

# AutoPatch: Multi-Agent Framework for Patching Real-World CVE Vulnerabilities

Minjae Seo<sup>1\*</sup>, Wonwoo Choi<sup>2\*</sup>, Myoungsung You<sup>3</sup>, and Seungwon Shin<sup>3</sup>

<sup>1</sup> Electronics and Telecommunications Research Institute, Daejeon, South Korea  
ms4060@etri.re.kr

<sup>2</sup> Agency for Defense Development, Daejeon, South Korea  
dnjsdnwja@add.re.kr

<sup>3</sup> Korea Advanced Institute of Science and Technology, South Korea  
{famous77, claude}@kaist.ac.kr

**Abstract.** Large Language Models (LLMs) have emerged as promising tools in software development, enabling automated code generation and analysis. However, their knowledge is limited to a fixed cutoff date, making them prone to generating code vulnerable to newly disclosed CVEs. Frequent fine-tuning with new CVE sets is costly, and existing LLM-based approaches focus on oversimplified CWE examples and require providing explicit bug locations to LLMs, limiting their ability to patch complex real-world vulnerabilities. To address these limitations, we propose **AutoPatch**, a multi-agent framework designed to patch vulnerable LLM-generated code, particularly those introduced after the LLMs' knowledge cutoff. **AutoPatch** integrates Retrieval-Augmented Generation (RAG) with a structured database of recently disclosed vulnerabilities, comprising 525 code snippets derived from 75 high-severity CVEs across real-world systems such as the Linux kernel, Chrome, and others. **AutoPatch** combines semantic and taint analysis to identify the most relevant CVE and leverages enhanced Chain-of-Thought (CoT) reasoning to construct enriched prompts for verification and patching. Our unified similarity model, which selects the most relevant vulnerabilities, achieves 90.4% accuracy in CVE matching. **AutoPatch** attains 89.5% F1-score for vulnerability verification and 95.0% accuracy in patching, while being over 50× more cost-efficient than traditional fine-tuning approaches.

**Keywords:** LLM · Multi-Agent · RAG · Vulnerability Detection · Real-World CVE · Software Patching

## 1 Introduction

Large Language Models (LLMs) have become integral tools in software development, demonstrating remarkable capabilities in automated code generation, analysis, and debugging. Prominent examples include ChatGPT [1], Codex [2], CodeLlama [3], StarCoder2 [4], and DeepSeek [5]. These models are widely adopted by developers; Over one million programmers have actively adopted GitHub Copilot by 2024 [6]. Such adoption shows the profound impact of LLM-assisted coding by accelerating the software development cycle.

---

\* Minjae Seo and Wonwoo Choi contributed equally to this work.

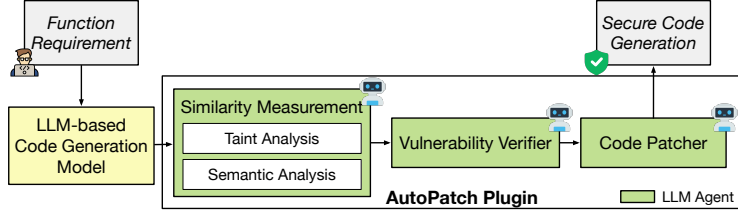


Fig. 1: The overall workflow of AutoPatch.

While LLMs accelerate software development, the prevalence of software vulnerabilities has surged rapidly. In 2024 alone, over 40,000 publicly disclosed vulnerabilities were reported [7], and in just the first two months of 2025, 1,148 Linux kernel vulnerabilities and 39 critical Chrome flaws were disclosed [8]. Despite this trend, LLMs do not automatically learn about vulnerabilities discovered after their training cutoff. Consequently, they may unwittingly suggest code that contains known security vulnerabilities because those issues were not part of their training data. Existing studies have shown that even state-of-the-art LLMs have a notable propensity to generate insecure code; in one analysis, roughly 30% of the code suggestions from LLMs contained known vulnerabilities [6]. This limitation implies that, without proper secure coding guidelines, naive use of LLMs can introduce outdated or insecure coding patterns, potentially resulting in severe risks such as financial loss, service disruption, or data breaches.

To address these concerns, frequent fine-tuning with new vulnerability data (e.g., newly disclosed vulnerabilities and patch code) is one solution. However, this approach is prohibitively time-consuming, resource-intensive [9,10,11]. Instead, recent research has turned to prompt engineering techniques, such as Chain-of-Thought (CoT) prompting, which structures reasoning to improve vulnerability analysis [12,9,13,14]. However, they often **i)** rely on a limited set of simplified examples covering only a narrow range of CWE vulnerability patterns within a few-shot learning setup, primarily focusing on general code snippets rather than LLM-generated code containing real-world vulnerabilities [12,9], **ii)** provide explicit bug locations to the LLM instead of enabling autonomous detection [9,14], and **iii)** place the burden of vulnerability identification entirely on the model, limiting applicability to complex, real-world security scenarios [15,16]. Consequently, these approaches predominantly reflect a bug tester’s viewpoint, overlooking the software developer’s perspective.

To overcome these limitations, we propose **AutoPatch**, a multi-agent-based system designed to patch LLM-generated code relevant to real-world vulnerabilities, particularly those introduced after an LLM’s training cutoff. As illustrated in Figure 1, **AutoPatch** is structured as a security plugin for LLM-integrated IDEs and consists of three specialized LLM agents: the Similarity Analyzer, the Vulnerability Verifier, and the Code Patcher. When developers provide a functional requirement, an LLM generates initial code. To proactively detect vulnerabilities, the Similarity Analyzer agent extracts key terms and contextual descriptions from the LLM-generated code and performs semantic analysis against the semantic representations of recently disclosed vulnerabilities stored

in a RAG database (RAG DB), calculating a semantic similarity score. In parallel, the agent conducts taint analysis on the LLM-generated code, summarizing the flow of variables and function calls into symbolic representations that omit explicit naming, and calculates a taint similarity score by matching against the database. The semantic and taint similarity scores are then combined into a unified similarity score. To optimize this process, we train a machine-learning model (unified model) which learns optimal weights via pairwise ranking loss, ensuring that relevant CVEs are consistently ranked above irrelevant ones.

Upon identifying a match, the Vulnerability Verifier retrieves the corresponding entry from the RAG DB and constructs a one-shot verification example to explain how the matched vulnerability manifests and its root cause, enriching the LLM query for more accurate assessment. If the generated code is deemed vulnerable, the Code Patcher constructs a corresponding one-shot patching example and queries the LLM to generate a secure revision. The revised code is then re-evaluated by the Vulnerability Verifier, and this cycle repeats until the code is verified to be free of vulnerabilities.

We implement a full prototype of the **AutoPatch**. The unified similarity model is trained using the Adam optimizer, and multi-agent coordination with RAG-enhanced retrieval is achieved using LangChain and a PostgreSQL vector database. We evaluate with GPT-4o, Code Llama, DeepSeek, and o3-mini, generating a total of 525 code snippets across 75 recent high-severity CVEs (including Linux kernel, Chrome, and others). GPT-4o achieves 89.3% fidelity in recreating vulnerabilities; our similarity model achieves 90.4% accuracy in matching CVEs. During verification, **AutoPatch** with GPT-4o reaches F1-scores of 89.5% for vulnerability detection. For patching, **AutoPatch** with GPT-4o successfully patches the vulnerable code with 95.0% accuracy. Notably, fine-tuning at an interval of 5 CVEs across the entire set incurs a cost that is 5,230% higher, demonstrating the efficiency gains provided by our plugin-based approach.

**Contributions.** We make the following contributions:

- We propose **AutoPatch**, a cost-efficient multi-agent framework that eliminates the need for fine-tuning of LLMs to handle newly disclosed CVEs.
- We enhance vulnerability detection and patching accuracy by leveraging a high-severity CVE RAG database with semantic and taint analysis to identify relevant vulnerabilities, and applying advanced reasoning for verification and patch generation.
- We implement a full prototype of **AutoPatch** using LangChain and a PostgreSQL vector database, and evaluate it on high-severity CVEs collected from real-world codebases.

## 2 Background and Motivation

### 2.1 Code Generation Model

Code generation models, a specialized subset of large language models (LLMs), translate natural language prompts into executable code. Trained on large-scale datasets from diverse programming languages, they capture both syntactic and semantic patterns to generate code snippets, functions, or full programs. By 2024,

Table 1: Comparison of Knowledge Cutoff for Code Generation Models

Model	Model Variant	Knowledge Cutoff
ChatGPT [1]	GPT-4o and o3-mini	Oct 2023
Llama 3 [17]	Llama-3-70B	Dec 2023
DeepSeek [18]	DeepSeek-Coder-V2	Nov 2023

over one million developers had adopted GitHub Copilot [6], reflecting their adoption in contemporary programming practices. Prominent models include OpenAI’s ChatGPT [1] and Codex [2], Meta’s LLaMA [17], DeepSeek [18], and Hugging Face’s StarCoder [4]. These models significantly accelerate software development, supporting tasks such as prototyping and debugging.

## 2.2 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) [19] offers a compelling alternative to the resource-intensive retraining and fine-tuning tasks by enhancing pre-trained models with external, domain-specific data. Instead of modifying the model’s internal parameters, RAG integrates a retrieval mechanism that accesses up-to-date and relevant information from external sources, thereby enriching the model’s responses for specialized tasks. This approach not only minimizes computational demands but also ensures that the model remains adaptive and context-aware, making it an attractive solution for applications requiring continuous updates and precision in domain-specific outputs.

## 2.3 Motivation

**Knowledge Cutoff of LLMs.** LLMs are inherently constrained by a knowledge cutoff, meaning that they can only access information available up to a specific date. As shown in Table 1, The GPT-4o and o3-mini models have knowledge cutoffs in October 2023 [1]. Copilot’s Codex model [2], which currently adopts GPT-4o, shares the same cutoff. Meta’s Llama-3-70B model has a cutoff in December 2023 [17], while the recent DeepSeek-Coder-V2 model’s cutoff is November 2023 [18]. Thus, even the most recent iterations of LLMs are constrained by knowledge cutoff dates. This limitation implies that any modifications or updates made to the target software after the cutoff are not recognized by these models. In contrast, open-source platforms such as Chrome or Linux undergo frequent source code changes. For instance, in 2024 alone, the Linux Git repository recorded 75,314 commits, with over 3.6 million lines added and nearly 1.5 million removed [20]. Additionally, a total of 8,093 Linux kernel-related CVEs have been reported [21]. Despite this continuous code revision and security issues, LLMs remain unaware of these updates, as their knowledge remains static beyond their training cutoff.

**Vulnerable Code Generation.** This discrepancy between dynamic code repositories and static knowledge of LLMs can introduce serious vulnerabilities. When developers prompt models with context from recently updated source code,

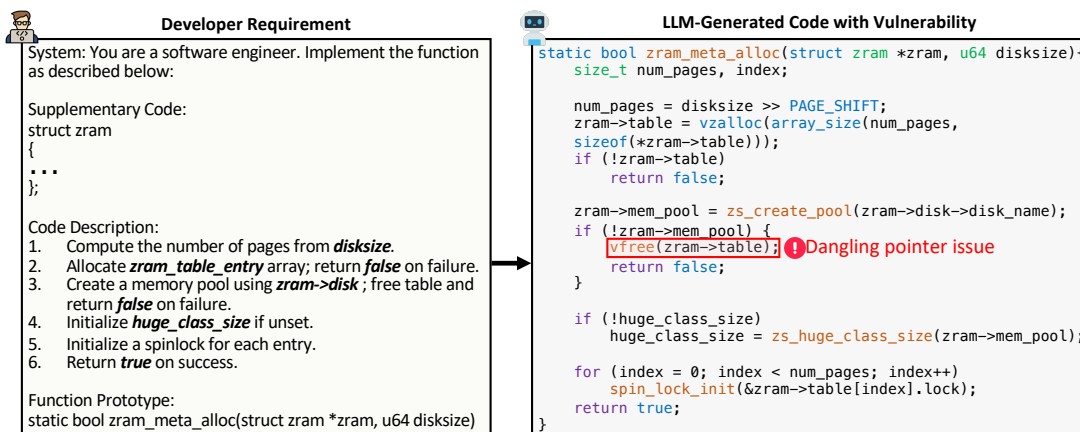


Fig. 2: GPT-4o-2024-11-20 generated vulnerable code for CVE-2025-21671, exposing a dangling pointer that may cause a use-after-free vulnerability.

the generated output may inadvertently include security flaws. For example, as shown in Figure 2, developers may request a function to initialize metadata and memory structures for a zram device, a compressed RAM-based block storage system in the Linux kernel, by providing a prompt with supplementary code snippets referencing existing code snippets. However, the code generated by the gpt-4o-2024-11-20 model contained a vulnerability, specifically a dangling pointer issue due to failure to set the pointer to null, which can potentially lead to a use-after-free vulnerability. These observations emphasize the urgent need for a systematic approach to regularly identify and mitigate high-severity bugs in real-world code. Such a system would not only improve the security of the LLM-generated code but also bridge the gap between the evolving nature of open-source projects and the static knowledge embedded within LLMs.

### 3 AutoPatch Design

Our approach with AutoPatch is to identify and patch vulnerabilities discovered after the knowledge cutoff of code generation models at the point where developers request code. As shown in Figure 3, AutoPatch is built on a multi-agent framework comprising three core LLM agents: the Similarity Analyzer, Vulnerability Verifier, and Code Patcher. In this section, we introduce these agents and their roles in the system.

#### 3.1 AutoPatch Deployment Scenario

We assume a typical development scenario in which developers use LLM-integrated IDEs (e.g., Copilot) to generate code through inline comments, with AutoPatch operating as a security plugin within the IDEs. Given the LLM’s inherent knowledge cutoff, it may produce vulnerable code lacking awareness of recently disclosed CVEs. Therefore, our approach focuses on identifying

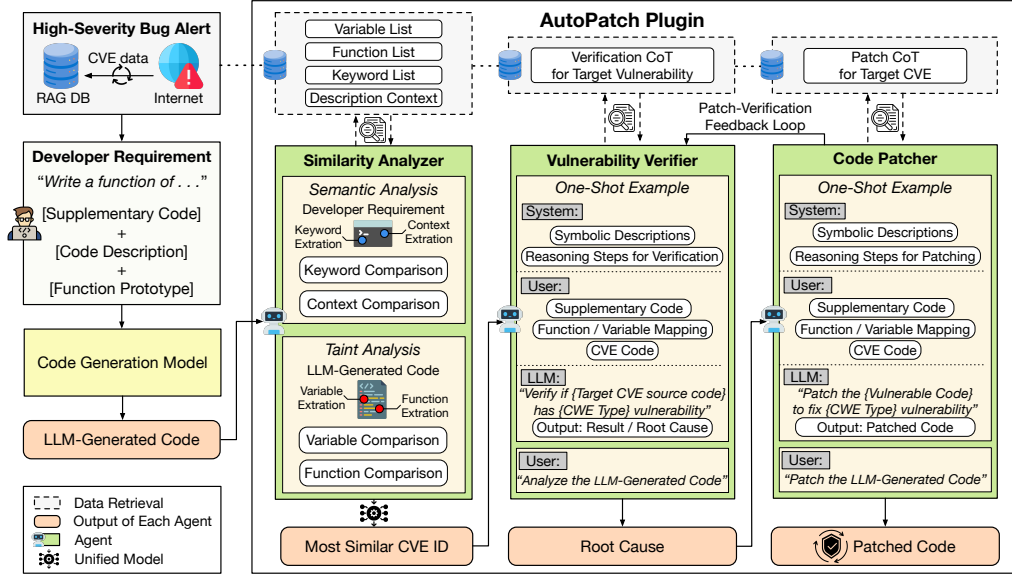


Fig. 3: The overall architecture of AutoPatch.

vulnerabilities in LLM-generated code that exhibit similar patterns to previously disclosed vulnerabilities. Furthermore, to ensure seamless integration with the LLM and real-time usability, we avoid relying on static analysis tools, which typically require code to be fully parsable and at least partially compilable [22,23]. Instead, we propose a lightweight plugin capable of real-time patch generation. In alignment with realistic development settings, we do not disclose vulnerable code locations to the LLM.

### 3.2 Similarity Analyzer

The Similarity Analyzer agent has two key abilities: (i) semantic analysis and (ii) taint analysis. These abilities work in combination to address two key challenges: detecting code that exhibits **similar** structures to known vulnerabilities, and identifying **different** code structures that nonetheless share similar vulnerability patterns. Semantic analysis compares keywords and description contexts from LLM-generated code against known CVEs in our RAG DB, while taint analysis abstracts variables and functions into symbolic representations for pattern-based matching. To unify these different types of similarity features, we propose a Unified Similarity Model that learns optimal weights over multiple similarity metrics—including keyword, context, variable, and function-level comparisons—to rank the most relevant CVE.

#### 3.2.1 Semantic Analysis

With semantic analysis ability, the agent calculates a semantic similarity score using two principal strategies: keyword comparison and context comparison.

**Keyword Comparison.** In this strategy, keywords are extracted from the developer-provided code description using the top 10,000 most frequently used

tags from Stack Overflow [24], and compared against keywords stored in the RAG DB, which are derived from CVE code descriptions using the same tag set. To calculate similarity between two keywords, the Jaccard similarity score is typically utilized. However, exact keyword matching may miss semantically similar terms with lexical variation. To address this, we incorporate rapidfuzz [25], a fuzzy string matching library, and treat two keywords as equivalent if their similarity ratio exceeds 80%. We modify the traditional Jaccard formulation by using fuzzy set operations, where  $\cap_{\text{rf}}$  and  $\cup_{\text{rf}}$  represent rapidfuzz-based intersection and union, respectively. The final similarity score is computed as  $J_{kw} = \frac{|A \cap_{\text{rf}} B|}{|A \cup_{\text{rf}} B|}$ .

**Context Comparison.** While keyword comparison focuses on matching discrete terms, context comparison captures the broader semantic meaning of the code description. In this strategy, the developer-provided code description is encoded into a high-dimensional vector and compared against vulnerable code descriptions in the RAG DB using cosine similarity. This approach enables alignment based on functional intent, even when exact terminology differs. Let  $\mathbf{d}$  and  $\mathbf{v}$  denote the vector representations of the developer’s and CVE descriptions, respectively. The similarity score is computed as  $C_{\text{desc}} = \frac{\mathbf{d} \cdot \mathbf{v}}{\|\mathbf{d}\| \|\mathbf{v}\|}$ .

### 3.2.2 Taint Analysis

With taint analysis ability, the agent calculates a taint similarity score by performing two principal strategies: variable comparison and function comparison. First, the agent extracts variables and functions from the LLM-generated code and abstracts them into symbolic descriptions by removing specific naming details, thereby focusing on their inherent roles rather than literal identifiers, similar to the ones shown in Figure 7: [Vulnerability-Related Variables] and [Vulnerability-Related Functions]. Once these symbolic descriptions are obtained, the Similarity Analyzer compares them with the corresponding representations stored in our RAG DB. To quantify the similarity between the symbolic descriptions of variables and functions, we adopt cosine similarity. Let  $\mathbf{d}$  denote the vector corresponding to the symbolic description extracted from the LLM-generated code, and  $\mathbf{v}$  denote the vector from the RAG DB. The cosine similarity for variable comparison and function comparison are computed as  $C_{\text{var}} = \frac{\mathbf{d} \cdot \mathbf{v}}{\|\mathbf{d}\| \|\mathbf{v}\|}$  and  $C_{\text{func}} = \frac{\mathbf{d} \cdot \mathbf{v}}{\|\mathbf{d}\| \|\mathbf{v}\|}$ , respectively. In addition to obtaining similarity scores, the most probable mappings from symbolic descriptions to variables and functions are utilized during vulnerability verification and code patching (Section 3.3).

### 3.2.3 Unified Similarity Model

**Unified Similarity Score.** We define a unified similarity score  $S$  as a weighted linear combination of the four metrics described above. Let  $J_{\text{kw}}$  be the Jaccard similarity on keywords (as defined earlier), and let  $\tilde{C}_{\text{desc}}$ ,  $\tilde{C}_{\text{var}}$ , and  $\tilde{C}_{\text{func}}$  be the normalized cosine similarities for the descriptions, variables, and functions respectively. The cosine similarity  $C$  is normalized using  $\tilde{C} = \frac{C+1}{2}$ , which maps  $-1 \mapsto 0$  and  $1 \mapsto 1$  (and 0 to 0.5). The score  $\mathbf{S}$  for a given generated code snippet and a particular candidate CVE code is computed as:

$$S = w_1 \cdot J_{\text{kw}} + w_2 \cdot \tilde{C}_{\text{desc}} + w_3 \cdot \tilde{C}_{\text{var}} + w_4 \cdot \tilde{C}_{\text{func}} \quad (1)$$

where  $w_1, w_2, w_3, w_4$  are trainable weights that determine the contribution of each similarity metric. These weights are real-valued parameters that will be learned from training data. A higher unified score  $S$  should indicate a greater likelihood that the candidate CVE corresponds to the same vulnerability or issue present in the LLM-generated code.

**Pairwise Ranking Loss** To learn the optimal weights  $\mathbf{w} = [w_1, w_2, w_3, w_4]$ , we employ a pairwise ranking loss on training examples. For each generated code snippet in the training set, we have one known positive CVE (the correct vulnerability that matches the code) and rest of negative CVE candidates (irrelevant or incorrect vulnerabilities for that code). Let  $\mathbf{S}^+$  denote the unified similarity score for the positive (correct) CVE and let  $\mathbf{S}^-$  be the score for a negative (incorrect) candidate. We define the pairwise ranking loss for this example as:

$$L_{\text{pair}} = \max\left(0, m - (S^+ - S^-)\right) \quad (2)$$

where  $m$  is a margin hyperparameter that specifies how much higher the positive score needs to be compared to a negative score for the pair to be considered correctly ranked. This pairwise loss formulation encourages the model to assign a higher unified score to the true CVE than to any incorrect CVE, with a safety margin. It directly penalizes cases where an irrelevant CVE is ranked too close or higher than the correct one.

**Weight Optimization and Final Outcome.** The weight vector  $\mathbf{w}$  is trained to minimize the total pairwise ranking loss across all training examples. We employ gradient-based optimization (i.e., Adam) to adjust the weights in the direction that reduces  $L_{\text{pair}}$ . The final system takes an LLM-generated code and computes  $J_{\text{kw}}$ ,  $C_{\text{desc}}$ ,  $C_{\text{var}}$ , and  $C_{\text{func}}$  against each CVE candidate in the database and then calculates the unified score  $S$  using Equation 1. Then, the CVE with the highest  $S$  is returned as the most likely relevant vulnerability.

### 3.3 Vulnerability Verifier and Code Patcher

Given the most related CVE ID along with the mapping from symbolic descriptions to variables and functions provided by the Similarity Agent, the remaining tasks are to verify whether the LLM-generated code is vulnerable to a pattern similar to the identified CVE and to patch it if necessary. These tasks are handled by the Vulnerability Verifier agent and the Code Patcher agent, respectively. In this section, we present the operational design and functionality of the Vulnerability Verifier agent and the Code Patcher agent.

Each agent’s primary task is to construct a prompt tailored to achieve its respective objective—vulnerability verification and code patching for the CVE specified by Similarity Agent (Section 3.2). Figure 4 illustrates the final prompts generated by the agents. These prompts follow a typical role-based structure, consisting of three components: System, One-Shot Example, and User. The System component, shown at the top-left, defines the overall task and provides symbolic descriptions of the variables and functions that play critical roles in triggering the CVE. Notably, the names of variables and functions are abstracted (e.g.,

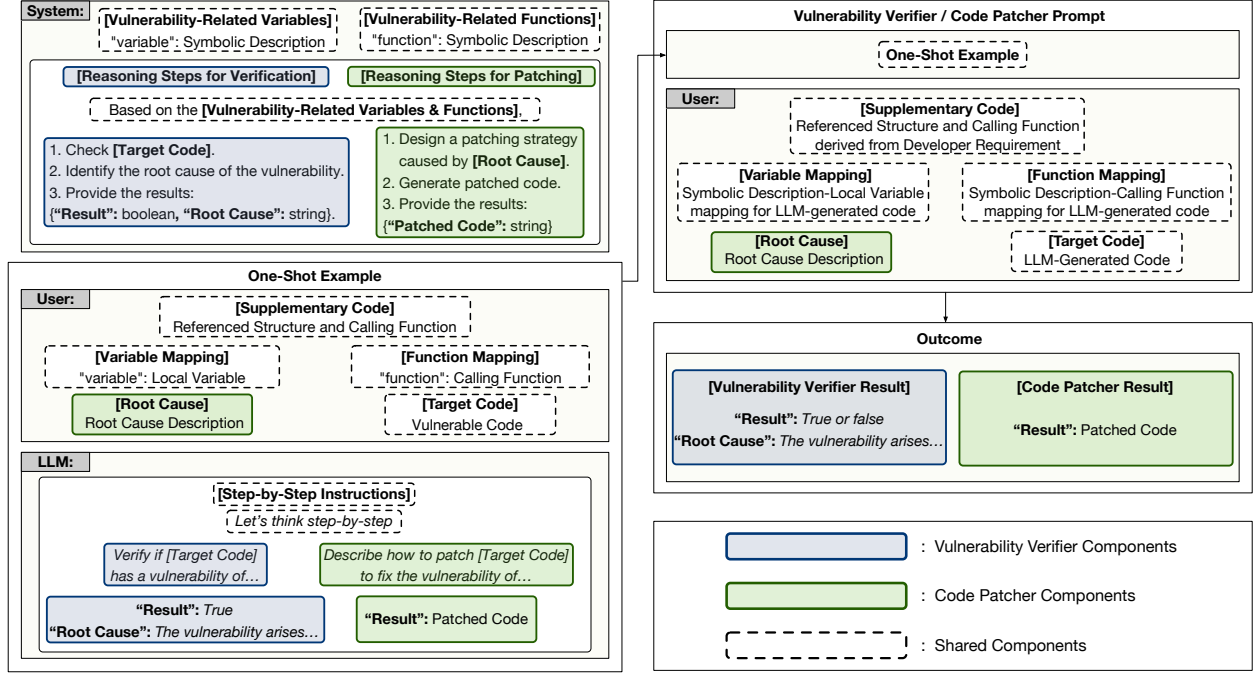


Fig. 4: Verification and Patch prompt for LLM-generated code.

"variable\_1" and "function\_1") to enable generalized vulnerability verification and patching. The one-shot example, located at the bottom-left of the figure, serves as an in-context demonstration of correct reasoning, illustrating how each mapped variable and function should be processed to complete the agent's task. It includes CVE data along with mapping information linking the symbolic descriptions to their corresponding code elements in the CVE. Finally, the User component appears on the top-right of the figure and is structured similarly to the user part of the one-shot example, but instead encodes the LLM-generated code that requires vulnerability verification and patching.

### 3.3.1 Vulnerability Verifier

The Vulnerability Verifier agent constructs a verification prompt to assess whether the LLM-generated code exhibits a vulnerability similar to the CVE identified by the Similarity Analyzer agent. The agent's core abilities are (i) constructing a one-shot example from retrieved CVE metadata and (ii) generating the final verification prompt.

**One-Shot Example.** The one-shot example is dynamically generated from a retrieved CVE entry. Its user part includes the vulnerable code associated with the CVE, supplementary code (e.g., structure definitions and one-hop calling functions), and the actual mapping from the symbolic descriptions to variables and functions. The LLM response demonstrates how to reason over the symbolic mappings, identify the root cause, and deliver a boolean verdict accompanied by an explanatory rationale.

**Verification Prompt.** The agent constructs the final verification prompt by concatenating three components. The System component serves as a fixed preamble, instructing the LLM to analyze the provided code for vulnerabilities and identify their root cause. It introduces symbolic descriptions of vulnerability-related variables and functions from the RAG DB and outlines a structured reasoning process for vulnerability verification. The one-shot example, inserted immediately after the System prompt, serves as an in-context demonstration aligned with these symbolic descriptions. Finally, the User component mirrors the structure of the one-shot User input, including relevant structure definitions, one-hop calling functions, symbolic mappings derived from taint analysis, and the LLM-generated code to be verified. This complete prompt enables the LLM to determine whether a vulnerability exists and explain its root cause. We include a real CVE example, CVE-2025-21671, in the Appendix B (see Figure 7), which corresponds to the same case presented in the motivating example (Figure 2).

### 3.3.2 Code Patcher

Once a vulnerability and its root cause are identified, the Code Patcher agent formulates a prompt to guide the LLM in generating a patch. While maintaining the same role-based structure described in Section 3.3, its focus shifts from verification to patch generation. The agent’s core abilities are (i) constructing a one-shot example based on CVE patch data, (ii) generating the patching prompt, and (iii) providing patch feedback to the Vulnerability Verifier agent.

**Patching Prompt.** The construction of the one-shot example and patching prompt follows the same structure as in the Vulnerability Verifier (Section 3.3.1) but is adapted to guide patch generation. The user exchange within the one-shot example additionally includes the root cause identified by the Vulnerability Verifier to help the model determine which variables and functions contribute to the vulnerability. The LLM response of the one-shot example demonstrates a reasoning path that leads to a patching strategy and the synthesis of a patched version of the code, rather than a vulnerability verdict. The System prompt is updated to instruct the LLM to generate a secure patch for the given code. As a result, the full prompt enables the generation of a patched variant of the vulnerable LLM-generated code (see Appendix B, Figure 7).

**Patch-Verification Feedback Loop.** After the Code Patcher agent generates a patched version of the code, vulnerabilities may remain. To ensure the reliability and security of the final output, the system employs a patch-verification feedback loop, executed for a developer-specified number of iterations. In this loop, the code generated by the Code Patcher agent is returned to the Vulnerability Verifier agent, where it performs the same verification process using the previously constructed one-shot example. This cycle continues until either no vulnerability is detected or the maximum number of iterations is reached. Upon completion of the loop, the system outputs the final version of the code, which is considered to be secure by the Vulnerability Verifier agent.

## 4 Implementation

We implement a full prototype of **AutoPatch**. To rank the most related vulnerabilities, we design a unified model trained with a pairwise loss function using

Table 2: Comparison of Code Reimplementation Accuracy among LLMs

Model	Vuln. Rate	Details				
Code Llama	68.0%	51	19	5	/75	
DeepSeek Coder	80.0%	60	10	5	/75	
DeepSeek-R1	85.3%	64	6	5	/75	
GPT-4o	<b>89.3%</b>	67	3	5	/75	
o3-mini	<u>86.7%</u>	65	2	8	/75	

the Adam optimizer. For seamless multi-agent coordination and RAG-enhanced DB retrieval, we utilize LangChain [26], and adopt PostgreSQL [27] with vector search for entry retrieval, such as variable/function symbolic descriptions, verification/patch reasoning paths, and other details.

**Dataset Collection and Augmentation.** We develop a custom crawler to continuously collect high-severity CVEs from the GitHub Advisory Database [28], Openwall [29], and the Chromium issue tracker [30]. From these sources, we collected 75 high-severity CVEs disclosed in late 2024 and 2025, including 57 from the Linux Kernel and 10 from the Chromium project. Each CVE is reimplemented by extracting the developer’s intent from the vulnerable code and converting it into a natural-language prompt. This prompt is used to guide five LLMs—Code Llama, DeepSeek Coder, DeepSeek-R1, GPT-4o, and OpenAI o3-mini—in generating both vulnerable and patched implementations. In total, this yields 375 code snippets, enabling us to assess LLMs’ ability to reproduce vulnerable patterns and capture structural diversity. Among the models, DeepSeek-R1, GPT-4o, and o3-mini are further used for verification and patch generation due to their reasoning capabilities.




To further increase variability, we apply targeted augmentation strategies for each CVE type, generating an additional 75 vulnerable and 75 patched code snippets. These strategies include renaming local and function parameters using terms typical of other CWE types (e.g., query, freedPtr), aligning parameter names with renamed variables, injecting unreachable code blocks containing potentially vulnerable variable names (e.g., malloc), embedding CWE-style comments, and adding non-functional whitespace and newline variations.

Our implementations are publicly available at <https://github.com/ai-llm-research/autopatch>.

## 5 Evaluation

We conduct a comprehensive evaluation of **AutoPatch** including unified model performance, vulnerability verification, and code patching effectiveness. Also, we analyze the verification and patching performance in relation to CWE types and compare its operational cost against traditional fine-tuning approaches.

Table 3: Comparison of AutoPatch Plugin Performance During the Verification.

Task Details	Metric	AutoPatch with Reasoning Models			Existing Techniques	
		DeepSeek-R1	GPT-4o	o3-mini	VSP [12]	Baseline
CoT Reasoning 	Accuracy	75.74%	<b>87.13%</b>	76.73%	28.22%	28.22%
Vulnerability 	F1-score	81.78%	<b>89.52%</b>	80.33%	20.77%	27.86%
Patched 	Accuracy	85.59%	<b>95.04%</b>	91.30%	55.56%	46.38%

True Positive (TP): Predicted a vulnerability, and a vulnerability existed; CoT was correct.  
False Positive (FP): Predicted a vulnerability, but no vulnerability existed or CoT was incorrect.  
False Negative (FN): Predicted no vulnerability, but a vulnerability existed or CoT was incorrect.  
True Negative (TN): Predicted no vulnerability, and there was no vulnerability; CoT was correct.

### 5.1 Unified Model and Code Reimplementation Performance

Table 2 shows a comparative analysis of code reimplementation accuracy among various LLMs, based on their vulnerability rates. To assess correctness, we manually verify whether each LLM-generated snippet reproduces real-world CVE vulnerabilities. These annotations serve as the ground truth for training our unified model to identify the most closely matching CVE ID.

The Code Llama model exhibits a 68.0% vulnerable code generation rate, likely due to higher hallucination and reduced fidelity to the original logic. DeepSeek Coder and DeepSeek-R1 demonstrate higher vulnerability rates of 80.0% and 85.3%, respectively, indicating improved structural alignment with ground truth code. Notably, GPT-4o and o3-mini show the highest vulnerability rates, 89.3% and 86.7%, respectively, which suggests minimal hallucination and high fidelity in replicating real-world vulnerable patterns.

We train the unified model on the annotated dataset described in Section 4, using the Adam optimizer with a pairwise loss function. The data is split into training, validation, and test sets with a ratio of 70:15:15. Training is performed over 500 epochs with a batch size of 12 and a learning rate of 0.005. On the test set, the unified model achieves 90.41% accuracy in mapping each code snippet to its corresponding CVE ID.

### 5.2 AutoPatch Vulnerability Verifier Performance

In this section, we evaluate the Vulnerability Verifier agent, which assesses whether LLM-generated code contains a vulnerability and generates a corresponding CoT explanation. Since the collected code snippets exhibit a significant class imbalance, with non-vulnerable examples being relatively sparse, we apply random sampling for each CVE to maintain a 2:1 ratio of vulnerable to non-vulnerable snippets. A prediction is considered correct only if both vulnerability detection and CoT reasoning are accurate. To contextualize the performance of AutoPatch, we also compare it against two baselines: compare it against two alternative approaches: VSP [12], which uses a one-shot prompt constructed from a simple CWE-style example relevant to the vulnerability type, and a reasoning-only model, which employs a capable LLM without any in-context examples.

The top portion of Table 3 presents a comparative evaluation of the AutoPatch plugin’s performance during the verification phase. AutoPatch with

GPT-4o achieves the highest performance—87.13% accuracy and 89.52% F1-score—followed by DeepSeek-R1 and o3-mini, both outperforming traditional methods. The results highlight its strength in both identifying vulnerabilities and generating accurate reasoning paths. In contrast, VSP achieves only 28.22% accuracy and 20.77% F1-score, underscoring its inability to handle the step-by-step reasoning required for real-world vulnerabilities.

Overall, **AutoPatch**, particularly when paired with GPT-4o, demonstrates strong capability for accurate and interpretable verification. Existing techniques, such as VSP and the baseline model, perform significantly worse across both vulnerability detection and reasoning tasks. These results underscore the importance of context-aware verification, as conventional methods often struggle to capture the semantic complexity of vulnerable code.

### 5.3 AutoPatch Code Patcher Performance

Among the code snippets identified as vulnerable by the verifier, we employ the Code Patcher agent to generate secure versions of the code. The bottom portion of Table 3 provides a comparative evaluation of the **AutoPatch** plugin’s patching performance, measuring the success rate in remediating identified vulnerabilities. The results clearly demonstrate that **AutoPatch** substantially outperforms existing techniques. GPT-4o achieves the highest patching accuracy at 95.04%, followed by o3-mini at 91.30% and DeepSeek-R1 at 85.59%, showcasing the strength of advanced language models in capturing and acting upon vulnerability semantics. In contrast, VSP and the baseline model achieve significantly lower accuracies of 55.56% and 46.38%, respectively, underscoring their limitations in handling complex, real-world vulnerability scenarios.

These findings highlight the effectiveness of our context-aware patching strategy, which provides models with rich, semantically grounded information about the vulnerable code. Rather than relying on isolated or oversimplified patterns, our approach allows reasoning-capable models to better interpret the structural and functional context of the code, ultimately guiding the generation of more accurate and secure patches.




### 5.4 Performance Comparison Based on CWE Type

To further understand the **AutoPatch**’s performance, we analyze vulnerability verification and patching results across four representative CWE categories: C1 (Arithmetic & Type Errors), C2 (Concurrency Issues), C3 (Memory Safety), and C4 (Validation, Logic, and Resource Handling).

As shown in the first sub-table of Table 4, **AutoPatch** with GPT-4o consistently outperforms all other models, demonstrating strong robustness in detecting diverse types of vulnerabilities. It achieves the highest F1-scores in all four categories, including 96.2% in C2 (Concurrency Issues) and 92.1% in C4 (Validation, Logic, and Resource Handling)—highlighting its ability to capture both syntactic and semantic vulnerability patterns. DeepSeek-R1 and o3-mini also perform well in C4, achieving F1-scores of 82.5% and 90.6%, respectively. These results suggest that our context-aware verification and patching approach

Table 4: Comparison of Performance Based on CWE Type.

D.S.: DeepSeek-R1    4o: GPT-4o    o3-m: o3-mini    VSP: VSP    Base: Baseline

CoT Reasoning  & Vulnerability 							Patched 						
CWE	Metric	D.S.	4o	o3-m	VSP	Base	CWE	Metric	D.S.	4o	o3-m	VSP	Base
C1	Acc	68.2%	<b>80.0%</b>	<u>78.8%</u>	45.9%	47.1%	C1	Acc	90.9%	<b>95.8%</b>	<u>95.7%</u>	68.4%	73.7%
	F1	75.7%	<b>83.8%</b>	<u>82.0%</u>	39.5%	47.1%	C2	Acc	84.6%	<b>96.2%</b>	<u>90.9%</u>	25.0%	38.5%
C2	Acc	<u>80.0%</u>	<b>95.0%</b>	73.8%	20.0%	23.8%	C3	Acc	83.9%	<b>92.7%</b>	<u>87.0%</u>	50.0%	33.3%
	F1	<u>85.7%</u>	<b>96.2%</b>	78.8%	13.5%	22.8%	C4	Acc	85.7%	<b>93.8%</b>	<u>93.7%</u>	45.5%	40.0%
C3	Acc	63.2%	<b>75.9%</b>	<u>63.7%</u>	24.1%	25.5%							
	F1	70.6%	<b>79.1%</b>	67.5%	13.5%	21.2%							
C4	Acc	<u>80.0%</u>	<b>90.9%</b>	<u>89.1%</u>	38.2%	21.8%							
	F1	<u>82.5%</u>	<b>92.1%</b>	<u>90.6%</u>	39.3%	29.5%							

**C1:** Arithmetic & Type Errors, **C2:** Concurrency Issues, **C3:** Memory Safety, **C4:** Validation, Logic, and Resource Handling

is particularly effective at surfacing semantic inconsistencies, especially in validation and logic-related vulnerabilities. In contrast, VSP and the baseline model show significantly lower performance across all CWE types. Their F1-scores drop markedly in C2 and C3, with VSP achieving only 13.5%, and the baseline model scoring 22.8% and 21.2%, respectively. This suggests that the approaches lacking in-context examples struggle to capture the complex interactions and semantic context present in real-world vulnerabilities.

The second sub-table of Table 4 presents patching accuracy across CWE categories. GPT-4o achieves the highest performance, peaking at 96.2% in C2. DeepSeek-R1 and o3-mini also perform well, maintaining over 83% accuracy across all categories, which reflects their robustness in addressing a wide range of vulnerability patterns. In contrast, VSP and the baseline model show limited effectiveness, particularly in more complex categories such as C2 and C3, where their patching accuracy drops to 25.0% and 33.3%, respectively. These results emphasize the critical role of contextual understanding in generating trustworthy and effective vulnerability patches.

### 5.5 Cost Comparison: AutoPatch vs Fine-Tuning

Figure 5 compares the cost of patching CVEs between **AutoPatch** and traditional fine-tuning strategies. We use GPT-4o as the base model, as it achieved the highest performance in our prior evaluations.

Figures 5a and 5b illustrate the cost trends under two common fine-tuning paradigms. In the incremental fine-tuning setting, the model is updated sequentially as new CVEs are introduced. While this avoids retraining from scratch, it still incurs repeated training overhead, resulting in linearly increasing costs. Following standard practices in prior work [31,32], using 5 or 10 epochs leads to a cost of \$37.3 and \$74.6, respectively, when patching 75 CVEs. In contrast, the non-incremental fine-tuning setting retrains the model from scratch using all CVE data seen so far, leading to quadratically increasing costs. For instance,

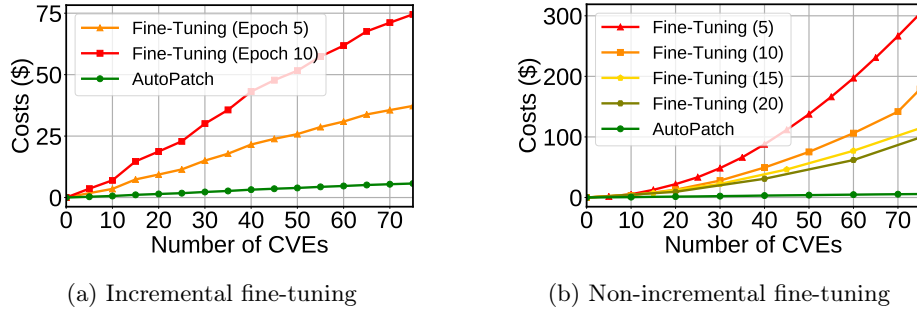


Fig. 5: Cost comparison for AutoPatch and fine-tuning.

fine-tuning every 20 CVEs costs \$99.1, while fine-tuning every 5 CVEs drives the cost up to \$303.8 by the time 75 CVEs are processed.

**AutoPatch**, on the other hand, avoids model updates entirely. Instead, it performs lightweight RAG database entry updates when new high-severity CVEs are disclosed. This results in a nearly constant cost, peaking at just \$5.7, regardless of CVE count. Compared to **AutoPatch**, incremental fine-tuning with 10 epochs is approximately 1,209% more expensive, non-incremental fine-tuning at a 20-CVE interval is 1,639% more expensive, and non-incremental fine-tuning at a 5-CVE interval is 5,230% more expensive. These results highlight **AutoPatch**’s exceptional cost-efficiency and scalability, making it a practical and sustainable alternative to fine-tuning-based approaches for real-world vulnerability patching.

## 6 Discussion and Limitations

**AutoPatch** leverages a retrieval-augmented generation (RAG) framework over a CVE-based knowledge base to automatically verify and patch vulnerable code. While its design allows for generalization beyond the original application context, as shown in Appendix A, **AutoPatch** is fundamentally limited to known vulnerabilities. Specifically, it relies on prior examples of CVEs and their associated patches to reason about and fix new code snippets. As a result, it cannot detect or repair vulnerabilities that have no precedent in the knowledge base, such as zero-day vulnerabilities or novel exploit patterns. While leveraging LLM assistance to discover unknown vulnerabilities is an interesting research direction, it is out of scope for this work.

An additional limitation in our framework is the relatively small number of Chrome-related CVEs. This is primarily because Chrome vulnerabilities, particularly those classified as high-severity, are not made publicly available immediately. These vulnerabilities often undergo a delayed disclosure process to allow time for patch deployment. Nevertheless, by regularly crawling Chrome’s issue tracker, we were able to identify and include 10 CVEs in our dataset.

## 7 Related Work

Alongside our approach, several attempts have explored using LLMs for automated software patching. For example, Nong et al. introduced a vulnerability-semantics-guided CoT approach (VSP) [12], which improved the detection of

vulnerabilities (both a given type and unknown types) and the generation of correct patches, outperforming several baselines. While VSP enhances prompting through semantic guidance, it lacks a deep reasoning process for vulnerability analysis. Instead, it primarily optimizes prompt engineering based on semantic information from a given code snippet. In contrast, **AutoPatch** combines semantic analysis and taint analysis with prior CVE data to guide LLMs toward more context-aware vulnerability analysis.

APPATCH proposed an LLM-based automated patching framework [9]. It applies the prompting techniques from VSP for patching and engages an LLM in adaptive reasoning steps to fix code. However, APPATCH has practical usability constraints, particularly in its reliance on precisely identifying the vulnerable line of code as an input to the model. While this assumption may be feasible for code snippets with known vulnerabilities with predefined locations (e.g., those found through static analysis or CVE reports), it is impractical for detecting and patching unknown or newly emerging security flaws. **AutoPatch** is not limited to code snippets with known vulnerable lines, as it allows LLMs to actively utilize patterns learned from previous CVEs.

ThinkRepair is a framework that leverages LLMs with CoT prompting to generate bug fixes with reasoning [13]. It operates in two phases: first, it constructs a knowledge base of buggy and fixed code annotated with reasoning steps; then, it uses this pool for few-shot prompting to repair new code. While ThinkRepair is the most closely related work to **AutoPatch**, our approach further enhances LLM guidance by incorporating variable and function mappings to strengthen the connection between the generated code and the knowledge base.

## 8 Conclusion

We present **AutoPatch** plugin, a multi-agent framework that secures LLM-generated code through retrieval-augmented vulnerability detection and patching. We reimplement 525 code snippets based on 75 high-severity, real-world CVEs using five popular LLMs to evaluate our system. Among them, GPT-4o shows the best performance, achieving an F1-score of 89.52% in vulnerability verification and 95.04% in patching, particularly excelling in concurrency-related issues. Compared to traditional fine-tuning approaches, **AutoPatch** is significantly more efficient. Compared to traditional fine-tuning approaches, **AutoPatch** demonstrates significantly greater efficiency. Specifically, incremental fine-tuning with 10 epochs incurs approximately a 1,209% higher cost, while non-incremental fine-tuning at 5-CVE intervals results in a 5,230% increase. These results show that **AutoPatch** provides an effective and scalable solution for adapting LLMs to newly disclosed vulnerabilities.

## Appendix

### A AutoPatch Demonstration

To demonstrate how **AutoPatch** verifies and patches vulnerable code, we implemented a simple Image-Processing Daemon that accepts RGB/RGBA image buffers from local clients, processes them through a configurable pipeline of dynamically loaded filter plug-ins (shared objects), and returns the transformed

image. Figure 6 illustrates the moment `AutoPatch` intervenes as a developer leverages an LLM to implement the `load_plugin` function—responsible for loading plug-in files. The LLM-generated `load_plugin` is vulnerable to a Use-After-Free, closely resembling CVE-2024-27530, a vulnerability in the WebAssembly interpreter (wasm3) where a `module` is freed without being properly unregistered from the global module list managed within `runtime`.

Through semantic analysis and taint analysis, `AutoPatch` queries its RAG-backed database and identifies `load_plugin` as being semantically similar to the vulnerable function in CVE-2024-27530. Taint analysis further maps key variables and functions to aid verification. In this case, the mappings are:

- **Variables:** `plg` → `module`, `g_plugins` → `runtime`
- **Functions:** `plugin_register` → `m3_LoadModule`, `free` → `m3_FreeModule`

Along with these mappings, `AutoPatch` retrieves both the verification CoT and patch CoT from the database entry for CVE-2024-27530. It then proceeds to verify and patch the Use-After-Free vulnerability in `load_plugin` by ensuring that the global list (`g_plugins`) is cleared when the plugin (`plg`) is freed.

## B Actual Prompts for CVE-2025-21671

## References

1. OpenAI Platform. <https://platform.openai.com/docs/models>, 2025.
2. Changing the AI model for Copilot code completion. <https://docs.github.com/en/copilot/using-github-copilot/ai-models/changing-the-ai-model-for-copilot-code-completion?tool=vscode>, 2025.
3. Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.
4. Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.
5. Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.
6. Md Imran Hossen, Jianyi Zhang, Yinzhi Cao, and Xiali Hei. Assessing cybersecurity vulnerabilities in code large language models. *arXiv preprint arXiv:2404.18567*, 2024.
7. Information Technology Laboratory at NIST. National vulnerability database(nvd) dashboard. <https://nvd.nist.gov/general/nvd-dashboard>, 2024.
8. The MITRE Corporation. <https://cve.mitre.org/>, 2024.
9. Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. Automated software vulnerability patching using large language models. *arXiv preprint arXiv:2408.13597*, 2024.
10. Ze Sheng, Zhicheng Chen, Shuning Gu, Heqing Huang, Guofei Gu, and Jeff Huang. Llms in software security: A survey of vulnerability detection techniques and insights. *arXiv e-prints*, pages arXiv–2502, 2025.
11. Mete Keltek, Rong Hu, Mohammadreza Fani Sani, and Ziyue Li. Lsast—enhancing cybersecurity through llm-supported static application security testing. *arXiv preprint arXiv:2409.15735*, 2024.

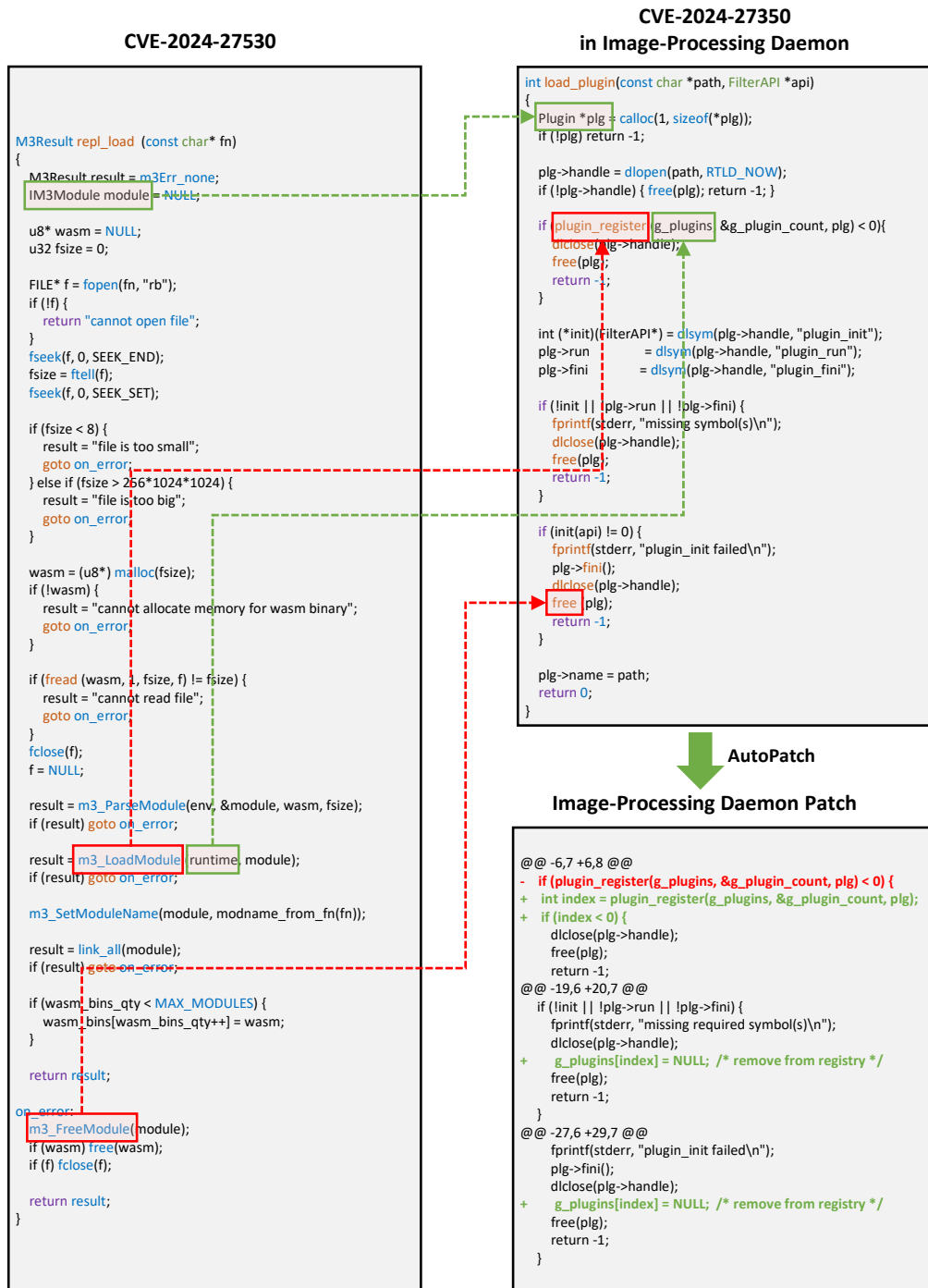
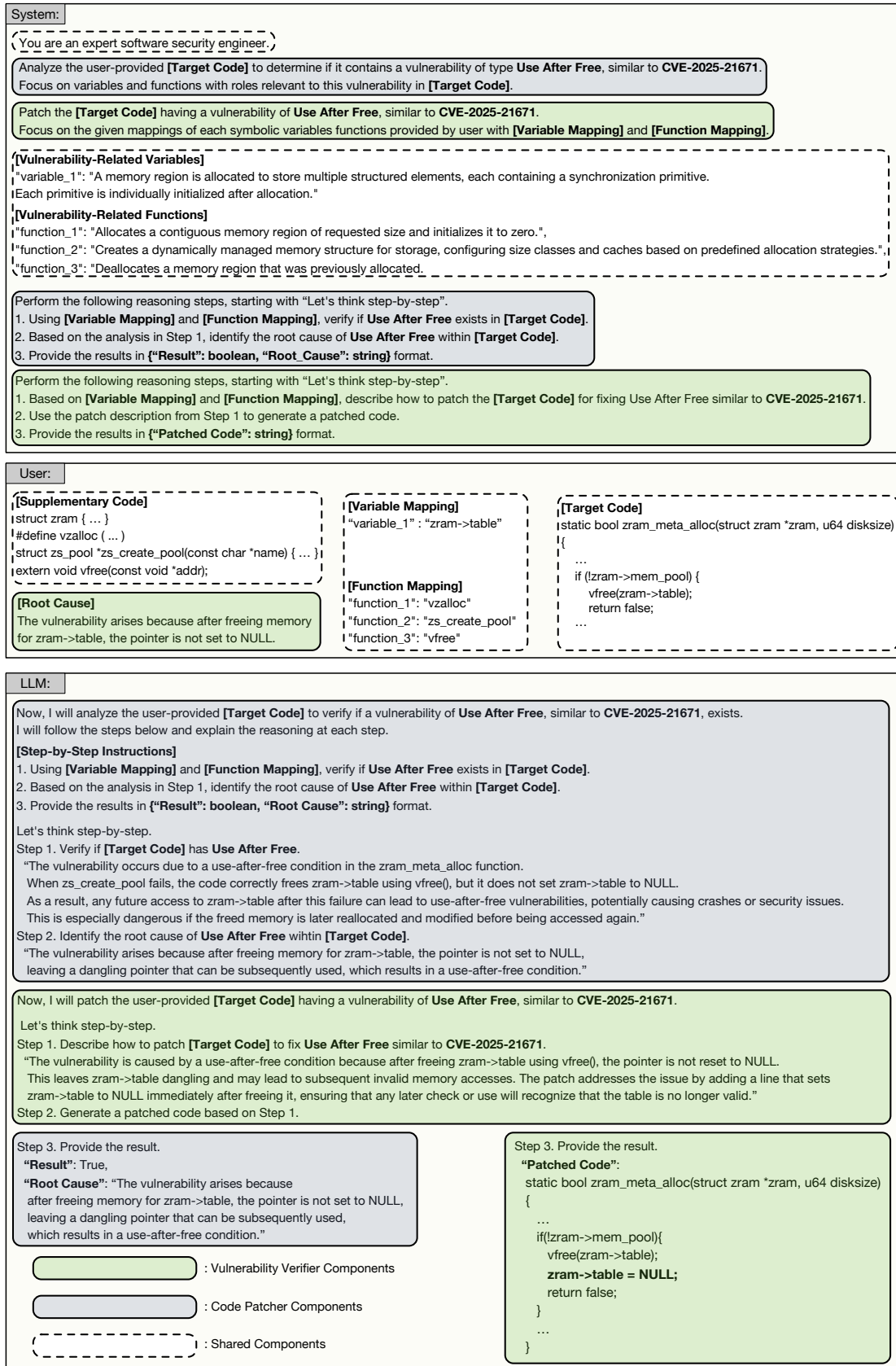


Fig. 6: AutoPatch with load\_plugin function of Image-Processing Daemon.



12. Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities. *arXiv preprint arXiv:2402.17230*, 2024.
13. Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. Thinkrepair: Self-directed automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1274–1286, 2024.
14. Yuwei Zhang, Zhi Jin, Ying Xing, Ge Li, Fang Liu, Jiaxin Zhu, Wensheng Dou, and Jun Wei. Patch: Empowering large language model with programmer-intent guidance and collaborative-behavior simulation for automatic bug fixing. *ACM Transactions on Software Engineering and Methodology*, 2025.
15. Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. Faithful chain-of-thought reasoning. In *The 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (IJCNLP-AACL 2023)*, 2023.
16. Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 862–880. IEEE, 2024.
17. Meta-Llama-3. <https://huggingface.co/meta-llama/Meta-Llama-3-8B>, 2025.
18. Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*, 2024.
19. Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.
20. The Linux Kernel Hit A Decade Low In 2024 For The Number Of New Commits Per Year. <https://www.phoronix.com/news/2024-Linux-Git-Stats>, 2024.
21. 2024 CVE Data Review. <https://jerrygamblin.com/2025/01/05/2024-cve-data-review/>, 2025.
22. Codeql: Semantic code analysis engine. <https://codeql.github.com/>, 2019.
23. Inc. Meta Platforms. Infer: A static analysis tool for java, c, c++, and objective-c. <https://fbinfer.com/>, 2015.
24. Stack exchange data explorer. <https://data.stackexchange.com/>.
25. rapidfuzz. Rapidfuzz. <https://github.com/rapidfuzz/RapidFuzz>, 2020.
26. Introduction to LangChain. <https://python.langchain.com/>, 2025.
27. PostgreSQL: The World’s Most Advanced Open Source Relational Database. <https://www.postgresql.org/>, 2025.
28. GitHub Advisory Database. <https://github.com/advisories/>, 2025.
29. Openwall: Bringing Security into Open Environments. <https://www.openwall.com/>, 2025.
30. Chromium issue tracker. <https://issues.chromium.org/>.
31. Essa Jan, Nouar AlDahoul, Moiz Ali, Faizan Ahmad, Fareed Zaffar, and Yasir Zaki. Multitask mayhem: Unveiling and mitigating safety gaps in llms fine-tuning. *arXiv preprint arXiv:2409.15361*, 2024.
32. Yongchao Chen, Yilun Hao, Yueying Liu, Yang Zhang, and Chuchu Fan. Codesteer: Symbolic-augmented language models via code/text guidance. *arXiv preprint arXiv:2502.04350*, 2025.