CVE-Genie: An LLM-Based Multi-Agent Framework for Reproducing CVEs

Saad Ullah
saadu@bu.edu
Boston University
Boston, Massachusetts, USA

Praneeth Balasubramanian
praneeth@ucsb.edu
University of California, Santa Barbara
Santa Barbara, California, USA

Wenbo Guo
henrygwb@ucsb.edu
University of California, Santa Barbara
Santa Barbara, California, USA

Amanda Burnett
aburne22@asu.edu
Arizona State University
Tempe, Arizona, USA

Hammond Pearce
hammond.pearce@unsw.edu.au
University of New South Wales
Sydney, New South Wales, Australia

Christopher Kruegel
chris@cs.ucsb.edu
University of California, Santa Barbara
Santa Barbara, California, USA

Giovanni Vigna
vigna@cs.ucsb.edu
University of California, Santa Barbara
Santa Barbara, California, USA

Gianluca Stringhini
gian@bu.edu
Boston University
Boston, Massachusetts, USA

Abstract

High-quality datasets of real-world vulnerabilities and their corresponding verifiable exploits are crucial resources in software security research. Yet such resources remain scarce, as their creation demands intensive manual effort and deep security expertise. In this paper, we present CVE-Genie, an automated, large language model (LLM)-based multi-agent framework designed to reproduce real-world vulnerabilities, provided in Common Vulnerabilities and Exposures (CVE) format, to enable creation of high-quality vulnerability datasets. Given a CVE entry as input, CVE-Genie gathers the relevant resources of the CVE, automatically reconstructs the vulnerable environment, and (re)produces a verifiable exploit. Our systematic evaluation highlights the efficiency and robustness of CVE-Genie's design and successfully reproduces approximately 51% (428 of 841) CVEs published in 2024-2025, complete with their verifiable exploits, at an average cost of $2.77 per CVE. Our pipeline offers a robust method to generate reproducible CVE benchmarks, valuable for diverse applications such as fuzzer evaluation, vulnerability patching, and assessing AI's security capabilities.

CCS Concepts

• **Security and privacy → Vulnerability management**; • **Computing methodologies → Machine learning**.

Keywords

CVE Reproduction; LLM Agents

CCS '26, The Hague, Netherlands

ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3832602

ACM Reference Format:
Saad Ullah, Praneeth Balasubramanian, Wenbo Guo, Amanda Burnett, Hammond Pearce, Christopher Kruegel, Giovanni Vigna, and Gianluca Stringhini. 2026. CVE-Genie: An LLM-Based Multi-Agent Framework for Reproducing CVEs. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3832602

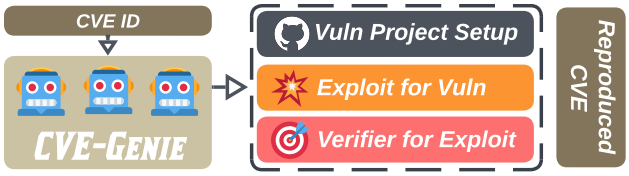


Figure 1: CVE-GenieOverview.

1 Introduction

Tens of thousands of software vulnerabilities are discovered every year, with more than 40,000 vulnerabilities tracked by the National Vulnerability Database (NVD) [43] in 2025 alone. Many of these vulnerabilities are cataloged using the CVE format, which provides a standardized identification system to help organizations address security flaws in both software and hardware. Still, existing vulnerability detection tools, such as those listed by OWASP [45], struggle to keep up with the increasing volume of code released each year.

Developing effective automated vulnerability detection techniques depends on high-quality datasets with reliable ground truth, as low-quality data can lead to unreliable evaluations [51] and to learn spurious correlations, especially in machine learning (ML) based approaches [3, 50]. Although public repositories, such as the

NVD, include hundreds of thousands of CVEs, they often lack critical details, including vulnerable code, environment setup, working exploits, and validation steps, which are all required to effectively reproduce a vulnerability and to build ground truth datasets for research [39]. This is because the main purpose of vulnerability advisories is to alert system administrators about software that needs to be updated, and concrete exploit code is often kept confidential for ethical reasons. Unfortunately, generating this missing information from CVEs alone is labor-intensive and requires significant expertise, e.g., Mu *et al.* [39] spent over 3,600 hours with 43 security expertise to reproduce only 368 memory vulnerabilities in Linux.

To address these challenges, the security community has adopted several approaches: manual annotation [22, 66], inserting bugs into real-world code [37, 42], and heuristic-based mining [6, 9, 12, 14, 41]. While these methods have advanced the field, they come with limitations, such as limited support for software and Common Weakness Enumerations (CWE) categories, mislabeled vulnerabilities [52], and data becoming stale. For instance, the DARPA Cyber Grand Challenge dataset from 2016 [10] quickly became outdated due to its small size and low-complexity bugs [20]. Meanwhile, automated data collection helps address some of these limitations, but often tends to be limited in scope, e.g., ARVO [35] automatically curates reproducible builds with triggering inputs from OSS-Fuzz reports, but its scope is limited to memory corruption bugs in C/C++. **This highlights a clear need for a dataset curation method that enables fully reproducible vulnerability instances across a wide range of software and vulnerability types**.

Recently, LLMs have demonstrated remarkable capabilities in solving complex software engineering (SWE) and security tasks, ranging from automated bug detection [16, 40] and repair [24, 47, 58], to writing fuzzer test harnesses [11]. Moreover, the shift to agentic workflows [7, 33, 62, 63], in which LLM-powered agents are provided access to real-world tools [1], has further pushed the capabilities of general-purpose LLMs and led to significant performance gains on extremely difficult real-world SWE-related benchmarks, such as SWE-bench [25]. This raises the question of whether LLM agents could also be useful in automatically setting up and reproducing existing CVEs, allowing us to build reliable, large, and realistic benchmarks for the security community.

In this paper, we present CVE-Genie, a fully-automated, LLM-driven, multi-agent framework for end-to-end CVE reproduction. CVE-Genie instantiates what we define as key properties of an ideal CVE reproduction system, captured by the acronym **EAGER**. Specifically, an ideal system: Generates working Exploit/proof-of-concept (PoC); Builds Assessors/verifiers for the exploit; Generalizes across broad range of CWEs, programming languages, and software projects; Is End-to-end automated; and Rebuilds vulnerable environments. Specifically, this paper makes the following contributions:

(1) We develop CVE-Genie, an automated end-to-end CVE reproduction pipeline that extracts CVE data (e.g., source code, advisories, patches, etc.), rebuilds the vulnerable environment, and generates exploits and verifiers to assess them.
(2) We design CVE-Genie to demonstrate *compositional intelligence* by a sophisticated architecture, systematically selected LLMs and prompts, and reliable EAGER-style CVE reproduction. As a result, CVE-Genie achieves strong performance across diverse CVEs, vulnerability types, programming languages, and projects, even when CVE information is incomplete.
(3) We systematically evaluate CVE-Genie's architecture and demonstrate that each component is essential for optimal performance. Notably, even advanced LLMs like o3, when used standalone, are incapable of performing EAGER-style reproduction for even a single CVE.
(4) We ran CVE-Genie on 841 CVEs (published between June 2024 and May 2025) and successfully reproduced 428 CVEs across 267 projects, 141 CWEs, and 22 programming languages, demonstrating broad coverage across vulnerability and software types.
(5) We make our framework, source code, datasets of reproduced CVEs, and full logs of agent interactions publicly available (Appendix B). This provides an ongoing valuable contribution for downstream research tasks (see Appendix D).

Table 1: Overview of vulnerability benchmark datasets. Real: synthetic ●, real-world ○, mixed ◗. CVE-Genie results reflect CVEs from Jun 2024–May 2025.

Name	Source	Real	# Projects	# CWEs	# Lang	Automated	Reproducible	PoC	Verifier
SARD [42]	Manual	●	~120	~150	6	✗	✓	✓	✗
SVEN [22]	Manual	○	n/a	9	3	✗	✗	✗	✗
Devign [66]	Manual	○	4	n/a	2	✗	✗	✗	✗
SecLLMHolmes [55]	Manual	◗	4	8	3	✗	✗	✗	✗
VulChecker [37]	Manual	●	n/a	5	2	✓	✗	✗	✗
Draper [53]	SA tool	◗	n/a	9	2	✓	✗	✗	✗
D2A [65]	SA tool	○	6	7	2	✓	✗	✗	✗
VUDENC [57]	Sec. issue	○	812	7	1	✓	✗	✗	✗
DiverseVul [9]	Sec. issue	○	797	150	12	✓	✗	✗	✗
PrimeVul [12]	Patch diff	○	755	140	2	✓	✗	✗	✗
BigVul [14]	Patch diff	○	348	91	2	✓	✗	✗	✗
CrossVul [41]	Patch diff	○	1,675	168	40	✓	✗	✗	✗
CVEfixes [6]	Patch diff	○	1,754	180	30	✓	✗	✗	✗
ARVO [35]	OSS-Fuzz	○	273	4	2	✓	✓	✓	✓
CyberGym [56]	OSS-Fuzz	○	188	4	2	✓	✓	✓	✓
CVE-Bench [67]	Public CVEs	○	36	8	6	✗	✓	✓	✓
Mu *et al.* [39]	Public CVEs	○	1	4	1	✗	✓	✓	✓
CVE-Genie	Public CVEs	○	267	141	22	✓	✓	✓	✓

2 Background and Related Work

2.1 Existing Benchmarks and Limitations

To evaluate various vulnerability detection techniques (rule-based methods [45, 59], ML-approaches [29–31, 37], and LLM-based systems [17–19, 48]), the community needs standardized benchmarks. Prior work has explored four main approaches for addressing these challenges. (1) *Manual efforts* involve analyzing [22] or reproducing [67] vulnerabilities in real-world code, or crafting synthetic examples [55], but datasets are small, lack ecological validity, and often suffer labeling errors due to human mistakes (e.g., 50% data

in Devign [66] have wrong labels [52]). (2) Automated labeling using *static analysis (SA) tools*, such as bandit [5] or infer [23], reduces manual effort but introduces many false positives [49, 65]. (3) Mining *real-world data*, such as *GitHub issues* or CVE *patch commits*, assumes that all removed or modified code is vulnerable; However, this assumption often leads to mislabeled samples, as Risse *et al.* [52] highlighted for DiverseVul [9] and BigVul [14]. (4) Relying on *proven sources*, such as *OSS-Fuzz*, offers verified exploits and sanitizer feedback; However, these are restricted to certain bug types, primarily memory corruption in C/C++, and do not generalize across languages or vulnerability classes [35, 56].

Besides data quality, existing datasets also lack necessary artifacts for dynamic evaluation beyond static labels [3, 50, 55], including dynamic execution environments and reproducible exploits, which is difficult to obtain [39]. Table 1 provides an overview of the current state of vulnerability benchmarks.

2.2 Call for EAGER Style for CVE Reproduction

A *CVE ID* is a unique identifier for publicly disclosed security vulnerabilities in real-world software. Each CVE ID corresponds to a *CVE entry* that includes key details for that CVE, such as vulnerability description, source code, affected versions, security advisories, CWE classifications, and patches, enabling security professionals to assess and mitigate risks. Creating a dataset from the standard CVE database is a promising way towards addressing the aforementioned challenges (see Appendix D). Therefore, we introduce EAGER, a set of crucial criteria for ideal CVE reproduction:

- **E**xploit Generation – (re)creation of an exploit or PoC that reliably triggers the vulnerability in the CVE.
- **A**ssessment – inclusion of a "verifier" or "sanitizer" capable of assessing whether the generated exploit or PoC successfully triggers the vulnerability.
- **G**eneralization – reproduction of CVEs across diverse CWEs, programming languages, and software projects.
- **E**nd-to-end Automation – execution of all stages of CVE reproduction in a fully automated manner.
- **R**ebuild project – reconstruction of the original vulnerable environment to facilitate exploit/PoC execution.

However, none of the existing methods satisfy all the criteria (see Table 2). Manual effort for producing EAGER datasets does not scale, as Mu *et al.* spent 3,600 man-hours to reproduce only 368 vulnerabilities [39]. Moreover, recent work [15, 32, 64, 67, 68] studied LLM-based approaches for PoC generation and exploitation in limited settings, typically on small, manually curated benchmarks (15-40 CVEs) that require substantial manual effort (up to 720+ man-hours) for environment preparation and validation. Similarly, CyberGym [56] and ARVO [35] rely on pre-existing project builds and triggering inputs from OSS-Fuzz, restricting their scope to memory corruption bugs in C/C++. In contrast, CVE-Genie is, to the best of our knowledge, the only specialized framework that performs fully automated end-to-end EAGER-style CVE reproduction, including autonomous environment reconstruction and verifier generation, even when project builds and PoCs are not available.

Table 2: Comparing Potential Vulnerability Reproduction Methods and their EAGER attributes.

Method	Exploit	Assess	General	E2E	Rebuild
Manual	✓	✓	✗	✗	✓
Fuzzers	✓	✗	✗	✗	✗
Metasploit [36]	✓	✗	✗	✗	✗
OSS-Fuzz [35, 56]	✓	✗	✗	✓	✓
CVE-Genie	✓	✓	✓	✓	✓

2.3 Large Language Models and Agents

LLMs are transformer-based neural network models with billions of parameters. They have demonstrated remarkable performance across various application domains, including question answering [44], mathematical reasoning [2], and code generation [4, 8]. Recent work also shows their utility in security tasks, e.g., bug detection and repair [24, 47, 58] and writing fuzzer test harnesses [11].

LLM agents are systems that combine LLMs with tools. They can finish complex tasks via a sequence of actions, including planning, generation, and tool executions [21, 26, 27, 46, 54]. An agentic system can have multiple agents, each with a sub-task, such as reasoning, critiquing, generating, perceiving, remembering, or teaching, as well as its own workflow and tool sets. In security, researchers have started to construct LLM agents to find and repair vulnerabilities. There are multi-agent patching systems, where different sub-agents are responsible for fault localization, patch generation, and validation (e.g., PatchPilot [28], and PatchAgent [61]). Through the recent DARPA AIxCC competition, researchers constructed agentic-based systems with end-to-end capabilities for vulnerability detection and patching [18]. These successes argue for using LLM agents in vulnerability analysis and software development, and for building an EAGER-style automated end-to-end CVE reproduction framework.

3 CVE-Genie

We design CVE-Genie to exhibit *compositional intelligence*, i.e., the ability to solve complex tasks by decomposing them into meaningful sub-tasks whose outputs are explicitly composed and verified to form a correct end-to-end solution. To this end, first, we design a multi-agent architecture that decomposes CVE understanding and reproduction into four interdependent modules: (1) the *Processor* (Section 3.1), which retrieves source code and constructs a structured knowledge base; (2) the *Builder* (Section 3.2), which reconstructs the vulnerable environment; (3) the *Exploiter* (Section 3.3), which reproduces the exploit; and (4) the *CTF Verifier* (Section 3.4), which generates a verifier for the exploit. Second, rather than relying on a single LLM or a fixed prompting strategy across tasks, we systematically select the most suitable LLMs and engineer task-specific prompts for each component of CVE-Genie, resulting in an optimized end-to-end configuration (Section 3.5). Third, we bridge the gap identified in Section 2, namely the lack of any system capable of true EAGER-style CVE reproduction, and ground CVE-Genie's design in the following guiding principles:

(1) **Modular Task Decomposition:** LLMs struggle with long, complex contexts [55] and tasks [25]. Thus, CVE-Genie

decomposes reproduction into focused modules and sub-modules handled by specialized agents with systematically engineered prompts, providing the most suitable architecture for end-to-end CVE reproduction (Section 4.2).

(2) **Robustness to Incomplete Data:** Since missing CVE fields can lower reproduction success by up to 44% [39], CVE-Genie mitigates this by relying on patch or source-code analysis when advisories/PoCs are unavailable (Section 4.3).

(3) **Reliability through Self-Critique:** To counter LLMs' reasoning limits [55], each module uses paired *developer* and *critic* agents in a ReAct-style loop [60], enabling iterative refinement and internal verification (Section 4.2).

Together, these principles elevate CVE-Genie beyond a fixed engineering pipeline and enable CVE-Genie to operationalize *compositional intelligence* in a fully automated setting, yielding a practical end-to-end framework for EAGER-style CVE reproduction at scale (see Appendix D for further details). The following subsections detail the workings of each CVE-Genie component, using CVE-2024-4340 as a running example corresponding to the end-to-end reproduction shown in Figure 2. Moreover, we provide a detailed case study on CVE-2024-5129, a missing-authorization flaw in `lunary-ai`, in the extended version (Appendix D).

Execution Environment. Every CVE reproduction run executes inside an isolated QEMU/KVM virtual machine backed by a single pre-provisioned Ubuntu 22.04 base image. The base image is prepared once via Ansible and includes Python 3, Node.js 20, `pip`, and the CVE-Genie agent library; each individual run then operates on a fresh copy-on-write overlay, ensuring full isolation and reproducibility across experiments. The VM is allocated 2 vCPUs and 8 GB of RAM, and communicates with the host exclusively through a shared directory (for intermediate reproduction artifacts and logs) and SSH (for orchestration). All pipeline stages, *Builder*, *Exploiter*, and *CTF Verifier*, execute within this VM, with the *Builder* module autonomously customizing the environment for the target CVE (e.g., installing vulnerable library versions, starting database services, or enabling AddressSanitizer). Agents interact with the environment only through CLI tools (see Appendix D for details).

3.1 Processor

This module comprises two sub-modules: the *Data Processor*, which locates the vulnerable project's source code and gathers relevant raw resources, and the LLM-based *Knowledge Builder*, which transforms these resources into a structured knowledge base. Below, we provide detailed functionalities of these sub-modules.

3.1.1 Data Processor ①. For the given CVE ID, this sub-module identifies and extracts the vulnerable version of the software along with all relevant CVE details, as follows:

Source Code Extraction. The *Data Processor* begins by locating the source code for the vulnerable project linked to the CVE. It searches the GitHub URLs referenced in the "cvelist" [1] to identify the project repository. For closed-source software, *Data Processor* allows users to supply the repository URL. Once the source code is located, it identifies affected software configurations: vulnerable versions, platforms, or settings. For instance, CVE-2024-4340 affects all versions of "sqlparse" prior to `v0.5.0`. Accordingly, the latest affected version (`v0.4.4`) is retrieved using the GitHub API, and its *source code* is downloaded (as shown in Figure 2).

Vulnerability Information Extraction. After the vulnerable version of the project source code is downloaded, *Data Processor* extracts the following four key pieces of information from "cvelist", if available: (1) *CVE description*: A high-level summary of the vulnerability in the target project. (2) *CWE data*: CWE information, a categorization system for hardware and software weaknesses that associates the CVE with a specific vulnerability type, e.g., CWE-674 (Uncontrolled Recursion). (3) *Patch commits*: The `git-diff` of code changes in the target source code that fix the vulnerability. These commits help LLMs localize the issue, understand its root cause [12, 14, 22], and generate effective exploits. (4) *Security advisories and PoC*: We filter URLs referenced in the CVE entry by keywords (e.g., "security", "advisory", "bounty", etc.), and scrape their contents.

Output. The vulnerable version of a project's source code alongside its directory structure and CVE raw data.

3.1.2 Knowledge Builder ②. This agent is instructed to analyze, distill, and organize the raw information, extracted by the *Data Processor*, into a structured knowledge base while retaining essential details required to reproduce the given CVE. This includes extracting PoC or exploit instructions from security advisories, which are invaluable for accurately reproducing the vulnerability. For instance, CVE-2024-4340 has a PoC provided in its security advisory, and the patch commit helps understand the root cause by highlighting where/how the vulnerability was mitigated. As shown in Figure 2, the *Knowledge Builder* included both of these details in the knowledge base. In case an advisory does not include an exploit, the LLM agent is tasked with generating an overview of what an exploit might entail. Additionally, capturing the patch details helps to localize the root cause of the vulnerability [6, 12, 22] and aids in crafting an effective exploit. This knowledge base acts as the long-term memory for agents, helping them to effectively reproduce CVEs, by including only essential information in their context to avoid overloading the LLM's context window.

Output. A structured *CVE knowledge base* providing details of CVE, including its associated CWEs, affected software configurations, root causes (if provided in the security advisory), and essential information from security advisories and patch commits.

3.2 Builder

Once the knowledge base is populated and the vulnerable version of the project's source code is downloaded, the next objective is to build the project in a way that allows the exploit to be executed. Starting from the base VM described previously, the *Builder* autonomously customizes the environment for each target CVE by employing two *developer* LLM agents: the Prerequisite Developer *Agent*, which analyzes and plans the environment setup, and the Setup Developer *Agent*, which executes this plan to configure the vulnerable environment. As well as one *critic*, the Setup Critic *Agent*, which analyzes the logs of the Setup Developer and evaluates the setup for the project, allowing the system to correct its mistakes. Below, we discuss these in detail.

[1] "*cvelist*" is an automated pilot program for CVE submission through GitHub (https://github.com/CVEProject/cvelist)

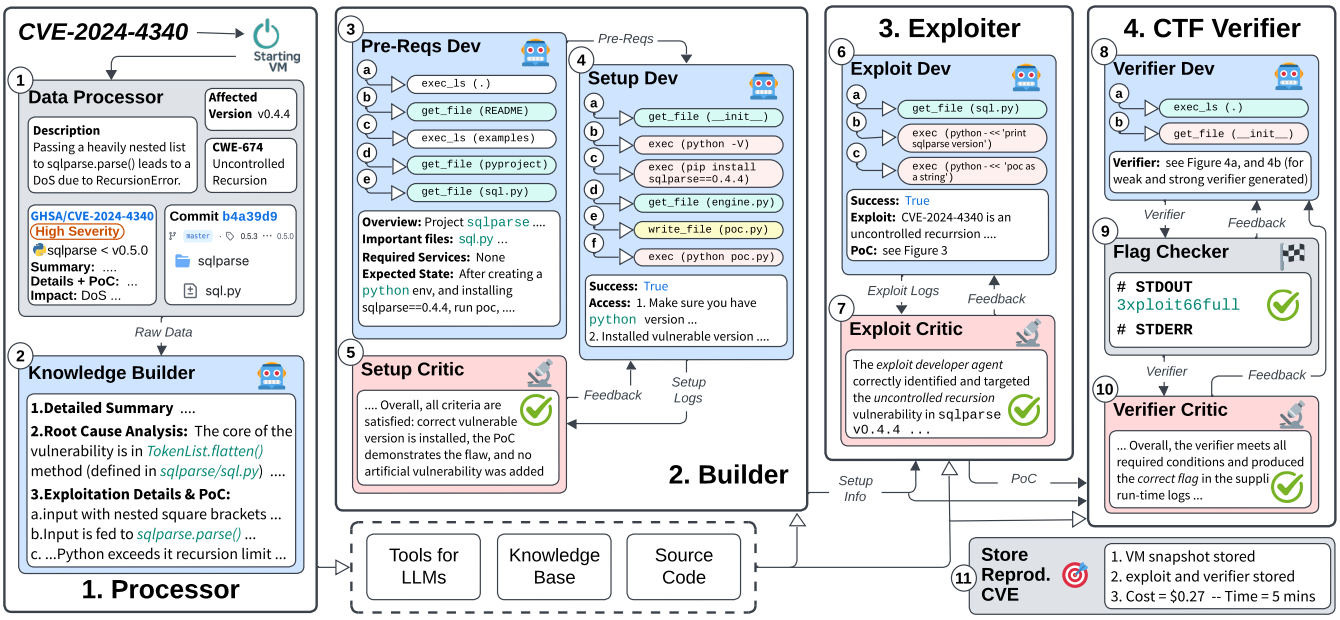


Figure 2: CVE-Genie architecture and an end-to-end example of workflow of reproduction for CVE-2024-4340, i.e., Denial of Service due to `RecursionError` in `sqlparse < v0.5.0`.

3.2.1 Prerequisite Developer Agent ③. We introduce this agent as an initial exploration and planning step before actually setting up the vulnerable project. This is done for three key reasons.
(1) Our preliminary experiments reveal that many projects contain inaccurate and incomplete information in their `README` files (e.g., vulnerable version `v1.2.7` of `lunary` for CVE-2024-5129, includes incorrect paths to `.env` files that are crucial for the setup). Without the *Prerequisite agent*, these details were consistently overlooked by LLMs, causing setup failures. Identifying and correcting such issues early reduces unsuccessful attempts, conserves the LLM's context window, and improves the likelihood of successful setup.
(2) During environment setup, the agent's job is to explore and analyze the source code to identify critical components and to build the project by executing a series of commands. For large projects, having to do both tasks with one agent can exceed the context limit and reduce efficiency (as demonstrated in the ablation study in Section 4.2). To address this, we delegate the exploration of the code base to a dedicated agent, the *Prerequisite Developer*, allowing the setup process to focus more on building the project.
(3) The *Prerequisite Developer* also defines the "expected state," i.e., when the project is fully set up and ready for exploitation. This state guides the *Setup Developer Agent* in verifying the correct configuration of the vulnerable environment. For example, in CVE-2024-4340 (Figure 2), the *Prerequisite Developer* explored the root directory and key files such as `README` and `sql.py` to understand the project context, then specified the expected state as having `sqlparse` version `v0.4.4` installed for the exploit to work correctly.

Tools. For this step, we only provide the LLM agent with access to read-only tools, i.e., `execute_ls_command`, and `get_file_by_name`, which the agent can use to traverse the code base and read files, because we do not want the agent to execute any commands or write to files in this phase (see Appendix D for tools details).

Output. (1) Detailed *overview* of the project, (2) *important files* for the setup, (3) *required services* and their configurations, and (4) *expected state* of the project after a successful setup.

3.2.2 Setup Developer Agent ④. The *Setup Developer* begins in the directory containing the vulnerable version's source code and focuses on configuring the project based on instructions from the *Prerequisite Developer*, exploring the codebase when necessary. For CVEs involving third-party libraries, our initial experiments showed that setup is greatly simplified by using package managers like `pip` or `npm` to install specific vulnerable versions instead of building from source. For CVE-2024-4340 affecting `sqlparse`, the *Prerequisite Developer* instructed the installation of version `0.4.4`, and the *Setup Developer* executed `pip install sqlparse==0.4.4` directly (command `4a`, Figure 2), resulting in a smooth setup. If the package manager fails, the *Setup Developer* falls back to building from source. The *Prerequisite Developer* also typically recommends running a basic PoC to confirm readiness before handing off (e.g., commands `4e` and `4f` in Figure 2).

Tools. For this step, we allow access to the following tools, i.e., `execute_ls_command`, `execute_linux_command`, `get_file`, `write_to_file`, and `set_environment_variable`.

Output. This agent returns a *final decision* whether the setup is successful or not. If successful, the output also includes instructions on how another agent can *access* the running project.

3.2.3 Setup Critic Agent ⑤. The *Setup Critic* evaluates whether the project setup performed by the *Setup Developer* is correct and complete, ensuring that the environment is properly configured

```
"""
PoC – CVE-2024-4340 (sqlparse < 0.5.0)

EXAMPLE THAT CRASHES v0.4.4
    python3 exploit.py 10000
"""
import sys
import sqlparse

def main() -> None:
    if len(sys.argv) != 2:
        print("Usage: python3 exploit.py <depth>")
        sys.exit(1)
    arg = sys.argv[1]
    try:
        depth = int(arg)
        payload = "[" * depth + "]" * depth
    except ValueError:
        payload = arg
    sqlparse.parse(payload)

if __name__ == "__main__":
    main()
```

(a) Exploit script for CVE-2024-4340. It constructs a deeply nested list payload and submits it to `sqlparse.parse()`, triggering uncontrolled recursion. The included comment documents a crashing input (depth 10,000) that reliably triggers the vulnerability.

```
Traceback (most recent call last):
  File "poc.py", line 20, in <module>
    sqlparse.parse(payload)
  ....
  File "sqlparse/sql.py", line 214, in flatten
    yield from token.flatten()
  [Previous line repeated 983 more times]
RecursionError: maximum recursion depth exceeded
```

(b) Execution traceback produced by the exploit, showing a `RecursionError` after excessive recursive calls within `sqlparse`.

Figure 3: PoC for CVE-2024-4340 in `sqlparse v0.4.4`, showing the exploit script and the resulting crash.

for the vulnerability described in the CVE knowledge base to be exploited. It also detects deceptive or shortcut behavior by the LLM, such as fabricating a mock project instead of performing a genuine setup, or injecting vulnerabilities into the codebase to simplify exploitation. These undesirable patterns are discussed in detail in Section 3.5.1, which informs the instruction design for the *Setup Critic*. Based on this, the *critic* is equipped to identify both functional and security-related flaws in the setup process.

Output. A comprehensive *analysis* of the setup logs, a binary *decision* on whether the setup is valid and complete, and actionable *feedback* for correcting issues or improving the setup if necessary.

3.3 Exploiter

After successfully configuring the vulnerable system, the next step in our pipeline is to generate an exploit for the given vulnerability. CVE-Genie accomplishes this using two LLM agents: the *Exploit Developer*, which develops the PoC for an exploit in the vulnerable environment, and the *Exploit Critic*, which analyzes the logs of the *Exploit Developer* and evaluates the exploit for the given vulnerability. Below, we discuss their detailed functionalities.

3.3.1 Exploit Developer Agent ⑥. The *Exploit Developer* is responsible for crafting and demonstrating exploits for vulnerabilities listed in the CVE knowledge base on pre-configured vulnerable systems. If a PoC is provided, the agent replicates and verifies it by executing the script with the appropriate triggering input before delivering the final, working PoC. For example, in CVE-2024-4340 (Figure 2), the PoC was available, so the agent validated the setup (6b), demonstrated the exploit (6c) by triggering `RecursionError` in `sqlparse/sql.py` (see Figure 3b), and submitted the verified PoC script (see Figure 3a). If no PoC or exploitation steps are available, the *Exploit Developer* iteratively analyzes the codebase using available tools to develop a functional exploit. Upon success, it produces a Python PoC script that accepts the crashing input via the command line, includes comments on the expected input format, and provides an example input. This ensures that the *CTF Verifier* can reliably validate the exploit without repeating the analysis.

Tools. This agent uses the same tools as the *Setup Developer*.

Output. The agent returns its *final decision* whether it considers the exploit is successful or not. If successful, it provides an *overview of the exploit* and a `Python` PoC script.

3.3.2 Exploit Critic Agent ⑦. Like the *Setup Critic*, the *Exploit Critic* evaluates the behavior of the *Exploit Developer* by analyzing its execution logs. Its primary goal is to determine whether the generated exploit is both valid and high-quality. It verifies that the exploit could plausibly succeed in a real-world setting; that it does not rely on manipulating the original setup in illegitimate ways; and that it avoids fabricated shortcuts, oversimplified logic, or assumptions that would not hold in practice.

To perform this assessment, the *Exploit Critic* leverages a predefined set of behavioral patterns and common failure cases, as detailed in Section 3.5.2. By comparing the exploit's behavior against these known issues, the critic can detect actions that are unfaithful to the original intent or technically incorrect. When flaws are identified, the *Exploit Critic* provides structured, actionable feedback to improve the robustness of the exploit in subsequent iterations.

3.4 CTF Verifier

After generating a candidate exploit for a given CVE, the next step is to automatically verify whether the exploit truly triggers the intended vulnerability. To achieve this, we derive a CVE-specific success signal for the intended vulnerability from the official CVE data (e.g., a `RecursionError`, an `AddressSanitizer` crash, an unauthorized access to a specific data item, etc.), and leverage it to create a security Capture-The-Flag (CTF) challenge. In this challenge, the hidden "flag" (i.e., a string of random characters or numbers) is revealed only if the exploit correctly satisfies the CVE-specific success signal. This allows CVE-Genie to use a standardized flag-retrieval

interface for verification, while grounding the underlying verification logic in the intended vulnerability semantics of the target CVE. The *CTF Verifier* uses three sub-modules to develop and validate such verifiers. The *Verifier Developer* agent module generates a candidate verifier, the *Flag Checker* module ensures that the exploit correctly retrieves a flag, and the *Verifier Critic* agent assesses the quality and effectiveness of the verifier and its success-signal verification logic for the given exploit.

3.4.1 Verifier Developer Agent ⑧. Following the CTF methodology, we prompt the *Verifier Developer* to create a general CTF-style `Python` verifier script for a given CVE and the candidate PoC. So, if the verifier script runs the PoC and it successfully triggers the vulnerability, it returns this flag (`3xploit66ful`). We prompt the *Verifier Developer* to format the verifier script in three structured steps:

(1) *Pre-Setup*: Prepares the necessary inputs for the exploit, sets up the environment, and prepares a flag to return if the exploit correctly triggers the vulnerability.
(2) *Exploit Execution*: Runs the provided PoC script to trigger the vulnerability. We make sure that the agent cannot modify the PoC script to avoid contamination.
(3) *Post-Setup*: Verifies the success of the exploit, and if successful, the script returns the flag.

This verifier architecture provides consistent and reliable exploit validation across a broad spectrum of vulnerability classes. It supports vulnerabilities with directly observable failure signals, such as a `RecursionError` in CVE-2024-4340 or an `AddressSanitizer: heap-buffer-overflow` in CVE-2024-10525, as well as higher-level logic flaws. For example, in the case of the missing-authorization flaw in CVE-2024-5129 (see Appendix D), the verifier first inserts a legitimate dataset with a known UUID, then executes the exploit by issuing an unauthenticated deletion request targeting that UUID, and finally confirms successful exploitation by verifying that the dataset has been removed from the database.

Tools. This agent follows the same read-only tools as the *Prerequisite Developer* agent, as in this step we don't want the model to execute anything, and we do all validations using the *Flag Checker* and the *Verifier Critic*.

Output. A custom `Python` *verifier* for the given CVE's exploit/PoC.

3.4.2 Flag Checker ⑨. Before passing the generated verifier to the *critic* agent, the *Flag Checker* executes it with the exploit script to ensure that it runs without errors and produces the expected output, i.e., the correct flag. For instance, figure 2 shows that the correct flag `3xploit66ful` is released when executing the verifier scripts generated for CVE-2024-4340. If a script fails due to a runtime error or if the flag is missing from the output, the system assumes that the verifier is incorrect. Feedback is then sent to the *Verifier Developer* to revise the script. This process is repeated until a maximum of five attempts is reached, with each attempt ensuring functional correctness of the verifier script.

3.4.3 Verifier Critic Agent ⑩. Once the verifier and exploit scripts pass functional validation via the *Flag Checker*, they are further examined by the *Verifier Critic*, which performs a similar evaluative role as the other critic agents, focusing specifically on the response

```python
import os, subprocess, sys
FLAG = "3xploit66full"

def pre_setup():
    # Configuring PYTHONPATH

def run_exploit():
    cmd = [sys.executable, "exploit.py", "10000"]
    return subprocess.run(cmd, timeout=20)

def post_exploit(proc):
    output = (proc.stdout or "") + (proc.stderr or "")
    return proc.returncode != 0 and "RecursionError" in
    ↪ output

if __name__ == "__main__":
    pre_setup()
    try: result = run_exploit()
    except subprocess.TimeoutExpired: sys.exit(1)
    if post_exploit(result):
        print(FLAG)
```

(a) Attempt 1 – Weak verifier script: `pre_setup()` lacks a check for the vulnerable `sqlparse` version. The exploit runs with a `20s` timeout, risking missed `RecursionErrors` past the limit. Verification relies only on detecting `RecursionError` in output, which is trivially spoofed.

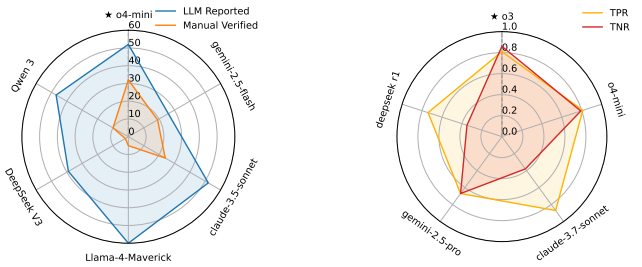
```python
import os, sys, importlib, traceback
FLAG = "3xploit66full"

def pre_setup():
    assert sqlparse.__version__ < "0.5.0"

def run_exploit_inprocess():
    sys.argv = ["exploit.py", "10000"]
    try:
        importlib.import_module("exploit").main()
        return "no_exception", ""
    except RecursionError: return "recursion_error",
    ↪ traceback.format_exc()
    except: return "other_exception",
    ↪ traceback.format_exc()

def exploit_succeeded(status, tb):
    return status == "recursion_error" and
    ↪ "sqlparse/sql.py" in tb

if __name__ == "__main__":
    pre_setup()
    status, tb = run_exploit_inprocess()
    if exploit_succeeded(status, tb):
        print(FLAG)
```

(b) Attempt 2 with *Verifier Critic*'s feedback – Improved verifier: Validates project version `sqlparse < 0.5.0`, runs the exploit in-process to avoid timeout issues, and explicitly checks for a genuine `RecursionError` with `traceback` originating from `sqlparse/sql.py`, preventing spoofed or misleading outputs.

Figure 4: Verifier scripts for exploit (in Figure 3) for CVE-2024-4340 corresponding to the run in Figure 2, illustrating the progression from a weak to a robust verifier based on *Verifier Critic* feedback.

Table 3: Studied LLMs.

Model	# Params	Max Inp	Max Out	Reasoning	Open Src	Cutoff
o3	n/a	200k	100k	✓	✗	05/2024
o4-mini	n/a	200k	100k	✓	✗	05/2024
Claude 3.7 Sonnet	n/a	200k	64k	✓	✗	11/2024
Claude 3.5 Sonnet	175B	200k	8k	✗	✗	04/2024
Gemini 2.5 Pro	n/a	1M	65k	✓	✗	01/2025
Gemini 2.5 Flash	n/a	1M	65k	✓	✗	01/2025
Llama 4 Maverick	400B	1M	n/a	✗	✓	08/2024
Qwen 3	235B	128k	64k	✓	✓	06/2024
Deepseek V3	671B	64k	8k	✗	✓	07/2024
DeepSeek R1	671B	128k	8k	✓	✓	07/2024



(a) Developer Agent **(b) Critic Agent**

Figure 5: Builder Optimal LLM Evaluation

produced by the *Verifier Developer*. It examines whether the verification process was properly carried out, without assumptions, omissions, or unreliable heuristics. Using the behavioral issues identified in Section 3.5.3, the agent flags incomplete or misleading verification strategies and provides feedback on how the verification could be made more robust or accurate. For instance, Figure 4 shows a detailed example of verifiers generated for CVE-2024-4340 with weak and bypassable verification criteria, which were corrected using the *Verifier Critic*'s feedback.

3.4.4 Store Reproduced CVE (11). If the given CVE passes the *Verifier Critic* check, CVE-Genie marks it as reproduced and stores the VM snapshot for the vulnerable environment as well as the *exploit* and *verifier* scripts. Moreover, CVE-Genie also stores the metadata, such as LLM cost, time spent, and all agents' conversations.

3.5 Selecting LLMs and Engineering Prompts for CVE-Genie's Optimal Performance

Upon finalizing the design of CVE-Genie's architecture, we perform a systematic LLM model and prompt selection process to identify the best performing LLMs and prompt integrations. We evaluate ten state-of-the-art LLMs (Table 3) across CVE-Genie's three core modules, *Builder*, *Exploiter*, and *CTF Verifier*, using a diverse set of 60 post–knowledge-cutoff CVEs, while simultaneously engineering and refining task-specific prompts for each module.

Dataset (D_S). To evaluate LLMs at reproducing vulnerabilities with CVE-Genie, we construct a small, diverse dataset (D_S) of 60 CVEs published after the knowledge cutoff of the LLMs under study (January 2025), listed in Table 3. For representativeness and diversity, we select 40 CVEs from the top 25 most dangerous CWE [38] categories, and the remaining 20 CVEs are randomly sampled from the other CWE types. The resulting D_S spans 44 CWE categories, 18 programming languages, and 56 software projects.

Methodology. CVE-Genie is composed of four modules. The *Processor*'s LLM-based component, the *Knowledge Builder*, only performs a trivial task of summarization, so we assign a lightweight LLM (`o4-mini`) to it. In contrast, the remaining modules, i.e., *Builder*, *Exploiter*, and *CTF Verifier*, require strong security vulnerability reasoning and SWE capabilities. Thus, for each of these modules, we systematically select the best performing LLM for "developer" and "critic" tasks using the following 5-step procedure:

(1) Candidate Selection: For each module, we identify subsets of LLMs from Table 3 as potential developers and critics, based on prior evidence of their capabilities, as well as time and cost constraints.

(2) Developer Evaluation: Each candidate developer LLM is run for the given module, and its outputs are manually scored by the authors. The highest-scoring LLM is chosen as the developer.

(3) Critic Prompt Design: We analyze common mistakes made by developers and use these insights to design a module-specific critic prompt, enabling the critic to effectively detect those mistakes.

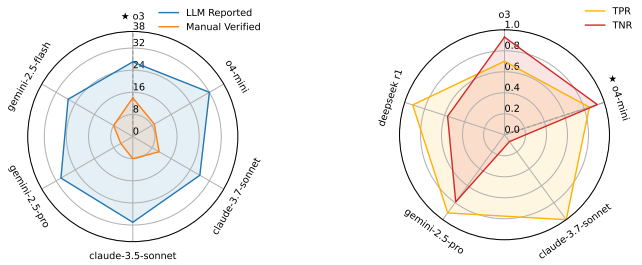
(4) Critic Evaluation: Each candidate critic LLM is evaluated on true positive rate (TPR) and true negative rate (TNR). When TPRs are equal, we prefer the model with the higher TNR, and vice versa. Then then critic with the best trade-off is selected.

(5) Feedback Loop Assessment: We rerun the chosen best developer–critic pair for the given module, and evaluate the correctness of the critic's feedback and the developer's capability to correctly address the critic's feedback.

3.5.1 Builder. In this section, we systematically select the most optimal configuration of LLMs and prompts for the *Builder*.

Candidate Selection. Setting up a project from scratch is the most time-consuming part of CVE-Genie, as advisories and CVEs give no setup instructions. To address this, we use two *developer* agents: one for exploring the project and another for setting it up (see Section 3.2). This requires extensive tool calls and command executions, making reasoning-heavy models like `o3` or `claude-3.7-sonnet` slow and costly ($10–20 per CVE). Therefore, we select fast, cost-effective LLMs with sufficient context capacity, `o4-mini`, `claude-3.5-sonnet`, `gemini-2.5-flash`, `llama-4-maverick`, `deepseek-v3`, and `qwen3`, as *developer* agent candidates. For *critic* agents, we opt for stronger reasoning models such as `o3`, `o4-mini`, `claude-3.7-sonnet`, `gemini-2.5-pro`, and `deepseek-r1`, as the critic's task requires only one call and incurs minimal overhead.

Developer Evaluation. We ran CVE-Genie up to the *Setup Developer* stage (4) on CVEs from D_S and manually evaluated each developer LLM using the following correctness criteria. (1) For each CVE, we first identified the official installation documentation for the software's vulnerable version and, based on it, established both the expected final build state and the sequence of commands required for installation. (2) We then reviewed the logs of runs where the LLM reported successful setup, comparing the executed commands



(a) Developer Agent **(b) Critic Agent**

Figure 6: Exploiter Optimal LLM Evaluation

against the documented installation process to assess alignment and completeness. (3) Finally, we validated the resulting environment: for libraries or binaries, we confirmed the vulnerable version by invoking the corresponding `-version` (or equivalent) command (e.g., for CVE-2025-1215 we verified that `vim v9.1.1096` was installed), while for server-based software we performed a health check to ensure the service was running and accessible at the expected port.

As shown in Figure 5a, `o4-mini` emerged as the best developer LLM for the *Builder*, successfully setting up 32 out of 60 CVEs projects, while making an average of 20 tool calls per CVE. In contrast, `claude-3.5-sonnet` mostly gave up on the given setup task, and models like `gemini-2.5-flash` and open-source LLMs were unreliable: `Qwen 3` often described setup steps without executing them using tool calls, while `gemini-2.5-flash` issued tool calls with frequent syntax errors, leading to failed setups.

Critic Prompt Design. From analyzing *developer* LLM setups, we observed three recurring mistakes: (1) when setup fails, the LLM builds a simplified mock-up substitute project instead; (2) for `pip`/`npm` packages, it installs the latest version instead of the specified vulnerable one; and (3) when a server is required, it assumes commands like `npm run dev` succeed without verifying server health. We designed the critic prompt to detect these errors while reviewing *Setup Developer* logs.

Critic Evaluation. As shown in Figure 5b, the `o3` model achieves the highest TPR and TNR when evaluating *Setup Developer*'s execution logs. Most of its false negatives involved setups requiring external hardware, which *Exploit Developer* cannot exploit anyway. Some valid setups were also mistakenly rejected due to limited post-setup checks. Given its strong balance of recall and precision, we choose `o3` as the critic model for the *Builder*.

Feedback Assessment. We evaluated CVE-Genie up to *Setup Developer* stage ④, using the best-performing developer (`o4-mini`) and critic (`o3`) models. Minor issues flagged by the *critic* (e.g., limited verification) were usually resolved in one iteration. However, fundamental issues (e.g., mock-up version of the project) persisted even after five iterations. We therefore limit feedback to one iteration, which addresses minor issues without excessive overhead.

3.5.2 Exploiter. In this section, we systematically select the most optimal configuration of LLMs and prompts for the *Exploiter*.

Candidate Selection. Generating a working exploit from a given CVE in a pre-configured vulnerable environment is a crucial task for reproducing CVEs. However, the *Exploiter* requires less code base exploration than the *Builder*, as advisories often provide PoCs, exploitation steps, or vulnerability details. Thus, we include expensive reasoning models like `o3`, `claude-3.7-sonnet`, and `gemini-2.5-pro` in the candidates list of developers. To mitigate instability seen in open-source models (Section 3.5.1), we employ only closed-source LLMs as developers, while using the same critic LLMs as the *Builder*.

Developer Evaluation. For the 32 successful setups from Section 3.5.1, we ran CVE-Genie up to the *Exploit Developer* stage ⑥ and manually evaluated each developer LLM against the following correctness criteria: (1) we checked whether the LLM executed a PoC or command sequence that demonstrated the exploit, (2) we verified that the vulnerability was clearly triggered (e.g., uncontrolled recursion in CWE-674 resulting in a `RecursionError`), (3) we assessed fidelity to the CVE description, (4) we ensured no mock-up environments or fake exploits were used, and (5) we confirmed the exploit script followed the expected format (see Section 3.3.1). Among all models, `o3` consistently performed best, successfully reproducing the highest number of exploits (13 out of 27) and demonstrating them end-to-end within the actual vulnerable environment. In contrast, `o4-mini` often produced incorrect or mock-up exploits and struggled with runtime errors. While `claude` and `gemini` models showed strong capabilities in demonstrating exploits, they lacked versatility across different vulnerability types. Hence, we select `o3` as the *Exploit Developer*.

Critic Prompt Design. We examined the failed exploit attempts and categorized the recurring error patterns, which directly mirrored our manual verification criteria (e.g., lack of exploit demonstration, unclear vulnerability trigger, deviation from CVE description, reliance on dummy/fake exploits, or incorrect script formatting). We then incorporated these categories into the *critic* agent's prompt, enabling it to automatically detect these issues for *Exploit Developer*.

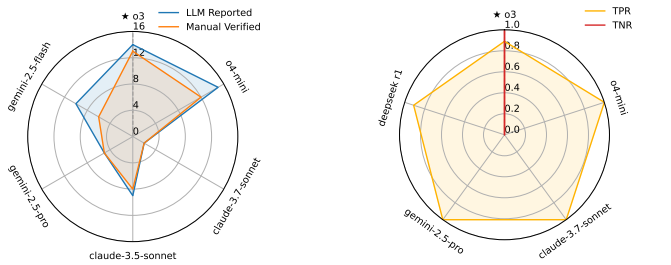
Critic Evaluation. As shown in Figure 6b, `o4-mini` achieved the best balance of recall and specificity. In contrast, `o3` was overly strict, rejecting correct exploits and focusing too much on code formatting, while `gemini-2.5-pro` frequently overlooked incomplete exploit verification. `Claude-3.7-sonnet` was too lenient, accepting nearly all exploits. Based on this evaluation, we selected `o4-mini` as the most effective critic model for the *Exploit Critic*.

Feedback Assessment. We evaluated feedback effectiveness by running CVE-Genie on 32 correct setups with the top developer (`o3`) and critic (`o4-mini`). This yielded valid exploits for 16 CVEs, of which 3 were fixed based on the critic's feedback. As in Section 3.5.1, the developer mainly resolved minor issues (e.g., incomplete verification or missing log triggers), while more complex flaws (e.g., incorrect exploit verification) persisted. To balance usefulness and cost, we therefore restrict feedback to a single iteration.

3.5.3 Verifier. In this section, we systematically select the most optimal configuration of LLMs and prompts for the *CTF Verifier*.

Candidate Selection. Like exploit generation, creating a verifier also demands deep understanding of security vulnerabilities to ensure the exploit is successfully triggered. Therefore, we use the same developer and critic candidates as in the *Exploiter*.

Developer Evaluation. For the 16 valid exploits from Section 3.5.2, we ran CVE-Genie up to the *Flag Checker* stage ⑨ and manually



(a) Developer Agent **(b) Critic Agent**

Figure 7: Verifier Optimal LLM Evaluation

evaluated each developer LLM against three correctness criteria: (1) whether the verifier script followed the required format (see Section 3.4.1), (2) whether it executed the provided exploit script without modification, and (3) whether its verification logic was precise and reliable (see Figure 4 for a detailed example).

As shown in Figure 7a, the `o3` model generated 13 valid verifiers out of 14, outperforming all others. While `o4-mini` achieved comparable numbers, its verifiers more often relied on weak logic. Other models struggled with functional correctness and error recovery. We therefore selected `o3` for the *Verifier Developer* due to its consistent one-shot success in producing correct verifiers.

Critic Prompt Design. We analyzed failures in verifier generation and found that the recurring error types matched our verification criteria (e.g., incomplete checks, incorrect validation logic, or misuse of the target environment). We used these categories to design the critic agent's prompt, enabling it to automatically flag such errors when evaluating verifiers produced by the *Verifier Developer*.

Critic Evaluation. As shown in Figure 7b, only the `o3` model successfully identified critical but subtle issues in the verifier scripts, such as weak or flawed verification logic (Figure 4). Other models provided weak or no critique, making `o3` the most effective critic for the *CTF Verifier*.

Feedback Assessment. We evaluated CVE-Genie on 16 correct exploits from the *Exploiter*, generating accurate verifiers for 15, of which 4 were improved by the critic's feedback. The maximum number of feedback iterations needed to resolve functional or critique-based issues was 4; we therefore allow up to 5 retries for both the *Flag Checker* and the *Verifier Critic*.

3.5.4 Time and Cost Constraints. In our evaluation of CVE-Genie, we found that successfully reproduced CVEs typically required about $2 and 18 minutes on average, with the most resource-intensive case costing $6 and taking 45 minutes. Failed reproductions, on the other hand, averaged up to $4 and 35 minutes. To balance flexibility with cost control, we set a per-CVE budget cap of $5 and a maximum runtime of 45 minutes for all experiments.

4 Experimental Methodology

In this section, we conduct a four-part evaluation of CVE-Genie: (1) a baseline study assessing consistency and efficiency (Section 4.1); (2) an ablation study to assess the impact of key components of CVE-Genie on its scalability and effectiveness (Section 4.2); (3) robustness testing under incomplete CVE information (Section 4.3); and (4) large-scale evaluation on 841 CVEs (Section 4.4). These evaluations are guided by the following research questions:

- **RQ1:** Does the performance of CVE-Genie vary over multiple runs of CVE reproduction?
- **RQ2:** Is the complex architecture of CVE-Genie necessary, or can standalone LLMs achieve comparable results?
- **RQ3:** Can CVE-Genie effectively reproduce CVEs with limited information, and which CVE report components are most crucial for reproduction?
- **RQ4:** Can CVE-Genie reproduce a broad range of CVEs across diverse CWEs, languages, and projects?

4.1 Baseline Assessment

In this section, we evaluate the baseline performance of CVE-Genie and examine how its reproduction success evolves across multiple runs in the presence of LLM non-determinism.

Dataset. For this study we use the same D_S dataset (Section 3.5).

Methodology. Due to the non-deterministic nature of LLMs, a single run may fail due to incorrect reasoning, while later attempts might succeed [34]. To account for this, we run CVE-Genie multiple times on D_S, using its most optimal configuration (Section 3.5). After each iteration, we store successfully reproduced CVEs and rerun on the remaining ones. This iterative approach helps maximize CVE reproduction and reveals CVE-Genie's convergence.

Results. We perform manual analysis of randomly selected 25 CVE reproduction runs, and observe that the variability in CVE-Genie's performance primarily stems from the *Builder* phase, which is more open-ended than the targeted exploit generation guided by CVE context. Project setup involves repository exploration (*Prerequisite Developer*) and command execution (*Setup Developer*), where environment-specific behaviors often cause failures. For example, one run failed because the *Setup Developer* modified the `PATH` variable, while a clean retry succeeded, highlighting the value of reattempting in a fresh environment. Since the *CTF Verifier* depends on the *Exploiter*, which in turn relies on the *Builder*, this initial project setup is a critical bottleneck. As a previously failed build, if later successful, can enable downstream modules to reproduce a new CVE (as illustrated in Figure 8). Over multiple runs, the number of successful project, exploit, and verifier builds showed consistent gains with convergence at run seven, indicating that most recoverable failures can be resolved within a few retries.

> **RQ1:** Due to the non-determinism of LLMs, CVE-Genie exhibits variability across runs, primarily during project setup and environment configuration. Once project setup succeeds, downstream exploit generation and verification are more stable and converge in a few retries.

4.2 Importance of CVE-Genie's Architecture

Having evaluated the selection of LLMs and prompts for CVE-Genie (Section 3.5), we now assess CVE-Genie's architecture via a systematic ablation study that isolates the impact of CVE-Genie's individual components on end-to-end CVE reproduction.

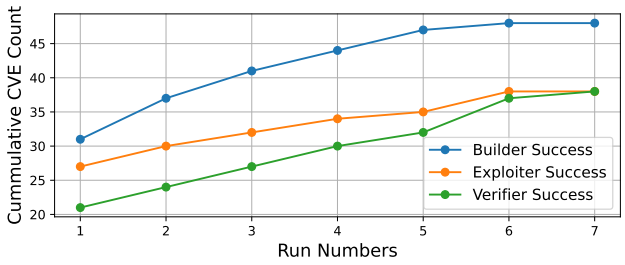


Figure 8: CVE-Genie performance over five runs on D_S.

Dataset. We perform this ablation study using 15 CVEs from D_S, each with complete information (i.e., CVE description, PoC, patch commit, and security advisory) and reproduced consistently in the first three iterations of baseline assessment (Section 4.1), and call this dataset D_R. By keeping the CVE context complete, we ensure that any performance differences arise solely from modifications

Methodology. As *Data Processor*, *Setup Developer*, *Exploit Developer*, and *Verifier Developer* are the four indispensable components of CVE-Genie for end-to-end CVE reproduction, for this evaluation of CVE-Genie's design we use the following five ablation settings:

(1) *No* Knowledge Builder*:* Bypass the Knowledge Builder and feed raw data directly to all agents.
(2) *No* Prerequisite Developer*:* Eliminate the *Prerequisite Developer*, allowing the *Setup Developer* to independently plan and execute the entire environment setup from scratch.
(3) *No Feedback Loops:* Enforce single-shot execution without iterative refinement, by removing all feedback loops.
(4) *No Critics:* Remove the *Verifier Critic* and treat a CVE as reproduced if the final scripts pass the *Flag Checker*.
(5) *Single Monolithic Agent:* Combine all modular agents into a single agent with access to the same tools, thereby evaluating the performance of a standalone LLM without CVE-Genie's agentic design and structured guidance (see Appendix D).

Results. In our ablation study (Table 4), removing the *Knowledge Builder* component reduces reproduction success to 9/15, as agents struggle to interpret unstructured advisory data and hit context window limits more often, which results in a 30% increase in exploit failures. Eliminating the *Prerequisite Developer* has a comparatively mild effect (13/15), but places additional burden on the *Setup Developer* as it has to do both codebase exploration and project setup at the same time, leading to 27% more tool-call limit violations. We observe this behavior particularly in complex repositories such as `CVE-2025-32389` in `Nameless`, a website software for Minecraft servers, with not very clear setup guides. Feedback loops emerged as the most critical design element, and without them the reproduction success dropped sharply to 5/15 (–67%). Similarly, removing critic agents reduced success to 8/15 while increasing false reproductions by 47%, highlighting their role in reliable end-to-end CVE reproduction. Finally, collapsing all components into a single monolithic agent resulted in zero successful reproductions, even when using advanced standalone LLMs (`o3`), underscoring the necessity of CVE-Genie's modular, feedback-driven, critic-guided design.

RQ2: Standalone LLMs are currently unable to reproduce CVEs end-to-end. CVE-Genie's modular, feedback-driven, critic-guided design is essential for reliable CVE reproduction, and removing any component significantly degrades performance (Table 4).

Table 4: Ablation study of CVE-Genie's design.

Ideal Case	Knw Build	Pre-Reqs	Feed-back	Critic Agents	Single Agent
15 / 15	9 / 15	13 / 15	5 / 15	8 / 15	0 / 15
-	↑ 30% Exploit Failure	↑ 27% Max Tool	↓ 67% Reprod Rate	↑ 47% False Reprod	-

4.3 Robustness to Loss of CVE Context

To evaluate how CVE-Genie performs when critical components of a CVE report are missing, we study its robustness under systematically reduced CVE context. We first present a detailed *qualitative* case study of a single CVE to illustrate how CVE-Genie adapts its reasoning under different context losses, and then conduct a *quantitative* ablation study to quantify the impact at scale.

4.3.1 Case Study for CVE-2024-4340. We begin by performing a detailed case study for CVE-2024-4340 reproduction under three levels of CVE context losses, and compare their reproductions against the ideal setting shown in Figure 2.

1) No Security Advisory / PoC. In this setting, all security advisories are removed from the CVE context, leaving only the developer patch commit[2] and the CVE description[3]. As a result, CVE-Genie relies heavily on analyzing the patch commit, exploring modified files, and inferring the PoC from the regression test added by the developer in `tests/test_regressions.py`. This test closely resembles the PoC described in the existing advisory[4], and the generated exploit payload (Figure 9a) is comparable to the ideal run shown in Figure 3a. Due to the absence of explicit PoC information, this run required additional exploration, resulting in a longer execution time (9 minutes) and higher cost ($0.67) compared to the ideal scenario.

2) No Patch. Here, the developer patch commit is removed while security advisories remain available. CVE-Genie primarily leverages the PoC provided in the advisory, which includes PoC exploit code, execution traces, vulnerable code, and root cause. The reproduction thus closely matches the ideal run in PoC exploit (Figure 9a), verification scripts, execution time, and cost.

3) Only CVE Description. In the most constrained setting, CVE-Genie has access only to the CVE description. Based on this limited information, it starts by identifying the implementation of `sqlparse.parse` and hypothesizes on how a deeply nested inputs

[2]**CVE-2024-4340 Developer Patch**: https://github.com/andialbrecht/sqlparse/commit/b4a39d9850969b4e1d6940d32094ee0b42a2cf03
[3]**CVE-2024-4340 CVE Description**: "Passing a heavily nested list to sqlparse.parse() leads to a Denial of Service due to RecursionError."
[4]**CVE-2024-4340 GHSA**: https://github.com/advisories/GHSA-2m57-hf25-phgg

```
payload = '[' * depth + ']' * depth
sqlparse.parse(payload)
```

(a) Security advisory/PoC or patch commit is not available.

```
payload = 'select' + '(' * depth + '1' + ')' * depth
sqlparse.parse(payload)
```

(b) Only CVE description is available.

Figure 9: PoC exploit payload generated by CVE-Genie for CVE-2024-4340 under varying levels of incomplete CVE context compared to the payload in ideal scenario in Figure 3a.

can trigger a recursive execution path leading to a `RecursionError`. During PoC exploit generation and verification, the first run failed to produce a payload that successfully triggered the `RecursionError`. However, a second run from scratch succeeded in generating a valid payload (shown in Figure 9b). This setting proved most challenging, requiring 20 minutes and $4.50 in LLM cost over two runs.

4.3.2 Evaluation at Scale. We perform an ablation study to quantify CVE-Genie's robustness to incomplete CVE context at scale.

Dataset (D_R). We perform this ablation with D_R dataset (Section 4.2). We intentionally select these "easier" CVEs as a controlled probe, since CVE-Genie can consistently reproduce them under full context. Hence, any performance degradation after removing specific information indicates that the missing context is genuinely critical. To simulate incomplete CVE context, we construct three variants of D_R by systematically removing: (a) security advisory/PoC, (b) patch commit, and (c) everything except CVE description.

Methodology. In this study, for each variant of D_R, we run CVE-Genie three times per CVE, iteratively.

Results. Removing the developer patch commit leads to a notable drop in reproduction success 10/15 (67%), even when a PoC is available. Patch commits provide essential localization signals by identifying relevant files and code paths. Without patch commits, the model must rely on broad codebase exploration, often resulting in inefficient search and budget exhaustion. We observed this behavior in CVE-2025-31481 reproduction, an authorization flaw in the PHP-based `API Platform Core`, where the absence of a patch caused CVE-Genie to explore unrelated components and fail to converge. Excluding security advisories reduces the success rate to 11/15 (73%). In these cases, failures are primarily attributable to environment setup and verification difficulties rather than exploit generation. Advisories often provide crucial contextual information, such as dependency versions, execution constraints, and validation guidance, that streamline reproduction, particularly when PoCs are not provided as executable code but rather as high-level textual descriptions. Despite these challenges, CVE-Genie maintains baseline robustness under minimal context. When only the CVE description is provided, it successfully reproduces 9/15 cases (60%), albeit with increased cost and runtime. Thus, while comprehensive CVE reports improve efficiency and reliability, meaningful reproduction remains feasible with severely limited information.

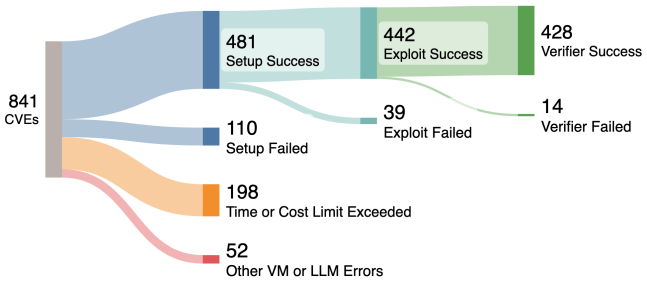


Figure 10: Breakdown of CVE-Genie's run on D_L

RQ3: CVE-Genie can reproduce CVEs under limited context. *Code-based PoCs* provide the strongest guidance, but in their absence, *patch commits* help localize vulnerabilities. Reproduction remains possible even with only *CVE descriptions*, but with lower success rates, higher cost, and increased time.

4.4 Large Scale Evaluation

Dataset (D_L). For our large-scale evaluation, we construct a dataset, (D_L), consisting of all CVEs published between June 2024 and May 2025 that include publicly available source code.

Selection Criteria. A CVE is included in D_L if at least one of the following conditions is satisfied: (1) the CVE entry contains a GitHub URL (e.g., a patch commit) in its references that links directly to the relevant source code, or (2) the CVE specifies affected software versions, and the source code corresponding to those versions is publicly accessible. No further filtering is applied based on operating system, CWE category, programming language, or software type: D_L includes every CVE from `cvelist` meeting the source-code-availability criterion, regardless of whether it targets Linux, Windows, or macOS, and regardless of vulnerability class. Additionally, because these CVEs were disclosed after the knowledge cutoff of the LLMs (`o3` and `o4-mini`) used in CVE-Genie, D_L ensures zero prior exposure during model training.

Representativeness. Despite the source-code-availability requirement, D_L remains representative of the broader distribution of vulnerabilities published in the same time period. Across all CVEs published in `cvelist` between June 2024 and May 2025, approximately 55% fall under the MITRE top 25 most dangerous CWEs [38]; D_L exhibits the same proportion (55%). Similarly, roughly 50% of CVEs in the broader population lack any PoC or exploit details in their advisory; in D_L, 49% lack a PoC. This close alignment indicates that the source-code-availability filter does not systematically bias the dataset toward particular vulnerability categories or toward CVEs with more available exploitation information.

Scope and Diversity. Overall, D_L comprises 841 CVEs spanning 186 CWEs, 29 programming languages, and 440 distinct open-source software (OSS) projects. The OSS projects cover a broad range of application domains, including web applications and backends (156), libraries and frameworks (147), CLI tools and binaries (28), cloud and DevOps systems (25), embedded and networking software (18),

operating systems and runtimes (17), AI/ML platforms (16), desktop applications (13), blockchain and cryptocurrency systems (7), mobile applications and SDKs (7), and security tools (6). In terms of vulnerability types, the majority of CVEs in D_L fall under injection and improper neutralization (29%), authorization and authentication (11%), resource management (8%), information exposure (6%), memory safety (3%), request forgery (3%), and others.

Methodology. We use the same methodology as Section 4.1, and run CVE-Genie on D_L three times, iteratively. We then manually audit a random sample of the resulting outcomes, consisting of 50 CVEs that passed the CTF verification and 50 CVEs that were reported as failed reproductions. For each sampled case, we manually review the builder, exploit, and verifier artifacts using the same correctness criteria defined in Sections 3.5.1, 3.5.2, and 3.5.3, and determine whether the reported outcome was correctly classified.

Results. CVE-Genie reproduced 428 CVEs out of 841 (Figure 10), spanning 22 programming languages, 141 vulnerability types, and 267 projects. To better understand performance and failure modes, we analyze the large-scale outcomes along two vulnerability settings (binary-heavy vs. Web/service), then separate failures by pipeline stage, and finally provide a qualitative analysis of verifier behavior and manual validation findings.

Binary-Heavy CVEs. CVE-Genie shows stronger binary exploitation results for library/framework targets, while desktop applications (18%), blockchain (8%), and mobile (29%) lag behind (see Appendix D). Successful reproductions relied on deterministic crash signals such as `AddressSanitizer` reports (e.g., Eclipse Mosquitto – CVE-2024-10525, MicroPython – CVE-2024-8946) or `SIGSEGV` (e.g., Assimp – CVE-2025-3160, Vim – CVE-2025-1215). However, without proper sanitizer instrumentation, some memory-safety bugs executed silently, producing no observable crash and failing verification. The absence of publicly available PoCs further limits reproduction; for instance, CWE-416 (use-after-free) achieves only 30% success, with zero PoCs available across all 10 cases (see Appendix D). Binary-heavy builds are also disproportionately affected by the 45-minute time constraint, as individual build retries can consume 10–15 minutes. We provide detailed end-to-end case studies for CVE-2024-10525 and CVE-2024-8946 in Appendix D; notably, for CVE-2024-8946 CVE-Genie did not have access to the PoC, yet it independently synthesized an exploit that produces the same `AddressSanitizer` crash as a human-authored PoC. Finer-grained CWE breakdowns by domain appear in Appendix D.

Web-Based CVEs. In Web and service domains, reproduction effectiveness varies substantially by vulnerability family. *DoS and resource exhaustion* CWEs are CVE-Genie's strongest family, CWE-1333 (ReDoS) reproduces at 79%, driven by signals such as RecursionError and timeout or memory usage thresholds (see Appendix D). *Injection flaws* show mixed results: SQLi (CWE-89) achieves 52% and XSS (CWE-79) 46%, but OS command injection (CWE-78) drops to 0% without a PoC. *Logic and access-control* vulnerabilities exhibit the strongest PoC dependence: CWE-287 (authentication bypass) achieves 100% with a PoC but only 36% without it, as these require stateful multi-step verification. A key differentiator within these domains is documentation quality: Web/backend applications often depend on external installation guides inaccessible on our offline VM, contributing disproportionately to build failures. We provide further per-domain CWE breakdowns in Appendix D.

Pipeline Failures. Of the 413 failed CVEs, 198 exhausted their cost or time budgets across all three iterative runs. Cost overruns account for ~80% of these, split between project builds (~41%), predominantly in Web applications, and exploit generation (~59%). Time overruns disproportionately affect binary-heavy targets, where compiled project builds consume significant time budget. Seven CVEs are fundamentally unsupported by our offline Ubuntu VM setup: four require Windows-specific filesystem behavior, two require macOS entitlements and XPC services, and one requires a full ASP.NET/IIS stack. Recurring failure patterns include dependencies on inaccessible documentation, unavailable Docker images, and missing PoCs for vulnerability families that depend on them (see Appendix D).

Manual Analysis. CVE-Genie's conservative validation, combining strict critics with the CTF verifier's CVE-specific success signals, produces reliable reproductions: our manual audit confirmed that all 50 sampled runs passing the CTF verifier were correct reproductions. However, when deterministic signals such as AddressSanitizer crashes or `RecursionError` were unavailable, especially in DoS-related CVEs, CVE-Genie relied on heuristic time and memory thresholds, e.g., a 3-second cutoff for CVE-2024-3651 and 300MB peak RSS for CVE-2025-27144, which are sensitive to system load and hardware configuration, meaning a legitimate DoS exploit could pass on one machine but fail on another. Moreover, our analysis of sampled failures revealed 12 false negatives cases where a correct exploit was incorrectly rejected due to lack of proof or details provided by the developer agents in the execution traces. This is consistent with the critic design study in Section 3.5, where all critics exhibit higher TNR than TPR, preferring to reject valid reproductions over accepting faulty ones.

RQ4: CVE-Genie reproduced 428 / 841 CVEs across 267 projects and 141 CWE types. Around half (46%) of successful reproductions had no prior PoC available, demonstrating that CVE-Genie can synthesize exploits from vulnerability descriptions and patch context alone.

5 Conclusion

In this paper, we present CVE-Genie, an automated framework for reproducing real-world vulnerabilities. CVE-Genie benefits downstream security research (see Appendix D), which suffers from a shortage of diverse, high-quality data for developing new tooling for automated vulnerability discovery and repair. CVE-Genie successfully reproduced around 51% (428) of 841 CVEs, with average $2.77 cost per CVE, across a wide variety of projects, CWEs, and programming languages. To the best of our knowledge, no other automated framework has demonstrated end-to-end CVE reproduction at this scale and cost-efficiency.

Acknowledgments

We gratefully acknowledge and thank Dave Aitel, Harold Nguyen, Xin Hu, and Ian Brelinsky (OpenAI), as well as Stijn Pletinckx and Lukas Dresel (University of California, Santa Barbara), for their valuable discussions and guidance that contributed to this work.

This material is based upon work supported by the National Science Foundation under grant no. 2229876 and is supported in part by funds provided by the Department of Homeland Security, and by IBM. This work is partially supported by a Google gift, an OpenAI Cybersecurity Grant (for Saad Ullah), the Australian Research Council's Discovery Projects funding scheme (project DP250101396), and a Google Research Scholar award (for Hammond Pearce).

References

[1] Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Sadhana Kumaravel, Matthew Stallone, Rameswar Panda, Yara Rizk, G P Shrivatsa Bhargav, Maxwell Crouse, Chulaka Gunasekara, Shajith Ikbal, Sachindra Joshi, Hima Karanam, Vineet Kumar, and Asim Munawar. 2024. Granite-Function Calling Model: Introducing Function Calling Abilities via Multi-task Learning of Granular Tasks. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, Franck Dernoncourt, Daniel Preoţiuc-Pietro, and Anastasia Shimorina (Eds.). Association for Computational Linguistics, Miami, Florida, US, 1131–1139. doi:10.18653/v1/2024.emnlp-industry.85

[2] Janice Ahn, Rishu Verma, Renze Lou, Di Liu, Rui Zhang, and Wenpeng Yin. 2024. Large Language Models for Mathematical Reasoning: Progresses and Challenges. arXiv:2402.00157 [cs.CL] https://arxiv.org/abs/2402.00157

[3] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2022. Dos and Don'ts of Machine Learning in Computer Security. In *31st USENIX Security Symposium*.

[4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *arXiv:2108.07732 [cs]* (Aug. 2021). arXiv: 2108.07732.

[5] Bandit. [n. d.]. Bandit Documentation. https://bandit.readthedocs.io/en/latest/

[6] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering* (Athens, Greece) *(PROMISE 2021)*. Association for Computing Machinery, New York, NY, USA, 30–39. doi:10.1145/3475960.3475985

[7] Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, Jie Wang, Xiao Cheng, Guangtai Liang, Yuchi Ma, Pan Bian, Tao Xie, and Qianxiang Wang. 2024. CodeR: Issue Resolving with Multi-Agent and Task Graphs. arXiv:2406.01304 [cs.CL] https://arxiv.org/abs/2406.01304

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, and Matthias Plappert. 2021. Evaluating Large Language Models Trained on Code. doi:10.48550/arXiv.2107.03374

[9] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. 2023. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses* (Hong Kong, China) *(RAID '23)*. Association for Computing Machinery, New York, NY, USA, 654–668. doi:10.1145/3607199.3607242

[10] DARPA. 2016. The Cyber Grand Challenge. https://www.darpa.mil/program/cyber-grand-challenge

[11] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 423–435. doi:10.1145/3597926.3598067

[12] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability Detection with Code Language Models: How Far Are We? arXiv:2403.18624 [cs]. doi:10.48550/arXiv.2403.18624

[13] D Dittrich and E Kenneally. 2012. *The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research.* Technical Report. U.S. Department of Homeland Security. doi:paper/2012_menlo_report_actual_formatted

[14] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 508–512. doi:10.1145/3379597.3387501

[15] Richard Fang, Rohan Bindu, Akul Gupta, and Daniel Kang. 2024. LLM Agents can Autonomously Exploit One-day Vulnerabilities. arXiv:2404.08144 [cs.CR] https://arxiv.org/abs/2404.08144

[16] Michael Fu, Chakkrit Kla Tantithamthavorn, Van Nguyen, and Trung Le. 2023. ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We?. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. 632–636. doi:10.1109/APSEC60848.2023.00085

[17] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*.

[18] Wenbo Guo, Yujin Potter, Tianneng Shi, Zhun Wang, Andy Zhang, and Dawn Song. 2025. Frontier AI's Impact on the Cybersecurity Landscape. *arXiv preprint arXiv:2504.05408* (2025).

[19] Hazim Hanif and Sergio Maffeis. 2022. VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection. In *International Joint Conference on Neural Networks (IJCNN)*.

[20] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* (2020).

[21] Junda He, Christoph Treude, and David Lo. 2025. LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision and the Road Ahead. *ACM Trans. Softw. Eng. Methodol.* (Jan. 2025). doi:10.1145/3712003

[22] Jingxuan He and Martin Vechev. 2023. Large Language Models for Code: Security Hardening and Adversarial Testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1865–1879. doi:10.1145/3576915.3623175

[23] Infer. [n. d.]. A tool to detect bugs in Java and C/C++/Objective-C code before it ships. https://fbinfer.com/

[24] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 1430–1442. doi:10.1109/ICSE48619.2023.00125

[25] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *ICLR*.

[26] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024. From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future. doi:10.48550/arXiv.2408.02479

[27] Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. CAMEL: Communicative Agents for "Mind" Exploration of Large Language Model Society. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 51991–52008.

[28] Hongwei Li, Yuheng Tang, Shiqi Wang, and Wenbo Guo. 2025. PatchPilot: A Cost-Efficient Software Engineering Agent with Early Attempts on Formal Verification. *arXiv preprint arXiv:2502.02747* (2025).

[29] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2022. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2022).

[30] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *Proceedings of Network and Distributed System Security Symposium* (2018). doi:10.14722/ndss.2018.23158

[31] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, and Yang Xiang. 2017. POSTER: Vulnerability Discovery with Function Representation Learning from Unlabeled Projects. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*.

[32] Bin Liu, Yanjie Zhao, Guoai Xu, and Haoyu Wang. 2025. LLM Agents for Automated Web Vulnerability Reproduction: Are We There Yet? arXiv:2510.14700 [cs.SE] https://arxiv.org/abs/2510.14700

[33] Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2024. How to Understand Whole Software Repository? arXiv:2406.01422 [cs.SE] https://arxiv.org/abs/2406.01422

[34] Rohin Manvi, Anikait Singh, and Stefano Ermon. 2024. Adaptive Inference-Time Compute: LLMs Can Predict if They Can Do Better, Even Mid-Generation. arXiv:2410.02725 [cs.CL] https://arxiv.org/abs/2410.02725

[35] Xiang Mei, Pulkit Singh Singaria, Jordi Del Castillo, Haoran Xi, Abdelouahab, Benchikh, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, Hammond Pearce, and Brendan Dolan-Gavitt. 2024. ARVO: Atlas of Reproducible Vulnerabilities for Open Source Software. arXiv:2408.02153 [cs.CR] https://arxiv.org/abs/2408.02153

[36] Metasploit. [n. d.]. RAPID7 metasploit. https://www.metasploit.com/

[37] Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. 2023. VulChecker: Graph-based Vulnerability Localization in Source Code. In *32nd USENIX Security Symposium*.

[38] MITRE. [n. d.]. 2024 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html
[39] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the Reproducibility of Crowd-reported Security Vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 919–936.
[40] Anh The Nguyen, Triet Huynh Minh Le, and M. Ali Babar. 2024. Automated Code-centric Software Vulnerability Assessment: How Far Are We? An Empirical Study in C/C++. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Barcelona, Spain) *(ESEM '24)*. 72–83.
[41] Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. 2021. CrossVul: a cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1565–1569. doi:10.1145/3468264.3473122
[42] NIST. [n. d.]. SARD. https://samate.nist.gov/SARD/
[43] NVD. [n. d.]. NVD - Home. https://nvd.nist.gov/
[44] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, and Shyamal Anadkat. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774
[45] OWASP. [n. d.]. Source Code Analysis Tools. https://owasp.org/www-community/Source_Code_Analysis_Tools
[46] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative Agents: Interactive Simulacra of Human Behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*. Association for Computing Machinery, New York, NY, USA, 1–22. doi:10.1145/3586183.3606763
[47] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2339–2356. ISSN: 2375-1207. doi:10.1109/SP46215.2023.10179324
[48] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Annibal, Alec Peltekian, and Yanfang Ye. 2021. CoTexT: Multi-task Learning with Code-Text Transformer. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog)*.
[49] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The Seven Sins: Security Smells in Infrastructure as Code Scripts. In *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*.
[50] Niklas Risse. 2023. Detecting Overfitting of Machine Learning Techniques for Automatic Vulnerability Detection. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 2189–2191. doi:10.1145/3611643.3617845
[51] Niklas Risse and Marcel Böhme. 2024. Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection. arXiv:2408.12986 [cs.CR] https://arxiv.org/abs/2408.12986
[52] Niklas Risse and Marcel Böhme. 2024. Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection. arXiv:2408.12986 [cs.CR] https://arxiv.org/abs/2408.12986
[53] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In *17th IEEE International Conference on Machine Learning and Applications (ICMLA)*.
[54] Yashar Talebirad and Amirhossein Nadiri. 2023. Multi-Agent Collaboration: Harnessing the Power of Intelligent LLM Agents. doi:10.48550/arXiv.2306.03314
[55] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. 2024. 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)*. IEEE Computer Society, Los Alamitos, CA, USA, 862–880. doi:10.1109/SP54263.2024.00210
[56] Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. 2026. CyberGym: Evaluating AI Agents' Real-World Cybersecurity Capabilities at Scale. arXiv:2506.02548 [cs.CR] https://arxiv.org/abs/2506.02548
[57] Laura Wartschinski, Yannic Noller, Thomas Vogel, Timo Kehrer, and Lars Grunske. 2022. VUDENC: Vulnerability Detection with Deep Learning on a Natural Codebase for Python. *Inf. Softw. Technol.* 144, C (April 2022), 15 pages. doi:10.1016/j.infsof.2021.106809
[58] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, Melbourne, Australia, 1482–1494. doi:10.1109/ICSE48619.2023.00129
[59] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *IEEE Symposium on Security and Privacy*.
[60] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv preprint arXiv:2210.03629* (2022).
[61] Zheng Yu, Ziyi Guo, Yuhang Wu, Jiahao Yu, Meng Xu, Dongliang Mu, Yan Chen, and Xinyu Xing. [n. d.]. PATCHAGENT: A Practical Program Repair Agent Mimicking Human Expertise. ([n. d.]).
[62] Daoguang Zan, Zhirong Huang, Ailun Yu, Shaoxin Lin, Yifan Shi, Wei Liu, Dong Chen, Zongshuai Qi, Hao Yu, Lei Yu, Dezhi Ran, Muhan Zeng, Bo Shen, Pan Bian, Guangtai Liang, Bei Guan, Pengjie Huang, Tao Xie, Yongji Wang, and Qianxiang Wang. 2024. SWE-bench-java: A GitHub Issue Resolving Benchmark for Java. arXiv:2408.14354 [cs.SE] https://arxiv.org/abs/2408.14354
[63] Yuntong Zhang, Jiawei Wang, Dominic Berzin, Martin Mirchev, Dongge Liu, Abhishek Arya, Oliver Chang, and Abhik Roychoudhury. 2024. Fixing Security Vulnerabilities with AI in OSS-Fuzz. arXiv:2411.03346 [cs.CR] https://arxiv.org/abs/2411.03346
[64] Mengyao Zhao, Kaixuan Li, Lyuye Zhang, Wenjing Dang, Chenggong Ding, Sen Chen, and Zheli Liu. 2025. A Systematic Study on Generating Web Vulnerability Proof-of-Concepts Using Large Language Models. arXiv:2510.10148 [cs.SE] https://arxiv.org/abs/2510.10148
[65] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2A: A Dataset Built for AI-Based Vulnerability Detection Methods Using Differential Analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.
[66] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems*.
[67] Yuxuan Zhu, Antony Kellermann, Dylan Bowman, Philip Li, Akul Gupta, Adarsh Danda, Richard Fang, Conner Jensen, Eric Ihli, Jason Benn, Jet Geronimo, Avi Dhir, Sudhit Rao, Kaicheng Yu, Twm Stone, and Daniel Kang. 2025. CVE-Bench: A Benchmark for AI Agents' Ability to Exploit Real-World Web Application Vulnerabilities. arXiv:2503.17332 [cs.CR] https://arxiv.org/abs/2503.17332
[68] Yuxuan Zhu, Antony Kellermann, Akul Gupta, Philip Li, Richard Fang, Rohan Bindu, and Daniel Kang. 2025. Teams of LLM Agents can Exploit Zero-Day Vulnerabilities. arXiv:2406.01637 [cs.MA] https://arxiv.org/abs/2406.01637

A Ethical Considerations

We study only CVEs already patched publicly on NVD, so neither our exploits nor our released artifacts aid attacks against unpatched systems. We release CVE-Genie to help defenders build reproducible benchmarks for vulnerability detection and patching tools; following the beneficence criterion of the Menlo Report [13], we judge this benefit to outweigh the risk of misuse. See Appendix D for the full statement.

B Open Science

CVE-Genie's source code, the reproduction runs of all reproduced CVEs (agent logs, intermediate artifacts, and final exploit and verifier scripts), and a web application to visualize them are available at https://github.com/BUseclab/cve-genie; the per-study experiment archives are at https://osf.io/dcej4/. See Appendix D for the full statement.

C Generative AI Usage

We used ChatGPT (OpenAI) for LaTeX formatting and language editing; all output was reviewed and approved by the authors.

D Extended Version

Material referenced above as "see Appendix"—tool specifications, applications, the monolithic-agent study, per-domain CWE breakdowns, three case studies, and the full Ethical Considerations and Open Science statements—appears in the extended version: https://arxiv.org/pdf/2509.01835.