Static Analysis Tool       | ✓                  | ✗             | ✓                 | ✗                  | ✗        | ✗                  | Limited                |
| AI Coding Assistant        | ✓                  | ✓             | Limited           | ✗                  | ✗        | Limited            | ✓                      |
| LLM Vulnerability Detector | ✓                  | ✓             | ✓                 | ✗                  | ✗        | ✗                  | ✓                      |
| Autonomous Coding Agent    | ✓                  | ✓             | Limited           | Limited            | Limit ed | ✓                  | ✓                      |
| Proposed AISII             | ✓                  | ✓             | ✓                 | ✓                  | ✓        | ✓                  | ✓                      |

Table 1. Research Gap Analysis

The proposed framework therefore attempts to bridge the gap between intelligent programming assistance and security-aware program execution.

IV.PROPOSED AI-POWERED SECURE INTELLIGENT INTERPRETER

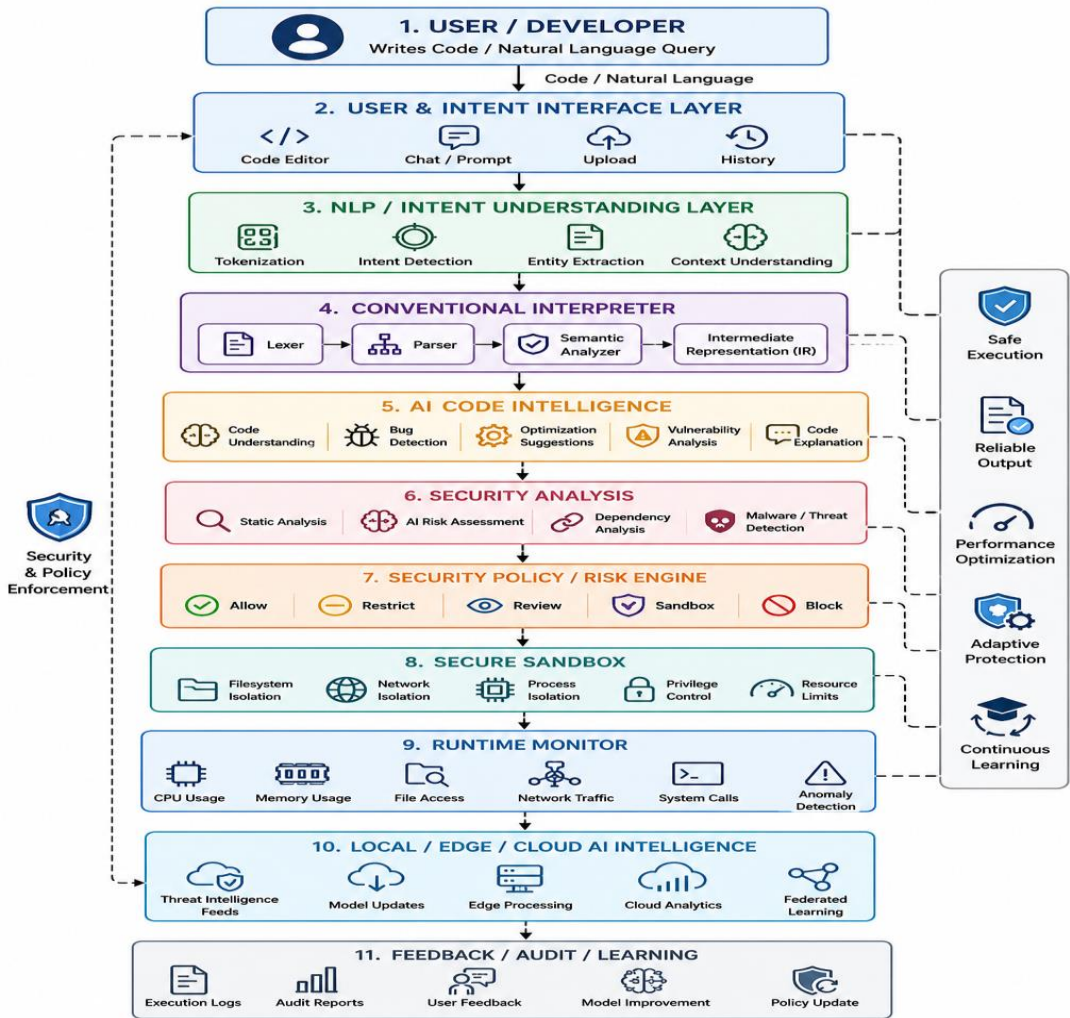


Fig 1. Conceptual Architecture of AISII

**4.1 Architectural Overview:** The proposed architecture consists of nine logical layers:

- a) **Interaction and Intelligent Interpretation Layer** – Integrates user interaction, natural-language/intent processing, conventional interpretation, and AI-based code understanding.
- b) **Security and Threat Analysis Layer** – Combines static analysis, dynamic analysis, AI-assisted threat detection, risk assessment, and runtime monitoring.
- c) **Secure Execution and Distributed Intelligence Layer** – Provides sandbox-based secure execution while utilizing local, edge, and cloud resources for scalable and privacy-aware processing.
- d) **Learning, Adaptation, and Governance Layer** – Supports continuous learning, adaptive security policies, performance optimization, and governance mechanisms for reliable and autonomous program execution.

**4.2 User and Intent Layer:** The system accepts either conventional programming code or natural-language instructions. For example: "Read the student-performance file, calculate the average score, identify students below the average, and generate a report". The NLP module interprets the user's intent and generates a structured representation. However, the resulting code is not executed directly. It enters the same security and interpretation pipeline as conventional source code.

#### **4.3 Conventional Interpretation Layer:**

The conventional interpretation layer consists of:

**Lexical Analyzer:** Converts source code into tokens.

**Parser:** Constructs the syntactic representation.

**Semantic Analyzer:** Checks type consistency, variable usage, scope, and semantic validity.

**Intermediate Representation:** Converts the program into an intermediate representation suitable for further analysis and execution. This layer preserves the reliability of traditional interpreter technology.

### **V. AI CODE INTELLIGENCE LAYER**

The AI layer performs functions that traditional interpreters cannot efficiently perform. These include: natural-language-to-code translation; code explanation; intelligent code completion; error identification; vulnerability identification; optimization recommendations; code refactoring; dependency analysis; execution prediction. The architecture can use an LLM or smaller code model depending on the required task. For privacy-sensitive operations, a local code model can be preferred. Larger models can be accessed through secure cloud infrastructure for complex tasks.

### **VI. PROPOSED SECURITY ARCHITECTURE**

Security is implemented at multiple stages.

**6.1 Static Security Analysis:** Before execution, the system examines: dangerous function calls; shell commands; network operations; file-system access; privilege escalation; hard-coded credentials; insecure dependencies; suspicious code patterns. Traditional static analysis should remain part of the architecture because deterministic rules provide explainable security controls.

**6.2 AI-Based Security Analysis :** The AI security module evaluates source code and intermediate representations for vulnerabilities that may be difficult to detect using simple signatures. The system can classify potential weaknesses using recognized vulnerability categories such as the Common Weakness Enumeration (CWE). AI-based detection should not directly determine execution permission. Instead, it contributes to a broader risk score.

**6.3 Dynamic Behavioral Analysis :** After execution begins, the runtime monitor observes: CPU utilization; memory usage; file system operations; network connections; system calls; process creation; privilege requests; execution duration. Dynamic analysis is particularly important because malicious behavior may not be apparent from source code alone.

### **VII. PROPOSED THREAT-DETECTION ALGORITHM**

#### **Algorithm 1. Adaptive Secure Execution**

Input: Program P

Output: Execution decision D

1. Receive program P
2. Validate input
3. Perform lexical analysis
4. Parse P
5. Perform semantic analysis
6. Generate intermediate representation IR
7. Perform static security analysis
8. Obtain AI-based vulnerability score A
9. Calculate initial risk R0
10. If  $R0 \geq T2$ :  
    Send P to sandbox or block execution
11. Else if  $R0 \geq T1$ :  
    Request review or execute with restricted permissions
12. Else:  
    Permit controlled execution

13. Start runtime monitoring
14. Collect behavioral features B
15. Calculate dynamic risk D
16. Update total risk R
17. If  $R \geq T_2$ :  
Suspend/terminate execution
18. Else if  $R \geq T_1$ :  
Restrict privileges and continue monitoring
19. Record execution outcome
20. Update learning/feedback system
21. Return final execution decision

The major distinction from conventional interpreters is that **execution is not a one-time decision**. The system continuously evaluates risk during runtime.

## VIII.SECURE SANDBOX

The sandbox is responsible for isolating potentially unsafe code.

The sandbox can restrict:

| Resource      | Proposed Control                    |
|---------------|-------------------------------------|
| File system   | Allow-listed directories            |
| Network       | Default deny / controlled endpoints |
| Processes     | Restricted process creation         |
| Privileges    | Least privilege                     |
| Memory        | Resource limits                     |
| CPU           | Execution quotas                    |
| System calls  | Allow/deny policies                 |
| External APIs | Authentication and authorization    |
| Credentials   | No direct access                    |

Container-based technologies can be considered for implementation. Rootless containers and restricted execution environments are particularly relevant where arbitrary code must be executed safely. The sandbox should be considered a containment mechanism, not a replacement for code analysis.

## IX.CLOUD AND EDGE INTEGRATION

**9.1 Local Intelligence:** Local AI should perform low-latency and privacy-sensitive tasks: syntax correction; basic code analysis; simple vulnerability detection; intent classification.

**9.2 Edge Intelligence:** Edge infrastructure can provide additional computing capacity close to the user.

Potential applications include: intermediate AI inference; local threat-intelligence caching; real-time behavioral analysis; low-latency security decisions.

**9.3 Cloud Intelligence:** Cloud infrastructure can provide: large language models; large-scale vulnerability analysis; global threat intelligence; model training; distributed analytics. The proposed hierarchy is therefore: Local→ Edge→ Cloud with the selection determined by latency, privacy, computational complexity, and security risk.

## X.COMPARISON WITH EXISTING APPROACHES

| Feature                        | Traditional Interpreter | AI Coding Assistant | Static Analyzer | Autonomous Coding Agent | Proposed AISII |
|--------------------------------|-------------------------|---------------------|-----------------|-------------------------|----------------|
| Program execution              | ✓                       | Limited             | ✗               | ✓                       | ✓              |
| Natural-language understanding | ✗                       | ✓                   | ✗               | ✓                       | ✓              |
| Code generation                | ✗                       | ✓                   | ✗               | ✓                       | ✓              |
| Code explanation               | ✗                       | ✓                   | Limited         | ✓                       | ✓              |
| Static security analysis       | Limited                 | Limited             | ✓               | Limited                 | ✓              |
| Dynamic monitoring             | Limited                 | ✗                   | ✗               | Limited                 | ✓              |

| Feature                  | Traditional Interpreter | AI Coding Assistant | Static Analyzer | Autonomous Coding Agent | Proposed AISII |
|--------------------------|-------------------------|---------------------|-----------------|-------------------------|----------------|
| AI anomaly detection     | X                       | Limited             | Limited         | Limited                 | ✓              |
| Sandbox execution        | Usually no              | Usually no          | No              | Variable                | ✓              |
| Adaptive security policy | X                       | X                   | X               | Limited                 | ✓              |
| Cloud intelligence       | Limited                 | ✓                   | ✓               | ✓                       | ✓              |
| Edge intelligence        | X                       | Limited             | Limited         | Limited                 | ✓              |
| Federated learning       | X                       | Limited             | X               | X                       | ✓              |
| Risk-based execution     | X                       | X                   | Limited         | Limited                 | ✓              |
| Runtime feedback         | ✓                       | Limited             | X               | ✓                       | ✓              |

Table 2. Comparative Analysis

The proposed framework therefore differs primarily through the integration of AI intelligence and security-aware runtime execution.

## XI. EXPERIMENTAL METHODOLOGY

The current paper proposes the architecture; therefore, empirical results should be obtained through a future prototype implementation. A rigorous experimental evaluation is proposed as follows.

**11.1 Prototype Environment:** A prototype may be developed using: Python; Docker or another container runtime; a lightweight local code model; a cloud-hosted LLM; static analysis tools; runtime monitoring; a vulnerability database; ML-based anomaly detection. The implementation should support at least Python initially because Python provides comparatively accessible interpreter architecture and a large ecosystem of code datasets.

**11.2 Dataset Categories :** Three categories of data should be used.

**Dataset A: Benign Programs:** Programs should represent normal programming tasks such as: file processing; mathematical computation; data analysis; database access; web requests; machine-learning tasks.

**Dataset B: Vulnerable Programs:** Vulnerable code should be obtained from recognized vulnerability datasets and categorized using CWE identifiers.

**Dataset C: Malicious/Adversarial Programs:** Controlled test cases should simulate behaviors such as: unauthorized file access; suspicious network communication; command execution; privilege escalation; resource abuse. The test environment must remain isolated from production systems.

## XII. EXPERIMENTAL SCENARIOS

Four systems should be compared.

**Model A: Conventional Interpreter:** Only conventional interpretation is enabled.

**Model B: AI-Assisted Interpreter:** AI assistance is added without integrated runtime security.

**Model C: Secure Interpreter:** Static analysis and sandboxing are added without AI-based adaptive analysis.

**Model D: Proposed AISII:** AI, static analysis, dynamic monitoring, sandboxing, and adaptive security are integrated. This experimental design enables researchers to determine the incremental contribution of each component.

## XIII. EVALUATION METRICS

### 13.1 Security Metrics

(i) Accuracy =  $(TP + TN) / (FP + FN + TP + TN)$

(ii) Precision =  $TP / (FP + TP)$

(iii) Recall =  $TP / (FN + TP)$

(iv) F1 =  $(2 \times \text{Precision} \times \text{Recall}) / (\text{Precision} + \text{Recall})$

(v) False Positive Rate =  $FP / (TN + FP)$

(vi) False Negative Rate =  $FN / (TP + FN)$

### 13.2 Execution Metrics

The following should be measured: average execution latency; AI inference latency; CPU utilization; memory consumption; sandbox overhead; cloud communication delay; throughput.

The overhead of the proposed framework can be calculated as:  $\text{Overhead}(\%) = (T_{\text{AISII}} - T_{\text{Base}} \times 100) / T_{\text{Base}}$  where  $T_{\text{AISII}}$  represents execution time using the proposed system and  $T_{\text{Base}}$  represents execution time of the conventional interpreter.

### 13.3 Security-Performance Trade-off

The central experimental question should not be whether AISII detects the maximum number of threats regardless of cost. Instead, evaluation should examine the relationship: Security Gain  $\leftrightarrow$  Performance Overhead.

A system that detects more threats but introduces unacceptable latency may not be appropriate for daily computing.

#### XIV.BENCHMARKING INTELLIGENT CODE UNDERSTANDING

The AI component should be evaluated using established code benchmarks. HumanEval can be used to evaluate functional code generation, while MBPP can evaluate basic programming problems. For repository-level tasks, SWE-bench provides a more realistic evaluation environment because it is derived from real GitHub issues. However, benchmark performance should not be interpreted as equivalent to secure real-world execution. This distinction is important because code that passes functional tests may still contain security vulnerabilities.

Consequently, AISII should use dual evaluation: Functional Correctness+ Security Correctness rather than functional correctness alone.

#### XV.DISCUSSION

The proposed framework extends the conventional interpreter into an **intelligent and security-aware execution environment**. While traditional interpreters determine whether code can be executed, and AI-assisted interpreters can analyze its intended behavior, the proposed framework additionally evaluates whether execution is safe within the current context. Its key advantages include **contextual code understanding** through LLM-based analysis, **adaptive security** through risk-based permission control, **runtime awareness** through dynamic behavior monitoring, and **containment** through sandboxing. The framework also supports **scalability** by leveraging cloud-based AI resources for computationally intensive tasks and local or edge models for latency-sensitive operations. However, several challenges must be addressed. LLM outputs are probabilistic and therefore cannot serve as the sole basis for security decisions; AI-generated code may contain vulnerabilities; AI models may be exposed to prompt injection, data poisoning, and manipulation; cloud-based processing may introduce data-privacy risks; and continuous monitoring can increase computational overhead. Accordingly, the proposed architecture treats AI as a **decision-support component rather than a trusted security authority**, combining AI-based recommendations with deterministic security controls, runtime monitoring, and sandboxing to provide layered protection.

#### XVI.SECURITY THREAT MODEL

The proposed threat model identifies five major attack categories. **Malicious user code** may attempt to compromise the host environment and is mitigated through static analysis, sandboxing, least-privilege execution, and runtime monitoring. **Malicious or vulnerable AI-generated code** may introduce security weaknesses and should be addressed through vulnerability detection, static analysis, testing, sandboxing, and human review. **Prompt injection attacks** may manipulate the AI component into producing unsafe instructions; input isolation, instruction hierarchy, output validation, restricted tool permissions, and policy enforcement can reduce this risk. **Data poisoning attacks** may compromise model training through manipulated data and can be mitigated using trusted datasets, data validation, anomaly detection, and secure federated-learning mechanisms. Finally, **supply-chain attacks** involving compromised dependencies or libraries can be addressed through dependency verification, software composition analysis, package reputation assessment, and cryptographic integrity checks. Together, these measures provide layered protection across the code-analysis, AI, execution, and dependency-management stages.

#### XVII.ETHICAL AND PRIVACY CONSIDERATIONS

An intelligent interpreter may process sensitive information, including source code, files, user commands, execution history, network requests, and security events, creating significant privacy and governance risks. The proposed framework addresses these concerns through **data minimization, local processing where feasible, encrypted cloud communication, explicit user consent, role-based access control, secure audit logging, federated learning where appropriate, and human oversight for high-impact decisions**. AI risk management should also align with recognized governance frameworks such as the **NIST AI Risk Management Framework (AI RMF)**, which organizes risk management around four functions: **Govern, Map, Measure, and Manage**. Within AISII, these functions correspond to establishing security policies and accountability (Govern), identifying threats and risks (Map), evaluating AI and security performance (Measure), and implementing adaptive risk mitigation (Manage). This alignment provides a structured approach to managing privacy, security, transparency, and accountability throughout the system lifecycle.

#### XVIII.IMPLEMENTATION ROADMAP

The proposed system can be implemented through four progressive phases. **Phase I – Secure Interpreter** establishes the core execution environment by integrating a conventional interpreter with static analysis, sandboxing, and runtime monitoring. **Phase II – AI-Assisted Interpreter** introduces LLM-based code understanding, code explanation, vulnerability detection, and intelligent optimization. **Phase III – Adaptive Security** enhances the system with ML-based anomaly detection, dynamic risk scoring, adaptive permission management, and automated threat response. **Phase IV – Distributed and Autonomous Intelligence** extends the framework through edge and cloud AI, federated learning, autonomous software engineering, and self-healing execution. This phased implementation enables incremental development, systematic security validation, and controlled integration of increasingly advanced AI capabilities.

#### XIX.FUTURE RESEARCH DIRECTIONS

The proposed framework offers several promising directions for future development. **Self-healing code** could enable the interpreter to detect vulnerabilities and generate patches, subject to testing and validation before deployment. **Autonomous programming** could further extend the system to generate, test, debug, and execute programs from high-level objectives, although reliable autonomous software engineering remains challenging. **Edge AI** can provide low-latency, privacy-preserving intelligence

through local models, while **federated threat intelligence** can enable multiple interpreter instances to collaboratively improve threat detection without sharing raw source code. **Explainable security decisions** should provide clear reasons for risk classifications, such as restricting execution because of privileged access requests or unauthorized network connections. Finally, **quantum computing** may support future advances in optimization, cryptography, and machine learning; however, its integration should currently be regarded as a long-term research direction rather than a mature component of the proposed framework.

## XX.LIMITATIONS

The current study has several limitations. First, AISII is presented as a conceptual architecture and has not yet been experimentally implemented as a complete system. Second, no performance values are claimed because doing so without a reproducible implementation and dataset would constitute unsupported empirical reporting. Third, LLM-based vulnerability detection remains imperfect. Fourth, runtime monitoring inevitably introduces computational overhead. Fifth, cloud-based AI introduces privacy and network-dependency concerns. Sixth, federated learning introduces its own security challenges, including model poisoning and malicious clients. Finally, the proposed architecture is broad and requires substantial engineering work before complete deployment.

## XXI.CONCLUSION

This paper proposes an **AI-Powered Secure Intelligent Interpreter (AISII)** that integrates adaptive intelligence and security into program execution. While conventional interpreters provide reliable and deterministic execution, they lack contextual understanding, adaptive security, and integrated threat detection. AISII addresses these limitations by combining conventional interpretation with AI-based code analysis, static security assessment, runtime monitoring, and adaptive risk scoring, sandboxing, and cloud-edge intelligence. A key design principle is that **AI supports security decisions but does not serve as the sole trust authority**; its outputs are therefore reinforced by deterministic security controls and isolated execution. The proposed risk model combines static, AI-based, and behavioral indicators to classify execution as allowed, restricted, sandboxed, or blocked, while the local-edge-cloud architecture balances privacy, latency, and computational requirements. Unlike conventional AI coding assistants, AISII emphasizes not only code generation and understanding but also **safe and context-aware execution**. Future evaluation should assess functional correctness, security effectiveness, computational overhead, and usability using benign, vulnerable, and adversarial programs. Further research may extend the framework toward federated threat intelligence, edge AI, autonomous programming, self-healing code, and quantum-assisted computing. Overall, AISII represents a shift from the traditional interpreter as a simple execution engine toward an **adaptive, intelligent, and security-aware execution environment**.

## REFERENCES

- Black, G. S., Rimal, B. P., & Vaidyan, V. M. (2024). Balancing security and correctness in code generation: An empirical study on commercial large language models. *IEEE Transactions on Emerging Topics in Computational Intelligence*. <https://doi.org/10.1109/TETCI.2024.3446695>.
- Bi, Y., Li, Y., Feng, X., & Mi, X. (2024). Enabling privacy-preserving cyber threat detection with federated learning. *arXiv*. <https://arxiv.org/abs/2404.05130>
- Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. de O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. (2021). Evaluating large language models trained on code. *arXiv*. <https://arxiv.org/abs/2107.03374>
- Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., & Zhang, J. M. (2023). Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering* (pp. 31–53). IEEE.
- Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, T., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020* (pp. 1536–1547). Association for Computational Linguistics. <https://doi.org/10.18653/v1/2020.findings-emnlp.139>
- Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Fu, S., Tufano, M., et al. (2021). GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations*.
- Alam, M. I., & Singh, S. N. (2021). Designing and implementing cloud security using multi-layer DNA cryptography in Python. In M. Chakraborty, R. K. Jha, V. E. Balas, S. N. Sur, & D. Kandar (Eds.), *Trends in wireless communication and information security (Lecture Notes in Electrical Engineering, Vol. 740, pp. 375–385)*. Springer. [https://doi.org/10.1007/978-981-33-6393-9\\_38](https://doi.org/10.1007/978-981-33-6393-9_38)
- Alam, M. I. (2020). Enhancing cloud security using multi-level DNA cryptography. *Splint International Journal of Professionals*, 7(1), 75–82.
- Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., & Wang, H. (2024). Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8), 1–79. <https://doi.org/10.1145/3695988>
- Islam, N. T., Khoury, J., Seong, A., Bou-Harb, E., & Najafirad, P. (2024). Enhancing source code security with LLMs: Demystifying the challenges and generating reliable repairs. *arXiv*. <https://arxiv.org/abs/2409.00571>
- Jiang, J., Wang, F., Shen, J., Kim, S., & Kim, S. (2024). A survey on large language models for code generation. *arXiv*. <https://arxiv.org/abs/2406.00515>
- Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2024). SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*.
- Li, J., Rabbi, F., Cheng, C., Sangalay, A., Tian, Y., & Yang, J. (2024). An exploratory study on fine-tuning large language models for secure code generation. *arXiv*. <https://arxiv.org/abs/2408.09078>
- Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., et al. (2023). StarCoder: May the source be with you! *arXiv*. <https://arxiv.org/abs/2305.06161>
- Li, Y., et al. (2022). Competition-level code generation with AlphaCode. *Science*, 378, 1092–1097.
- Li, Y., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., & Zhong, Y. (2018). VulDeePecker: A deep learning-based system for vulnerability detection. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS)*.

17. MITRE. (2025). MITRE ATLAS: Adversarial threat landscape for artificial-intelligence systems. MITRE.
18. National Institute of Standards and Technology. (2023). Artificial intelligence risk management framework (AI RMF 1.0). U.S. Department of Commerce.
19. National Institute of Standards and Technology. (2024). Artificial intelligence risk management framework: Generative artificial intelligence profile (NIST AI 600-1). U.S. Department of Commerce.
20. Open Worldwide Application Security Project. (2025). OWASP Top 10 for large language model applications. OWASP Foundation.
21. Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., et al. (2023). Code Llama: Open foundation models for code. arXiv. <https://arxiv.org/abs/2308.12950>
22. UNESCO. (2021). Recommendation on the ethics of artificial intelligence. UNESCO.
23. Wang, Y., Le, H., Gotmare, A. D., Bui, N. D. Q., Li, J., & Hoi, S. C. H. (2023). CodeT5+: Open code large language models for code understanding and generation. arXiv. <https://arxiv.org/abs/2305.07922>
24. Alam, I., & Singh, S. N. (2015). Mitigating cloud security challenges with intrusion detection technique. *Splint International Journal of Professionals*, 2(6), 7–15.
25. Wang, Y., Wang, W., Joty, S., & Hoi, S. C. H. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing* (pp. 8696–8708). Association for Computational Linguistics. <https://doi.org/10.18653/v1/2021.emnlp-main.685>
26. Xia, C. S., Deng, Y., Dunn, S., & Zhang, L. (2024). Agentless: Demystifying LLM-based software engineering agents. arXiv. <https://arxiv.org/abs/2407.01489>
27. Bihari, S., & Alam, M. I. (2025). Leveraging recommender systems for course selection in higher education: A pathway to informed decision-making. In *Proceedings of the Recent Advances in Artificial Intelligence for Sustainable Development (RAISD 2025) (Advances in Intelligent Systems Research)*. Atlantis Press. [https://doi.org/10.2991/978-94-6463-787-8\\_27](https://doi.org/10.2991/978-94-6463-787-8_27)
28. Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., & Press, O. (2024). SWE-agent: Agent-computer interfaces enable automated software engineering. arXiv. <https://arxiv.org/abs/2405.15793>
29. Zhang, F., Chen, B., Zhang, Y., Keung, J., Liu, J., Zan, D., Mao, Y., Lou, J.-G., & Chen, W. (2023). RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing* (pp. 2471–2484). Association for Computational Linguistics. <https://doi.org/10.18653/v1/2023.emnlp-main.151>
30. European Union. (2024). Regulation (EU) 2024/1689: Artificial Intelligence Act. Official Journal of the European Union.