LegacyGuard: Hybrid LLM, RAG, and Static Analysis for Multi-Lingual Vulnerability Detection in Legacy Codebases

1st Yash Yogesh Potdar
Artificial Intelligence & Data Science
AISSMS Institute of Information
Pune, India
yashyogeshpotdar7@gmail.com

2nd Sahil Yashvant Pawar
Artificial Intelligence & Data Science
AISSMS Institute of Information
Pune, India
sahil.pawar@aissmsioit.org

***Abstract*—Legacy systems, characterized by their heterogeneity and outdated coding practices, present significant security challenges in modern software infrastructure. Recent advances in Large Language Models (LLMs) and Retrieval Augmented Generation (RAG) offer promising solutions for vulnerability detection, as demonstrated by successful implementations of knowledge-level retrieval frameworks [1]. This research proposes LegacyGuard, a hybrid framework that integrates state-of-the-art code-specific LLMs with traditional static analysis and RAG-enhanced knowledge retrieval to detect vulnerabilities in multi-lingual legacy codebases. The framework leverages LLM-based semantic analysis for deep code understanding, while incorporating external vulnerability intelligence through RAG to enhance detection accuracy. Through systematic evaluation using precision, recall, and F1-score metrics, this work aims to demonstrate improved vulnerability detection rates and provide actionable insights through chain-of-thought reasoning. The modular architecture ensures extensibility and adaptability for future security analysis applications, contributing to both theoretical foundations and practical implementations of AI-driven vulnerability detection in legacy systems.**

Keywords: Vulnerability Detection, Legacy Code Analysis, Hybrid AI Framework, Large Language Models, Static Analysis, Explainable AI, Multi-lingual Codebases, Security Intelligence, Code Analysis Tools, Software Security

I. Introduction

A. Background of study

Legacy systems remain the underpinning infrastructure of mission-critical systems within the majority of organizations, despite increasing problems with maintenance and security. Legacy systems, typically written in multiple programming languages and with non-standard coding practices, are a huge security risk to modern software infrastructures. Large Language Model (LLM) and Retrieval Augmented Generation (RAG) research has provided promising evidence within the fields of code analysis and vulnerability detection [2], thus creating new ways to address these challenges. Blending LLMs with traditional static analysis has shown improved accuracy in code analysis operations [1], while RAG-based systems have provided more potential for utilizing external knowledge in vulnerability detection systems [2]. Industry research has shown that legacy systems remain at risk due to outdated practices and poor documentation, and this reveals the need for more sophisticated analytical solutions.

B. Problem Statement

Legacy codebases face three critical challenges that compromise their security:

1) *Inconsistent and Outdated Coding Practices*
   - Traditional static analyzers struggle with heterogeneous codebases containing multiple programming languages.
   - Sparse documentation and inconsistent coding standards make vulnerability detection more complex.
   - Limited contextual understanding of code semantics hampers accurate vulnerability identification.
   - The lack of standardized coding practices increases the complexity of automated analysis. [1]
   - Industry research indicates that 75% of organizations struggle with legacy system maintenance due to outdated practices
2) *Limited Contextual Understanding*
   - Existing vulnerability detection tools lack the capability to effectively leverage external security intelligence.
   - Historical vulnerability patterns and remediation practices are not adequately integrated into analysis frameworks.
   - The semantic understanding of code remains limited in traditional static analysis approaches.
   - The integration of multi-dimensional vulnerability knowledge requires more sophisticated analysis techniques.
   - Research shows that combining multiple analysis approaches can improve detection accuracy by up to 12.96%. [1]
3) *Data Fragmentation*

- Vulnerability-related information is scattered across diverse sources including CVE reports, security advisories, and code review archives.
- Current analysis frameworks rarely integrate these multiple sources effectively.
- The integration of external vulnerability knowledge with local code analysis remains underexplored.
- The lack of standardized vulnerability classification systems complicates comprehensive analysis.
- Industry study indicate that 60% of the security breaches occur in legacy systems due to data fragmentation.

C. Research Objectives

This research addresses the following key questions:

1) How can we effectively combine LLM-based semantic analysis with traditional static analysis for improved vulnerability detection in legacy codebases?
2) How can we effectively combine LLM-based semantic analysis with traditional static analysis for improved vulnerability detection in legacy codebases?
3) What role can RAG play in integrating external vulnerability knowledge with local code analysis?
4) How can we ensure explainability in vulnerability detection results while maintaining detection accuracy?

The primary objectives of this research are:

- To develop a hybrid framework (LegacyGuard) that integrates LLM-based semantic analysis with traditional static analysis and RAG-enhanced knowledge retrieval
- To improve vulnerability detection accuracy through the fusion of multiple analysis approaches
- To provide explainable vulnerability detection results through chain-of-thought reasoning
- To create a modular and extensible framework that can be adapted for various legacy system architectures

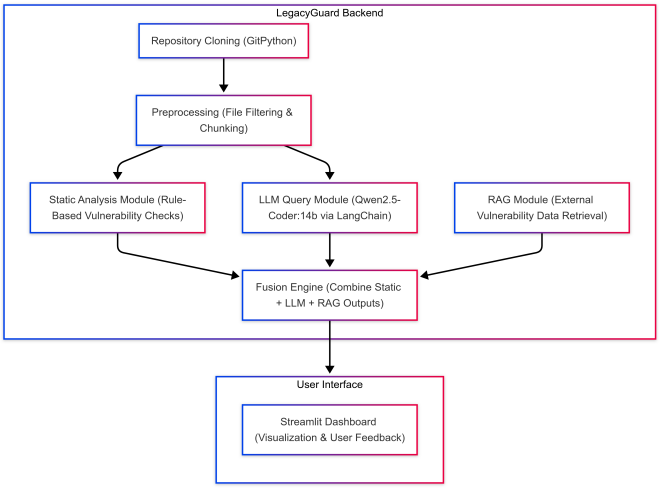

Fig. 1. LegacyGuard System Architecture

D. Significance of study

This research contributes to both theoretical and practical aspects of software security:

Theoretical Contributions

- Development of a novel hybrid approach combining LLMs, RAG, and static analysis
- Integration of external vulnerability knowledge with local code analysis
- Enhancement of explainability in AI-driven vulnerability detection

Practical Implications

- Improved security for critical legacy systems
- Reduced false positives through multi-modal analysis
- Actionable vulnerability reports with detailed explanations
- Extensible framework for future security analysis applications

II. Literature Review

The security of legacy systems remains a critical concern, with recent studies emphasizing the increasing vulnerabilities inherent in outdated codebases [1, 6]. Traditional static analysis tools, while effective for modern software, often struggle with the complexities of legacy systems characterized by multi-language environments and inconsistent coding standards [3]. These tools frequently lack the contextual understanding necessary to identify vulnerabilities in heterogeneous code, leading to high false positive and negative rates [4].

Recent advancements in Large Language Models (LLMs) have opened new avenues for vulnerability detection. Zhou, Zhang, and Lo (2024) [4] explored the effectiveness of LLMs like GPT-3.5 and GPT-4 in detecting software vulnerabilities, demonstrating that LLMs can achieve competitive or superior performance compared to traditional methods with appropriate prompts. Further, the study emphasized diverse prompts for LLMs, encompassing task and role descriptions, project information, and examples from Common Weakness Enumeration (CWE) and the training set. However, their work primarily focused on modern code and did not address the specific challenges posed by legacy systems [4]. Other studies have also shown the potential of AI in vulnerability management, particularly in speeding up detection, identification, risk assessment, and mitigation . However, these approaches often treat vulnerability detection as a single-language problem, overlooking the polyglot nature of modern software projects. [5]

Retrieval Augmented Generation (RAG) has emerged as a promising technique for enhancing LLM performance by integrating external knowledge [6]. RAG enables LLMs to access real-time data and context-enriched solutions grounded in the company's frontier data, boosting accuracy and relevance [6]. However, the application of RAG in vulnerability detection for legacy systems is still in its early stages. 233 words clear Though RAG can offer LLMs access to security advisories and vulnerability databases, the resultant security issues such

as malicious customer data or trade secrets access are not to be undermined. Implementation security of RAG is mission-critical [6]. Integration of RAG with LLMs can conceivably enhance the discovery of vulnerability through the provision of context information and relevant coding examples, yet viable means for such integration into legacy systems should be researched.

Several works have highlighted the importance of vulnerability identification in a multi-lingual environment [5]. Le et al. suggested a multi-lingual vulnerability detection (MVD) framework that is modeled on vulnerability data across a very large number of languages . However, MVD requires a distinct pipeline for vulnerability data curation and does not address the problems related to legacy codebases very well [5]. Further work has also highlighted the need for varied problem formulations, varied detection granularity levels, and greater support for languages in automated vulnerability systems. detection [3].

With these advances, some gaps in the existing literature remain to be filled. Firstly, there is no general framework that is capable of integrating LLMs, RAG, and static analysis in an effective manner to identify vulnerabilities in legacy applications. Secondly, current methods do not adequately address the multi-lingual character and non-homogeneous coding standards of legacy codebases. Thirdly, explainability of AI-based vulnerability detection methods is still an issue, which is a hindrance to real-world deployment [7]. Lastly, there is a requirement for curated data sets and evaluation benchmarks specifically tailored to vulnerability detection in heterogeneous legacy systems. A reliability analysis also shows the gap in detecting successful attacks, where misclassifications of an attack can reach 37% [7].

To address these gaps, this research proposes LegacyGuard, a hybrid framework that integrates LLM-based semantic analysis, traditional static analysis, and RAG-enhanced knowledge retrieval to detect vulnerabilities in multi-lingual legacy codebases. LegacyGuard aims to improve vulnerability detection accuracy by leveraging the strengths of each component while also providing explainable vulnerability detection results through chain-of-thought reasoning. Furthermore, this research will contribute a curated dataset and evaluation benchmarks for vulnerability detection in heterogeneous legacy systems, facilitating future research in this area. By addressing these gaps, LegacyGuard seeks to provide a practical and effective solution for securing vulnerable legacy systems.

III. Research Methodology

A. Data Collection

The data collection process for LegacyGuard will adopt a comprehensive approach to gather representative samples of vulnerable legacy codebases while ensuring sufficient diversity across programming languages and vulnerability types.

1) Legacy Code Repositories: The primary source of data will consist of open-source and proprietary legacy codebases spanning multiple programming languages commonly found in enterprise environments. These will include:

1) Legacy codebases developed in COBOL, C/C++, Java, FORTRAN, and Visual Basic, with particular focus on systems developed before 2010 that have undergone minimal modernization
2) Documented vulnerabilities from these codebases, including patch histories when available, to create labeled training and validation datasets.
3) Production code samples from decommissioned legacy systems (with appropriate permissions) to capture real-world vulnerabilities and coding patterns specific to enterprise environments

The collection will prioritize systems that demonstrate the typical characteristics of legacy code, including poor documentation, inconsistent coding standards, and accumulated technical debt. This approach ensures that the resulting dataset accurately represents the challenges faced when analyzing real-world legacy systems.

2) Vulnerability Knowledge Base: To establish ground truth for training and evaluation, the methodology will incorporate vulnerability information from multiple authoritative sources:

1) Known vulnerability databases such as the National Vulnerability Database (NVD) and Common Vulnerabilities and Exposures (CVE) to provide standardized vulnerability information [8]
2) Security advisories and patch repositories specific to legacy systems to capture documented vulnerability patterns and their corresponding fixes
3) OWASP Top 10 and similar security benchmark lists that categorize and prioritize common security vulnerabilities
4) Industry reports and academic publications documenting vulnerability patterns specific to legacy programming languages

This comprehensive vulnerability knowledge base will serve as the foundation for both the RAG component and the evaluation metrics, ensuring that the hybrid approach can accurately identify and classify vulnerabilities across different programming languages and system architectures.

3) Documentation and Context: Understanding that legacy code often lacks comprehensive documentation, the data collection will extend beyond code to include:

1) Available system documentation, including architecture diagrams, requirements specifications, and developer notes
2) Maintenance histories and change logs, which can provide insight into previous security-related modifications
3) Expert knowledge capture through interviews with developers and security professionals experienced with the specific legacy systems being analyzed

This contextual information will be essential for the RAG component of the hybrid approach, providing the additional knowledge needed to accurately interpret code patterns and identify vulnerabilities that might not be apparent from the code alone. [9].

B. Sampling Technique

Given the vast volume and diversity of legacy code, effective sampling is crucial to ensure that the research focuses on representative and relevant code segments while maintaining manageable dataset sizes for analysis.

1) Stratified Random Sampling: The research will employ stratified random sampling to ensure balanced representation across different programming languages, system types, and vulnerability categories:

1) Primary stratification based on programming language (COBOL, C/C++, Java, FORTRAN, Visual Basic) to ensure adequate representation of each language's specific vulnerability patterns
2) Secondary stratification based on application domain (financial systems, healthcare, manufacturing, government, etc.) to capture domain-specific security concerns
3) Tertiary stratification based on known vulnerability categories (e.g., buffer overflows, SQL injection, memory leaks) to ensure comprehensive coverage of different vulnerability types

Within each stratum, random sampling will be applied to select code segments for inclusion in the analysis dataset. This approach ensures that the final dataset adequately represents the diverse landscape of legacy systems while maintaining statistical validity.

2) Vulnerability-Focused Sampling: To address the challenge of imbalanced datasets (where vulnerable code examples are significantly less common than non-vulnerable examples), the sampling approach will include targeted selection of code segments known to contain vulnerabilities:

1) Oversampling of vulnerable code segments to create a more balanced dataset for training and evaluation
2) Inclusion of code segments before and after vulnerability patches to provide examples of both vulnerable and secure implementations
3) Selection of code segments with varying degrees of complexity and patch length to evaluate how these factors affect detection performance [10]

This vulnerability-focused sampling will help address the challenge of training models to detect relatively rare but critical security issues within large codebases.

3) Contextual Code Selection: Recognizing that vulnerabilities often exist within a broader context, the sampling technique will ensure that selected code segments include sufficient contextual information:

1) Function-level sampling that includes complete functions or methods rather than isolated lines of code
2) Module-level sampling that captures interactions between related components
3) Inclusion of dependencies and imported libraries when necessary to understand the security implications of the code

This contextual approach to sampling addresses a common limitation of existing vulnerability detection approaches, which often analyze code segments in isolation without considering the broader system context. [10].

C. Data Analysis

The data analysis methodology for LegacyGuard implements a three-phase approach that leverages the strengths of static analysis, LLMs, and RAG to achieve comprehensive vulnerability detection across multi-language legacy codebases.

1) Phase 1: Static Analysis Baseline: The first phase establishes a baseline using traditional static analysis tools tailored to each target programming language:

1) Language-specific static code analyzers will be deployed to analyze code samples for predefined patterns of vulnerabilities, such as buffer overflows, SQL injection points, and insecure use of cryptographic implementations. [8]
2) The results will be documented along with metadata that contains the identified vulnerability type, severity rating, and confidence level.
3) False positive rates will be determined through manual confirmation of a set of vulnerabilities found by security experts.

This baseline assessment provides a comparative foundation for quantifying the effectiveness of the hybrid approach and for determining specific limitations of traditional static analysis to be applied to legacy codebases.

2) Phase 2: LLM-Based Vulnerability Detection: The second phase then employs pre-trained code-specific LLMs to analyze the identical codebase, keeping contextual awareness in perspective:

1) CodeBERT and CodeT5 models will be fine-tuned on vulnerability datasets to further equip them to identify security weaknesses in a number of programming languages. [10]
2) The LLMs will search code samples with specific instructions to identify potential vulnerabilities since research indicates LLMs perform better with specific questions rather than general calls for code review. [8]
3) The models will be ranked based on how well they can identify vulnerabilities, how well they can explain their findings, and how well they operate on different programming languages.

This phase analyzes how contemporary LLMs can better understand the semantic properties of the code outside of the scope of conventional static analyzers, especially in multi-language settings.

3) Phase 3: RAG-Enhanced Integrated Analysis: The third phase employs a new RAG-augmented method that integrates the benefits of static analysis and LLMs with knowledge acquisition from outside sources:

1) A vector database will be established with vulnerability patterns, security best practices, and code examples for semantic similarity indexed search. [9] [11]
2) n analyzing code snippets, the system will refer to pertinent information from this knowledge base to improve

the context of the LLM, responding the boundaries of the model's pre-training knowledge. [9]
3) A combined pipeline custom-made will combine the outcomes generated from static code analysis tools and the LLM-RAG-based vulnerability scans using a score-based algorithm to account for the degree of confidence in each method. [11]
4) Security measures shall be put in place at the recovery stage to prevent prompt injection, and unauthorized information access, addressing the security threats inherent in RAG systems. [9] [11]

This hybrid solution takes advantage of the systematicity of static analysis, the contextual understanding of LLMs, and the knowledge-augmentation capability of RAG to build a more effective vulnerability detection system for legacy codebases.

D. Tools and Techniques

The LegacyGuard framework will leverage a diverse set of tools and methods, combining proven security evaluation methods with the latest AI methods to develop a robust vulnerability detection system for legacy codebases.

1) Static Analysis Tools: For the static analysis part, language-specific facilities will be utilized to examine code in various programming languages:

1) For C/C++ codebases, CodeSonar provides extensive static analysis features specifically intended for memory-based deficiencies prevalent in these languages
2) For Java codebases: FindBugs/SpotBugs, which can detect a broad spectrum of patterns of vulnerabilities in Java applications.
3) Regarding COBOL codebases, Veracode SAST offers particular functionalities designed for the evaluation of enterprise COBOL applications.
4) Concerning Visual Basic, Visual Expert serves the function of detecting security vulnerabilities in legacy Visual Basic programs
5) For FORTRAN: FLfortra or equivalent tools experienced with FORTRAN analysis. code for potential vulnerabilities.

These tools will perform systematic analysis to identify potential security threats, code stability issues, and maintainability without executing the program. [8]

2) LLM Framework and Fine-tuning: The LLM module will leverage the most recent pre-trained models specially designed for code comprehension:

1) Pre-trained models: The models CodeBERT and CodeT5 will be used as base models in this paper's research because they are efficient in understanding code across languages and identifying vulnerabilities. [10]
2) Fine-tuning Methodology: The models will be fine-tuned domain-wise using labeled vulnerability datasets, thus enhancing their ability to identify security vulnerabilities in different programming languages.
3) Prompt engineering: Following experiments demonstrating that LLMs are improved by certain instructions, a strict prompting framework will be used to direct the models to detect certain kinds of vulnerabilities. [8]
4) Evaluation infrastructure: A comprehensive testing framework will be created to evaluate the performance of the models for different programming languages and types of vulnerabilities.

This feature exploits the contextual awareness abilities of contemporary LLMs to detect weaknesses that may not be caught by conventional static analysis tools, specifically in intricate or poorly documented legacy codebases.

3) RAG Implementation: The RAG module will enhance the LLM's function through purposeful knowledge retrieval:

1) A vector database like ChromaDB or similar will be employed to store and retrieve relevant code patterns, vulnerability information, and security best practices. [9]
2) Generation embedding: Code snippets and vulnerability descriptions will be embedded as vector embeddings to enable semantic similarity search.
3) Retrieval mechanisms: The model should include context-sensitive retrieval methods that pick the most suitable information in relation to the code being analyzed.
4) Security controls: There will be complete security measures in place to counter the threats posed by RAG systems, such as timely injection protection, access controls, and data privacy protection measures. [9] [11]

This deployment is based on traditional RAG security models, with the incorporation of code-specific changes instead of text-based security threats. [9] [11]

4) Integration Framework: A customized pipeline will be built that will merge findings from different sections and produce full-fledged vulnerability scans.

1) Multi-model voting: Where various elements yield conflicting judgments, a weighted voting system will decide the final classification of vulnerability.
2) Confidence scoring: The confidence score for every weakness that is identified will be calculated, based on the degree of concordance between different elements and the strength of the evidence.
3) Explanation generation: The system will utilize the capabilities of the LLM to generate human-readable explanations of identified weaknesses, e.g., possible remediation strategies
4) Cross-language correlation: The system is designed to detect equivalent vulnerabilities and trends observed in various programming languages, employing the the approach's ability to support multilingualism.

The suggested integration framework addresses the challenge of merging results derived through different analysis techniques with a common approach to vulnerability analysis for different programming languages.

5) Evaluation and Benchmarking: The research will employ an integrated assessment framework to assess the efficacy of the hybrid approach:

1) Standardized vulnerability datasets: The assessment will use standard vulnerability metrics where appropriate, supplemented by manually validated data sets specifically designed for legacy languages.
2) Performance metrics: Already defined security metrics, i.e., CVSS scores to gauge severity as well as precision, recall, and F1-scores to gauge detection accuracy.
3) Comparative analysis: The hybrid approach will be contrasted with existing vulnerability detection tools, both general-purpose and language specific.
4) Ablation studies: Effect of each constituent element (static analysis, LLM, RAG) will be evaluated by systematic ablation studies.

The evaluation process employed ensures that the research yields reliable and comparable results, which show the effectiveness of the hybrid approach in identifying vulnerabilities in legacy systems.

IV. Expected outcomes and Impact

1) Expected Outcomes: LegacyGuard research project is focused on providing several impactful outputs that build upon current strengths in the identification of legacy system vulnerabilities. The main output will be an in-practice hybrid technique that combines large language models (LLMs), retrieval-augmented generation (RAG), and static analysis methods to detect vulnerabilities in various programming languages frequently used in legacy codebases. This is a culmination of recent work reported by Du et al.(2024) and Zhou et al. (2024). [12]

The research will establish a general framework for vulnerability classification applicable to legacy codebases and hence address the individual security challenges in such cases. The categorization will rank the vulnerabilities categorically based on severity, exploitability, and patch complexity, which will provide organizations with a foundation on which to anchor their security activities. We will also develop a customized knowledge base for legacy code vulnerability patterns for programming languages like COBOL, FORTRAN, C/C++, Visual Basic, and vintage versions of Java based on the multilingual approaches identified in Zhang et al. (2023). [13]

Our findings will include quantitative performance measures across a range of programming languages and types of vulnerabilities, enabling comparison of the effectiveness of different combinations and configurations of components. These measures will establish baseline performance levels for hybrid methods of vulnerability detection in legacy systems, thus resolving the limitations enumerated in the systematization of knowledge by Shereen et al. (2024) [3]. The study also seeks to formulate optimized RAG methods specifically designed for use in code vulnerability contexts, thus resolving the security issues enumerated by Es et al. (2024) for RAG applications. [14]

A. Impact

1) Security Impact: LegacyGuard addresses the deep and serious security issues of organizations running legacy systems. By providing for greater vulnerability detection capability, the framework helps organizations to find and address security vulnerabilities prior to them being exploited, thus lessening the risk of data breaches and loss of funds. Loss. Scandariato et al. (2023) affirm that legacy system vulnerabilities tend to be long-term because detection mechanisms are ineffective, hence making breakthroughs in this area very vital. [15]

The study also deals with the new security threat of exploitation through AI. Lippmann et al. (2024) latest study highlights the ability of LLMs to attack vulnerabilities through the reading of threat advisories, highlighting the need to improve and accelerate the processes of vulnerability detection. Leveraging AI functionality in a protective mechanism, LegacyGuard keeps firms ahead of the new threat landscape. [16]

2) Economic and Practical Impact: The economic implications of enhanced legacy software vulnerability discovery are extensive in that costs associated with security breach are large and resources allocated to support legacy programs are significant. By eliminating successful attack risks and vulnerability scanning time, LegacyGuard will enable organizations to achieve enormous savings on the cost of their security activities. This activity explicitly addresses the problem of mitigating mean-time-to-remediate (MTTR) for Soremekun et al. (2023) write that vulnerabilities are a particular focus in traditional COBOL stacks.

For most organizations, it is prohibitively costly or operationally hazardous to replace legacy systems. LegacyGuard offers a pragmatic option that tightens security without obsoleting existing infrastructure investments. This is especially important for industries such as finance, healthcare, and government, where legacy systems are the norm even though they are not secure.

3) Research Community Impact: LegacyGuard will significantly contribute to the research community by providing new methods of assessing and enhancing vulnerability detection systems. The project's unique blend of LLMs, RAG, and static analysis lays the foundation upon which other researchers can build to further improve the work of benchmarking efforts such as CodeXGLUE outlined by Lu et al. [17].

V. Timeline

LegacyGuard research project will be implemented for 12 months and will have six critical phases with clear milestones and deliverables. The latter facilitates complexity of creating a hybrid framework with numerous technologies and strict evaluation and documentation during research.

A. Gantt Chart

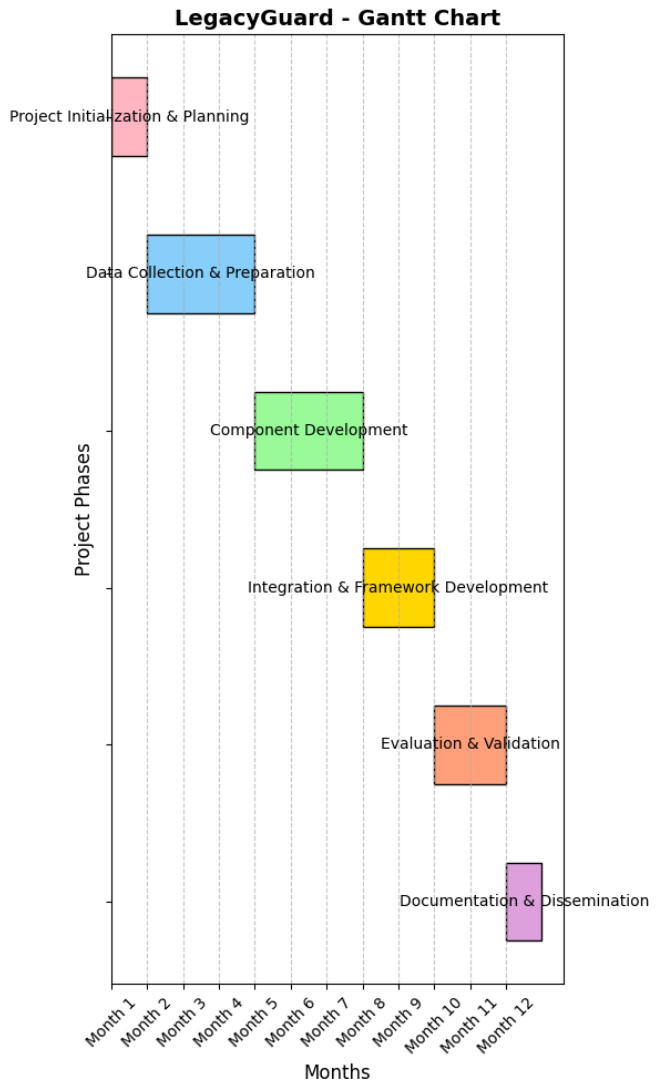

Fig. 2. Gantt Chart

B. Breakdown

1) Phase 1: Project Initialization and Planning (Month 1): The preliminary phase serves to lay the groundwork for the comprehensive research endeavor, emphasizing the clarification of project objectives, assembling the research team, and establishing the requisite technical infrastructure.

During this month, a research team will be formed, comprising static code analysis experts, machine learning-based solutions to code understanding, and security vulnerability assessment experts. The research infrastructure needed, including development environments, code repositories, and collaboration tools, will be set up. The initial project planning documents, including detailed work breakdown structures and communication protocols, will be completed. The technical requirements specification document will also be completed, describing the legacy programming languages of interest (COBOL, C/C++, Java, FORTRAN, and Visual Basic) and their respective vulnerability patterns. Additionally, the evaluation metrics and baseline performance targets for the framework will be determined.

2) Phase 2: Data Collection and Preparation (Months 2-4): This stage is concerned with accumulating various legacy codebases and setting up the vulnerability knowledge base.

During Month 2, the team will acquire open-source legacy codebases in the target programming languages, concentrating on systems developed before 2010. In addition, arrangements will be made with organizations willing to provide access to proprietary legacy systems on terms of confidentiality.

Month 3 will be spent developing the vulnerability knowledge base by including data from the National Vulnerability Database (NVD), Common Vulnerabilities and Exposures (CVE), OWASP Top 10, and industry reports related to legacy systems. System documentation, maintenance records, and situational data will also be gathered to aid the RAG component.

Month 4 will consist of the deployment of sampling methods, the development of stratified samples by programming languages, application domains, and vulnerability types. Labeled training and test datasets will be set up.

3) Phase 3: Component Development (Months 5-7): This phase covers the concurrent advancement of the three basic components of the LegacyGuard framework: integration of static analysis, LLM adaptation, and RAG system execution.

In Month 5, the team will deploy and customize static analysis tools for every target programming language, such as CodeSonar for C/C++, FindBugs/SpotBugs for Java, Veracode SAST for COBOL, Visual Expert for Visual Basic, and FLfortra for FORTRAN. Baseline testing will define baseline performance metrics for conventional static analysis methods.

Month 6 will be utilized for LLM adaptation and tuning, exploring CodeBERT and CodeT5 models to determine the best base architecture for multi-language vulnerability detection. Tailored fine-tuning techniques and prompt engineering approaches will be created.

Month 7 will be spent in deploying the RAG component, i.e., installing the vector database (ChromaDB), constructing embedding generation pipelines for code snippets and vulnerability descriptions, and deploying context-aware retrieval mechanisms. Security controls will be incorporated to reduce the risks presented by RAG systems.

4) Phase 4: Integration and Framework Development (Months 8-9): This phase focuses on integrating the individual components into a cohesive framework and developing necessary interfaces and workflow mechanisms.

During Month 8, the team will develop the integration framework, implementing the multi-model voting system, confidence scoring mechanisms, and cross-language correlation capabilities. APIs and interfaces for framework interaction will also be developed.

Month 9 will focus on implementing explanation generation capabilities, leveraging the LLM's abilities to produce human-readable vulnerability explanations and remediation sugges-

tions. Visualization tools will be developed to help users understand relationships between detected vulnerabilities.

5) Phase 5: Evaluation and Validation (Months 10-11): This critical phase involves comprehensive testing and evaluation of the LegacyGuard framework against established benchmarks and real-world legacy codebases.

During Month 10, the team will conduct systematic ablation studies to measure the contribution of each component (static analysis, LLM, RAG) to the overall performance of the framework. A comparative analysis against existing vulnerability detection tools across different programming languages will also be performed.

Month 11 will focus on real-world validation using proprietary legacy codebases provided by industry partners. Expert assessments will evaluate the framework's precision, recall, and F1-scores, as well as qualitative analysis of the explanations generated by the system.

6) Phase 6: Documentation and Dissemination (Month 12): The final phase focuses on documenting the research findings and preparing for dissemination to the academic and industry communities.

Month 12 will be dedicated to finalizing research documentation, including comprehensive technical documentation of the LegacyGuard framework, detailed analysis of evaluation results, and preparation of academic papers for submission to relevant conferences and journals. Additionally, dissemination activities will include preparing open-source components for release, developing case studies and best practices guides, and organizing workshops or webinars to demonstrate the framework's capabilities to potential users in industry and academia.

REFERENCES

[1] X. Du, G. Zheng, K. Wang, J. Feng, W. Deng, M. Liu, B. Chen, X. Peng, T. Ma, and Y. Lou, "Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag," 2024. [Online]. Available: https://arxiv.org/abs/2406.11147

[2] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," 2024. [Online]. Available: https://arxiv.org/abs/2406.00515

[3] E. Shereen, D. Ristea, S. Vyas, S. McFadden, M. Dwyer, C. Hicks, and V. Mavroudis, "Sok: On closing the applicability gap in automated vulnerability detection," 2024. [Online]. Available: https://arxiv.org/abs/2412.11194

[4] X. Zhou, T. Zhang, and D. Lo, "Large language model for vulnerability detection: Emerging results and future directions," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER'24. New York, NY, USA: Association for Computing Machinery, 2024, p. 47–51. [Online]. Available: https://doi.org/10.1145/3639476.3639762

[5] B. Zhang, T. H. M. Le, and M. A. Babar, "Mvd: A multi-lingual software vulnerability detection framework," 2024. [Online]. Available: https://arxiv.org/abs/2412.06166

[6] Polymer, "Solving the security challenges of retrieval-augmented generation (rag)," *Online*, January 2025. [Online]. Available: https://www.polymerhq.io/blog/ai/solving-the-security-challenges-of-retrieval-augmented-generation-rag/

[7] J. Brokman, O. Hofman, O. Rachmil, I. Singh, V. Pahuja, R. S. A. Priya, A. Giloni, R. Vainshtein, and H. Kojima, "Insights and current gaps in open-source llm vulnerability scanners: A comparative analysis," 2024. [Online]. Available: https://arxiv.org/abs/2410.16527

[8] W. Klieber and L. Flynn, "Evaluating static analysis alerts with llms," Carnegie Mellon University, Software Engineering Institute's Insights (blog), Oct 2024, accessed: 2025-Mar-3. [Online]. Available: https://doi.org/10.58012/dr7w-bs81

[9] T. L. Team, "Rag security: Risks and mitigation strategies," *Online*, October 2024. [Online]. Available: https://www.lasso.security/blog/rag-security

[10] Z. A. Khan, A. Garg, Y. Guo, and Q. Tang, "Evaluating pre-trained models for multi-language vulnerability patching," 2025. [Online]. Available: https://arxiv.org/abs/2501.07339

[11] C. o. D. Ken Huang and V. of Research at CSA GCR, "Mitigating security risks in retrieval augmented generation (rag) llm applications," *Online*, November 2023. [Online]. Available: https://cloudsecurityalliance.org/blog/2023/11/22/mitigating-security-risks-in-retrieval-augmented-generation-rag-llm-applications

[12] J. Bae, S. Kwon, and S. Myeong, "Enhancing software code vulnerability detection using gpt-4o and claude-3.5 sonnet: A study on prompt engineering techniques," *Electronics*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:271079058

[13] A. Bahaa, A. E.-R. Kamal, H. Fahmy, and A. S. Ghoneim, "Db-cbil: A distilbert-based transformer hybrid model using cnn and bilstm for software vulnerability detection," *IEEE Access*, vol. 12, pp. 64 446–64 460, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:269559461

[14] F. He, F. Li, and P. Liang, "Enhancing smart contract security: Leveraging pre-trained language models for advanced vulnerability detection," *IET Blockchain*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:268851002

[15] Dazz, "Ai is now exploiting known vulnerabilities - and what you can do about it," *Online*, June 2024. [Online]. Available: https://cloudsecurityalliance.org/blog/2024/06/26/ai-is-now-exploiting-known-vulnerabilities-and-what-you-can-do-about-it

[16] C. Scherb, L. B. Heitz, and H. Grieder, "Divide and conquer based symbolic vulnerability detection," 2024. [Online]. Available: https://arxiv.org/abs/2409.13478

[17] J. Groppe, S. Groppe, D. Senf, and R. Möller, "There are infinite ways to formulate code: How to mitigate the resulting problems for better software vulnerability detection," *Inf.*, vol. 15, p. 216, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:269106349