

Executive Summary

Large Language Models (LLMs) are increasingly used to assist software development by generating code from natural language prompts, improving productivity and accessibility. However, these models often produce outputs that are insecure, incomplete, or inconsistent due to prompt ambiguity, hallucinations, and the lack of structured requirements.

Key Challenges:

- Insecure and unreliable generated code
- Ambiguous prompts leading to inconsistent outputs
- Lack of structured specifications to guide generation

Limitations of Existing Approaches:

- Prompt engineering is often unreliable
- Model fine-tuning is computationally expensive and not adaptable

Key Gap: A systematic, security-aware framework for generating reliable and vulnerability-free code without retraining

Background



Fig 1 – Security Risks Of AI-Generated Code

Prior work has proposed multiple methods to improve LLM-generated code, each with notable limitations in security, scalability, or reliability.

Prompt Engineering (Zero-shot, CoT): Guides LLM outputs, however, they can be inconsistent, and not secure.

Our Approach: structured specifications to reduce ambiguity

Fine-Tuning (SafeCoder, DiSCo): Security-trained models which are costly, not readily adaptable.

Our Approach: training-free framework

Iterative Methods (PromSec): Feedback-based refinement which lacks full context of previous iterations.

Our Approach: combination of specification, memory, and contextual guidance

Specification Methods (ArchCode): Structured requirements which misses the security aspect.

Our Approach: integrates security standards (MITRE, OWASP, CERT) for concrete security foundations.

Experimental Setup

Framework Components:

Input: Natural language prompt

Specification Generation: LLM-based structured specification builder

Code Generation: LLM (e.g. Llama 3, Ollama)

Static Analysis Tools: CodeQL, Infer, tfsec, Checkov, TFLint

Static Analysis Tools:

CodeQL, Infer, tfsec, Checkov, TFLint

Repair Mechanism:

- Graph-based reasoning, Retrieval Augmented Generation (RAG), Memory component

Fig 2 – Overview of Pipeline and Tools used.

Experimental Test Plan

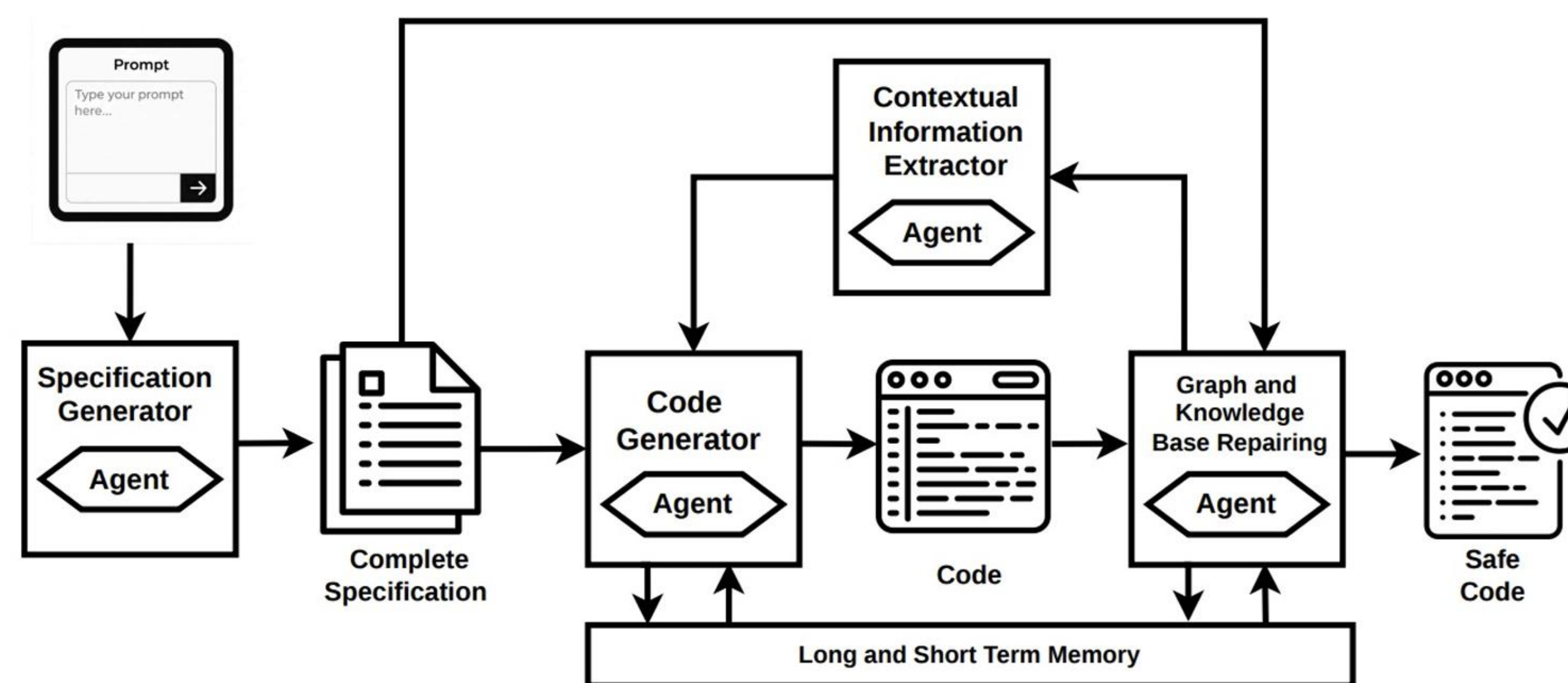


Fig 3 – High-level architecture of the Prompt Patcher framework

Specification Generation: Converts the raw natural language prompt into a structured specification with clear functional and non-functional requirements. This reduces ambiguity and ensures the LLM has complete context, improving consistency and reliability.

Code Generation: Generates initial code based on the structured specification using an LLM. Using a specification instead of a raw prompt improves code quality and aligns outputs with intended requirements.

Static Analysis: Analyzes the generated code using tools like CodeQL, Infer, and IaC scanners to detect vulnerabilities and errors. This enables early identification of security issues without executing the code.

Repair Mechanism: Iteratively refines the code using graph-based reasoning, security knowledge (MITRE, OWASP, CERT), and retrieval-based context. This step fixes vulnerabilities and improves robustness while preserving intended functionality.

Final Output: Produces secure, functional, and compliant code after iterative validation and repair. Ensures higher reliability and reduced vulnerability compared to baseline LLM-generated code.

Experimental Results

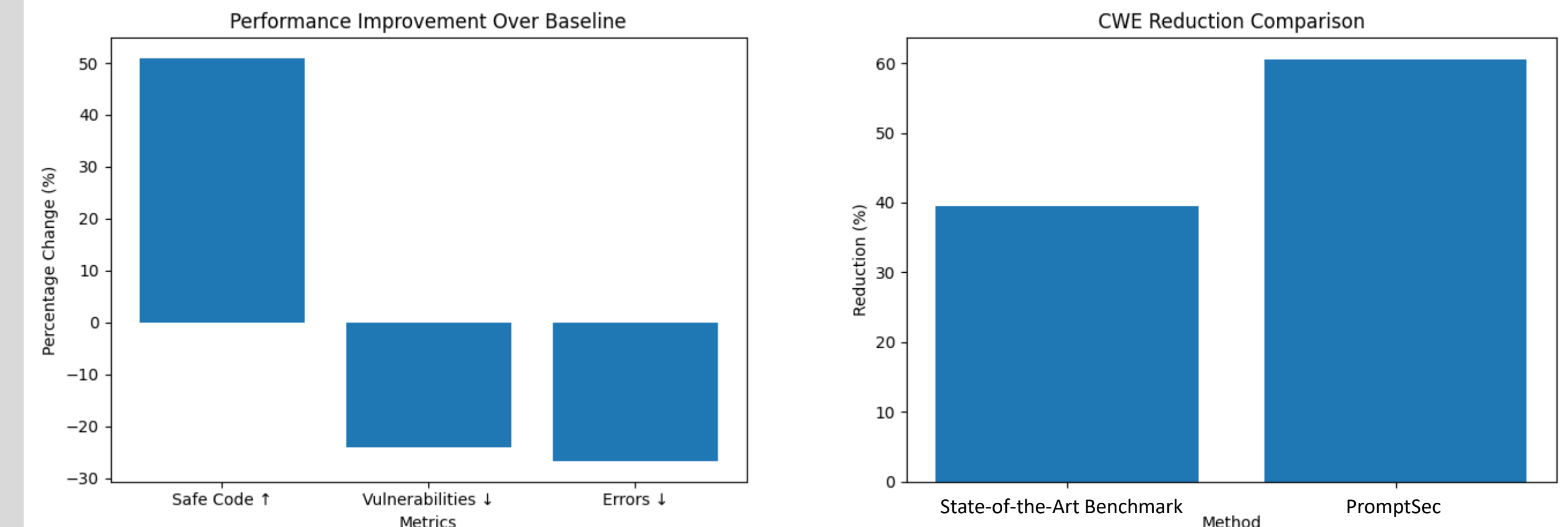


Fig 4 -Performance and Security Improvements of PromptSec Over Baseline and State-of-the-Art Methods

Figures 4 illustrates the effectiveness of the Prompt Patcher framework in improving the security and reliability of LLM-generated code. As shown in the left chart, the framework achieves a substantial increase in safe code generation while reducing both vulnerabilities and compilation errors, indicating that structured specifications and iterative repair significantly enhance output quality. The right chart further demonstrates that Prompt Patcher outperforms existing methods such as PromptSec in reducing detected vulnerabilities, achieving a higher overall CWE reduction.

- The improvement in safe code highlights better **consistency and correctness**
- Reduced vulnerabilities and errors confirm effective **security-aware refinement**

Conclusions

Prompt Patcher demonstrates that incorporating structured specifications into the code generation process significantly improves the security and reliability of LLM-generated outputs. By reducing ambiguity in prompts and guiding generation through well-defined requirements, the framework produces more consistent and functionally correct code.

Additionally, the integration of static analysis, security knowledge, and iterative repair enables effective detection and mitigation of vulnerabilities without compromising intended functionality. The results show that this multi-stage, training-free approach can outperform existing methods while remaining scalable and adaptable to new security challenges.

However, the combined use of multiple components introduces additional computational overhead and increases processing time per prompt. In this context, the increased runtime represents a tradeoff for improved security and reliability. Future work will focus on evaluating the framework across a broader range of LLMs to further validate its generalizability and performance.

Acknowledgements

I would like to sincerely thank Rifat Al Fahmid and Dr. Faysal Hossain Shezan for their continuous guidance, support, and encouragement throughout this project. Their mentorship and insight were instrumental at every stage, and this work would not have been possible without their contributions.

Scan the QR for a quick demo of our entire project

