Politecnico di Torino

Master's Degree in Cybersecurity

A.a. 2024/2025

Graduation Session December 2025

Design of an AI Agent for the Generation of Vulnerable Virtual Environments

Supervisors:

Prof. Danilo Giordano
Prof. Idilio Drago
Prof. Marco Mellia
Prof. Matteo Boffa

Candidate:

Gabriele Sannia

Abstract

The steady increase of new software vulnerabilities puts growing pressure on current cybersecurity systems. To improve detection and mitigation capabilities, security experts seek to discover novel attack patterns and understand how new vulnerabilities can be exploited. One way to do that, is to manually create in-vitro scenarios (e.g., virtual environments) to safely observe and log attack data, without exposing real systems to risks. However, making scenarios that accurately reproduce realistic conditions is a complex, time-consuming task that requires a wide range of skills in virtualization technologies, cybersecurity, and service configuration.

Recent developments in the Artificial Intelligence (AI) field and the continuous growth of Large Language Models (LLMs) have highlighted their potential to automate complex tasks. Applying these technologies to streamline the creation of in-vitro scenarios would enable security experts to save time and safely perform penetration testing, patch development, collect attack data, and analyse data on these environments.

Building on this idea, this thesis proposes an AI agent designed to automate the generation of vulnerable virtual environments. The LLM-automated workflow replicates the structured reasoning of a human expert and is divided into four sequential steps: (1) CVE validation, (2) information retrieval, (3) generation of the virtual environment, and (4) static vulnerability assessment. Particularly, the agent receives as input the identifier of the vulnerability, namely the CVE-ID (Common Vulnerabilities and Exposures Identifier); gathers the services required to reproduce the virtual environment; iteratively builds the environment through a build-and-test loop; and finally assesses whether the environment is vulnerable to the input CVE.

I systematically evaluate several LLMs (GPT-4o, GPT-5, and gpt-oss:120B) to automate the AI agent. Results on 100 CVEs show that the AI agent can develop a working virtual environment for 63% of the tested vulnerabilities, with 27% of them confirmed to be vulnerable to the input CVE. These preliminary results demonstrate the potential of AI agent in the automation process to aid the work of cybersecurity experts. Future works will need to improve the current architecture, analyse a larger set of vulnerabilities, and explore the benefits of a multi-agent framework.

Acknowledgements

Voglio ringraziare tutti i miei relatori per aver riposto la loro fiducia in me e per il grande aiuto che mi hanno dato nel portare avanti questo progetto, soprattutto nei momenti di difficoltà.

Table of Contents

List of Tables

List of Figures

Glossary

CVE	Common Vulnerabilities and Exposures
IDS	Intrusion Detection System
IPS	Intrusion Prevention System
AI	Artificial Intelligence
LLM	Large Language Model
SSL	Self-Supervised Learning
NLP	Natural Language Processing
PoC	Proof of Concept
RAG	Retrieval Augmented Generation
CTI	Cyber Threat Intelligence

SDK

Software Development Kit

GUI

Graphical User Interface

GPU

Graphics Processing Unit

WSL

Windows Subsystem for Linux

VM

Virtual Machine

API

Application Programming Interface

HTTP

Hypertext Transfer Protocol

URL

Uniform Resource Locator

Chapter 1

Introduction, Background & Related Works

1.1 Problem & Motivations

Tens of thousands of new vulnerabilities are discovered each year, with 40303 new vulnerabilities getting a **Common Vulnerabilities and Exposures** (CVE) identifier in 2024 alone [1]. Understanding how vulnerabilities are exploited is a complex task that requires to identify: **which** attack is being performed, **how** it is carried out, **who** is performing it, and **why**. Answering these questions demands **exhaustive data collection** for each attack.

Understanding attack patterns and how exploits are executed is essential to better **defend against future attacks**. The data gathered from such analyses can be used to **train Artificial Intelligence** (**AI**) **models** to recognize specific attack patterns or exploit signatures, and to **improve Intrusion Detection Systems** (**IDS**) and **Intrusion Prevention Systems** (**IPS**).

The problem lies in the **quality of the gathered data**. Few organizations are willing to share details about how their systems were compromised, as unsanitized data could expose **sensitive information** about the affected systems or users. This means that, companies, public institutions, and other entities that collect attack data are often reluctant to share it, leading to a **lack of reliable and representative attack datasets**.

This raises the question: **how can real-world attack data be gathered in a controlled and reproducible way?** Unfortunately, to fully understand the scope and impact of an attack, it must be allowed to continue, as early stopping prevents a complete analysis, which means that data gathering cannot be performed on any system. To address this challenge, security experts have developed several approaches, such as **honeypots** and **cyber ranges**, that allow them to collect realistic attack data safely. These solutions often rely on experimental scenarios (e.g., **virtual environments**) intentionally configured to be vulnerable to a well-know set of vulnerabilities.

Virtual environments offer clear advantages: they can be **easily monitored**, **safely contained** and **quickly deployed** depending on the type of attack data that needs to be gathered. Having a **rich and freely customisable** collection of vulnerable virtual environments can enable researchers, or even autonomous systems, to rapidly recreate complex experimental scenarios and study how threat actors interact with them.

Creating virtual environments that accurately reproduce real vulnerabilities is a **non-trivial task**. As demonstrated by *Mu et. al* in the paper "*Understanding the Reproducibility of Crowd-reported Security Vulnerabilities*" [2], performing these tasks manually is **very time-consuming** and requires a **wide range of skills** in virtualization technologies, cybersecurity, and in the service configuration, that even security experts may lack. Relevant information must be gathered from available public sources and weighted based on how reliable the source is. The vulnerable service version must be identified, and the whole environment has to be configured to ensure it is both **vulnerable and isolated**. Performing all these activities wastes precious time that security experts could otherwise dedicate to more important tasks like data analysis, penetration testing, and patch development.

Existing efforts address part of the problem but have limitations. Projects like *Vulhub* [3] depend largely on **manual effort** to create Docker-based virtual environments built around a specific CVE. More recent approaches, such as `CVE-Genie` [4], leverage the growth of **Large Language Models** (**LLM**) and their potential to automate complex tasks. `CVE-Genie` is an AI agent that utilizes the **source code** of the vulnerable project linked to the CVE to generate both a vulnerable environment and a corresponding exploit. However, the effectiveness of this solution is limited by the availability of open-source code and the existence of a publicly available exploit for validation - resources that are often inaccessible or proprietary.

This thesis combines elements from existing approaches to tackle the challenge of **automating the creation of vulnerable virtual environments** using an **AI agent**. The goal of the agent is to streamline the creation **safe and reliable experimental scenarios** for the observation, collection, and analysis of attack patterns. To achieve this, it leverages modern LLMs to automate tasks such as data gathering, summarization, code generation, validation, and debugging.

1.2 Methodology & Results

This project introduces an **Agentic Workflow** designated to automate the creation of a vulnerable virtual environment starting from a single CVE-ID. Taking inspiration from how a human developer approach the problem, the workflow is divided into four stages:

- **LLM Setup & Input Validation**: the agent verifies the existence of the provided CVE-ID by using the *MITRE*'s CVE Service API [5], ensuring that the given CVE is valid and that no unnecessary processing is performed.
- **Information Gathering**: the agent is tasked with collecting and summarizing information about the CVE by combining the knowledge of the LLM with the one extracted by analysing public web pages content. This data helps the agent identify which services are **causing the vulnerability**, and which are required to **make the virtual environment work** as intended.
- **Virtual Environment Creation**: the agent generates all files for to the virtual environment, tests it to ensure it functions properly, and iteratively fixes any detected errors.
- **Vulnerability Assessment**: the virtual environment undergoes **static vulnerability assessment** to check if it is vulnerable to the starting CVE.

Various AI Agent development frameworks were evaluated, and in the end, LangChain [6] was chosen. The workflow was tested using three LLMs:

- `GPT-4o` and `GPT-5`, hosted on OpenAI's servers
- `gpt-oss:120B`, running on the local SmartData cluster

The agent communicates with the LLM using OpenAI's API, while the Langfuse [7] platform is used to monitor **costs and token usage**.

Following the ideas of Vulhub [3], Docker was chosen as the underlying **virtualization technology** on which to base all the virtual environments that are created by the agent. Docker Desktop is used to **deploy and manage the virtual environments** generated by the agent, while Docker Scout is used to perform the **static vulnerability assessment** on them.

The only mandatory piece of information for the agent is the **CVE-ID string**, around which the virtual environments is built. The progress of each agent run is tracked through a series of **milestones**, which ensure that the final virtual environment is **functional** and **accurately configured**.

1.3 Thesis Organization

This thesis is organized on five chapters and an appendix:

- **Chapter 1, Introduction, Background & Related Works**: the first chapter - i.e., the current one - continues by exploring the theory and research behind the technicologies used in this project. Starting from presenting the historical evolution of Artificial Intelligence (AI), to a conceptual overview of Large Language Models (LLM), AI Agents and their applications in the context of cybersecurity.
- **Chapter 2, Designing an AI Agent**: provides a simple overview of the thought process that lies behind the decisions made while developing the AI agent, describing the main inspirations that greatly influenced its design.
- **Chapter 3, Implementation Details**: focuses on the details that make up the entire agentic workflow described in the previous chapter, providing also an overview on the technologies and frameworks used to develop the AI agent.
- **Chapter 4, Results & Evaluation**: analyses the results obtained while testing the AI agent on a group of vulnerabilities. The goal is to evaluate how well the agent performs when varying its initial parameters, analyse its workflow and the final virtual environments produced.
- **Chapter 5, Conclusion: Limitations & Future Works**: talks current limitations that afflict the AI agent, potential improvements that can be made to the counter these limitations and other future works such as integrating the agent in multi-agent frameworks.
- **Appendix A, Function Definitions**: contains the code of some of the most important functions used by the AI agent to perform its tasks.

1.4 From AI to LLM-driven Agents

Artificial Intelligence (**AI**) is one of the latest technological innovation of the twenty-first century. The main goal of AI is to create **advanced autonomous systems** that can **emulate human intelligence**, and **take decisions independently**. Nowadays AI is used by practically everyone and it is integrated in a wide range of research fields, spanning from the academic world and industrial sector, to government agencies and military. Modern AI systems have achieved **remarkable performance** in domains that were traditionally thought to require exclusively human capabilities, like language comprehension, visual perception, reasoning, and even creativity.

The term "**artificial intelligence**" was first conceived in 1956 during a conference at Darthmouth College as part of a proposal for a research project [8]. The project focused on studying whether it was possible to describe the aspects of learning and the features of intelligence with a level of detail such that a machine could **simulate** them. Moreover, concepts such as "**Neuron Nets**" (i.e., **neural networks**) and "**Self-Improvement**" (both of which are well-developed in modern research) are already introduced by this proposal as part of the problem.

Just a year later, in 1957, Frank Rosenblatt developed the **Perceptron** as one of the first attempts to create a machine capable of learning from data [9]. The Perceptron can be seen an **early form of neural network** designed to perform **pattern recognition**. Essentially, the Perceptron is the **simplest unit of a neural network**, it takes as input a set of features, applies an **activation function** to the weighted sum of the features and produces an output. Although the type of problems the Perceptron can solve is limited by its architecture, its development was significant because it introduced key ideas, such as the use of **adjustable weights** and **learning from data**, which became **central in neural network and machine learning research**, and demonstrated that machines could, in principle, learn from experience.

Machine Learning (**ML**) is a subset of the AI domain that focuses on algorithms that can learn the patterns training data and make decisions on new data based only on what it has seen in the past. The term "**machine learning**" was popularized by Arthur Samuel, an american computer scientist that worked for IBM, best known for its 1959 article "*Some Studies in Machine Learning Using the Game of Checkers*" [10] which described a software mechanism that, starting from a parameters whose exact weight are not well known, gradually adjusted those weights by comparing the outcomes of all game of checkers the machine played, so that the next move would be the most advantageous one.

Neural Networks (**NN**) are structures composed by different layers, each containing a set of nodes (or neurons), connected through links (or synapses). Their architecture is strongly inspired by how the human brain is organized, which also makes them highly versatile, meaning they can be used for a wide variety of tasks.

The breakthrough that lead to the popularization of NNs came in 1986, when David Rumelhart, Geoffrey Hinton, and Ronald Williams published the paper "*Learning Representations by Back-Propagating Errors*" [11], which applied the **backpropagation algorithm** to the field of NNs. Backpropagation provides a systematic way for neural networks to adjust their weights by propagating the error from the output layer backward through the network. The main advantage this application brought is that it allowed NNs to **learn complex, non-linear relationships**, which had previously been difficult or impossible to capture since then. This makes neural networks very effective for tasks where other ML algorithms struggle, such as **image recognition**.

Deep Learning (**DL**) is a subset of neural networks characterised by many (at least 4) interconnected neuron layers, also called **hidden layers**. By varying model weights and biases between individual neurons in adjacent layers, the **deep neural network** can be optimized to yield more accurate outputs. In 2006, Geoffrey Hinton publishes "*Learning Multiple Layers of Representation*" [12] which summarizes key breakthroughs in DL and outlines how, thanks to their **highly flexible structure**, deep neural networks are capable of identifying complex data patterns, giving them incredible performance and versatility.

DL models are most commonly trained through **supervised learning on labelled data** to perform regression and classification tasks. However, this learning paradigm is far from perfect:

- The model learns **only from from the data it has already seen** (i.e., the training data). This means that, if new classes or data values appear, it may not be labelled correctly.

- Classes can be underrepresented or overrepresented in the training data, making it harder for the model to learn spot anomalies and to generalize.

The biggest downside of supervised learning is given by the prohibitive cost and resources needed to acquire large datasets of labelled data (that has to be **validated by a human**), which are required by deep neural networks to reach their performance cap.

This has led to development of **self-supervised learning** (**SSL**), a learning technique that imitates supervised learning tasks by generating a **supervised signal** from the unlabelled data, allowing the model to **learn from the structure of the data** (i.e., without the need of labels).

1.4.1 Large Language Models

Pre-trained LLMs

The first-wave of LLMs was based on the **transformer architecture**, first introduced by *Vaswani et al.* in the paper "*Attention Is All You Need*" [13]. The main innovation of the transformer architecture is the **self-attention mechanism**, which allows to analyse long-term dependencies, and to solve the problem of gradient vanishing/explosion. Moreover, the architecture is parallelizable, which makes training faster.

SSL is the fundamental training paradigm behind modern **Large Language Model**s (**LLM**), a category of DL models trained on enormous amounts of data, comprising articles, entire web pages and code repositories. **Natural Language Processing** (**NLP**) is what allows LLMs to manage and understand all these data, thanks to a mechanism known as **tokens**. The process of absorbing knowledge and learning reasoning patterns from this general-purpose **unlabelled** dataset is named **pre-training**. It is a very expensive task, that requires requires thousands of GPUs and large amounts of money. The most important advantage gained by pre-training an LLM, is that it can be **fine-tuned** to perform well on a specific task. Fine-tuning a model means to train it further (using the weight learned during pre-training as a starting point), on a task-specific **labelled** dataset. This is drastically different from previous training methods like **end2end training**, which instead have to rely on **impractically large labelled datasets**, to achieve the same results.

The flexibility of fine-tuning allowed researchers to introduce the concept of **few-shot learning** and to study its influence on LLMs with various sizes. Few-shot learning **fixes the weights of the model** and provides it with a number of **demonstration** about how to solve a related tasks (e.g., the model might be prompted to translate a sentence in English, while it is provided with a demonstration of how to translate other sentences in English). *Brown et al.* explain in the paper "*Language Models are Few-Shot Learners*" [14] how few-shot learning enables LLMs to **perform well on new tasks which were not included in the training corpus**, just by providing a handful of demonstrations. When applied to large LLMs (like `GPT-3`, with its 175 billion parameters), the accuracy of models subject to few-shot learning is **similar** to the one achieved by fine-tuning the model.

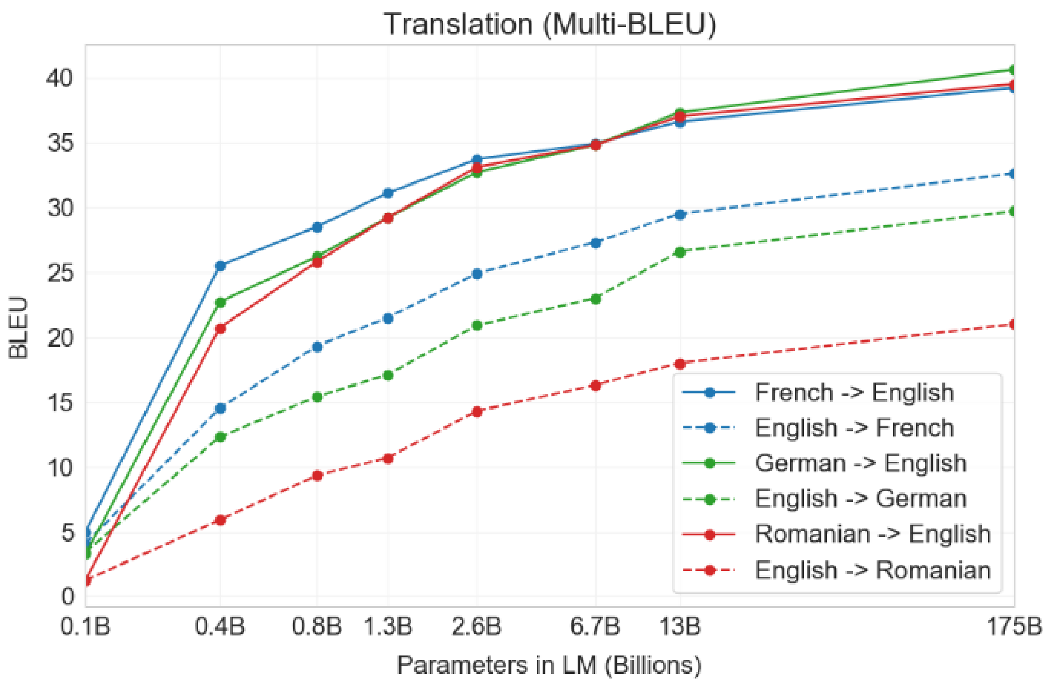


Figure 1.1: Translation Task, BLUE Performance and LLM Size Correlation. Source: *Brown et al.* [14]

The BLEU scale measures the performance of a model on task involving the translation of sentences in various language. Figure 1.1 shows the results obtained by training LLMs with varying parameter size trained with the **few-shot learning** approach. These results show how few-shot translation performance increases with the number of parameters of the model for all tasks. Unfortunately, it is not always possible to provide the model with enough demonstration to perform few-shot learning, which heavily limits its utility.

Instruction-tuning

Zero-shot is another learning methodology that works just like few-shot, expect this time **no demonstrations** are given to the model (e.g., the model is tasked with translating a sentence in English). This learning methodology is the most **convenient** one, as it can be used even when providing a demonstration is impossible, but it is also a much more **challenging** scenario for the model, causing a drastic performance drop, with respect to few-shot learning.

The paper "*Finetuned Language Models Are Zero-Shot Learners*" [15], by *Wei et al.*, explores how the zero-shot performance of the LLM can be improved via **instruction-tuning**, a learning methodology where the LLM is fine-tuned on a collection of task-specific datasets that are **described via instructions**. As observed by *Ouyang et al.* in the paper "*Training language models to follow instructions with human feedback*" [16], another way to improve the performance of a model, both for zero-shot and few-shot tasks, is to **incorporate human feedback** into the fine-tuning feedback loop. Models that are trained with this approach are **better at following instructions**, show improved truthfulness and align better with user goals. The paper also shows how making an LLM bigger does not automatically makes it better at following the given tasks, as demonstrated by comparing the outputs of the 1.3B parameters `InstructionGPT` model, with the outputs of the 175B parameters `GPT-3` model.

LLM Limitations

AI models like `GPT-3` and `InstructionGPT` can present misleading or even false information as factual. These events, named **hallucinations**, happen when the model detects non-existent patterns in its data. As of today there is not a clear way to prevent hallucinations, therefore AI generated content must always be validated before use.

The **context window** of an LLM is the amount of tokens that the model can process and memorize at any given time. A larger the context window, allows the model to evaluate longer inputs (e.g., longer conversations, extended documents, more files, bigger images, etc.) and increases the amount of relevant details in the output. If the context window of the model is too small to perform a task or to analyse a specific input, various solutions are possible:

- tasks can be split into more specialised sub-tasks
- inputs can be either truncated or summarized

However, it is important to understand that simply enlarging the context window of an LLM does not guarantee proportionally better results. Studies, such as the ones conducted by *Liu et al.* in the paper "*Lost in the Middle: How Language Models Use Long Contexts*" [17], show that positioning of relevant information in the middle of the context window can significantly degrade the performance of the model, emphasizing how LLMs can struggle to make use of relevant information for long inputs.

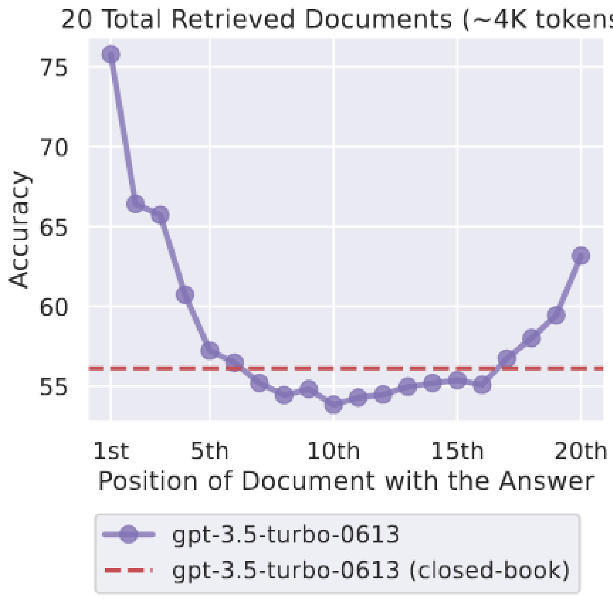


Figure 1.2: `GPT-3.5-Turbo` Accuracy Variation, Correlation with Information Position in the Context Window. Source: *Liu et al.* [17]

Figure 1.2 shows that `GPT-3.5-Turbo` performs better when the relevant information is placed at the very beginning of the context window (**primacy bias**), or at its end (**recency bias**). As revealed by *Huang et al.* in the paper "*TrustLLM: Trustworthiness in Large Language Models*" [18], in an attempt to solve, or at least mitigate the **lack of transparency and interpretability of LLMs**, some models (both open-source and especially proprietary ones) may be overly calibrated towards exhibiting trustworthiness, to the extent that they mistakenly treat benign prompts as harmful and consequently not respond, compromising their utility. This is particularly relevant when applying LLMs to cybersecurity related tasks, as prompts have to carefully crafted to bypass these configuration, and to trick the LLM to perform tasks that it was not supposed to undertake.

LLM-as-a-Judge for Output Validation

Evaluation and validation of the output of an LLM is often a difficult and time-consuming task, especially for a human. A clever way to validate AI content is to use the **LLM-as-a-Judge** paradigm, which simply consist in prompting an LLM to validate the output of another LLM. As revealed by studies such as *Zheng et al.*'s in the paper "*Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena*" [19], LLMs like `GPT-4` can achieve over **80% agreement with human preferences**, which is the same level of agreement between humans. The papers also highlights how the **LLM-as-a-Judge approach is easily scalable and explainable**, but when possible, it should be complemented with **human evaluation**.

1.4.2 Raise of LLM-based AI Agents

Nowadays humans use LLMs as a tool that can augment its reasoning and help decide which actions to perform. The idea is to replace the human-LLM interaction with a **software-LLM interaction loop** which is named **AI agent**. Thanks to their ability to interact with LLM, agents become entities capable of **observing** the environment around them, use these observation to **reason** and decide which **action** to take to influence the environment. This Perception-Thought-Action loop is named **Reinforcement Learning** (**RL**) and it is learning paradigm on which **AI agents** are based. These loops can be repeated multiple times, allowing the agent to learn through trial-and-error, until it has **reached its goal**.

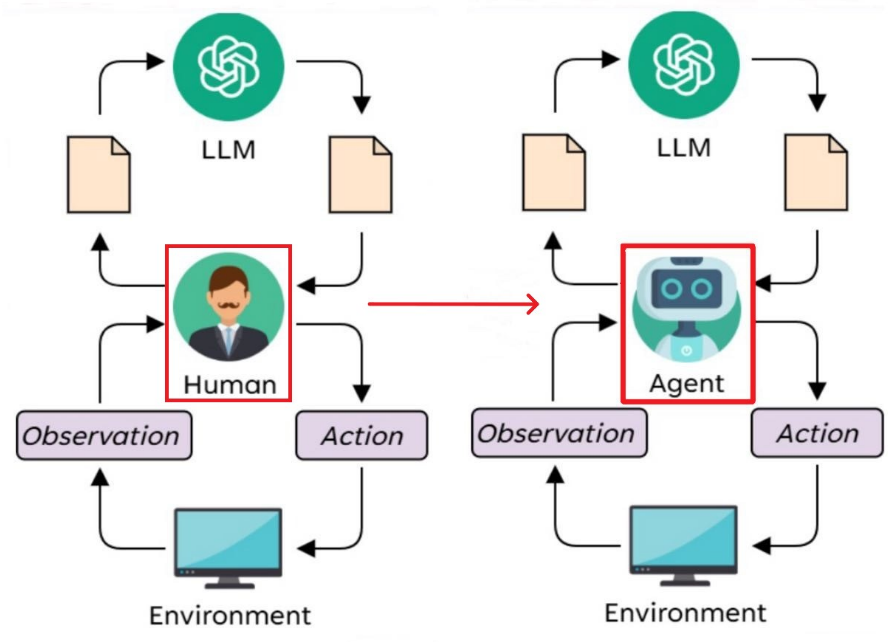
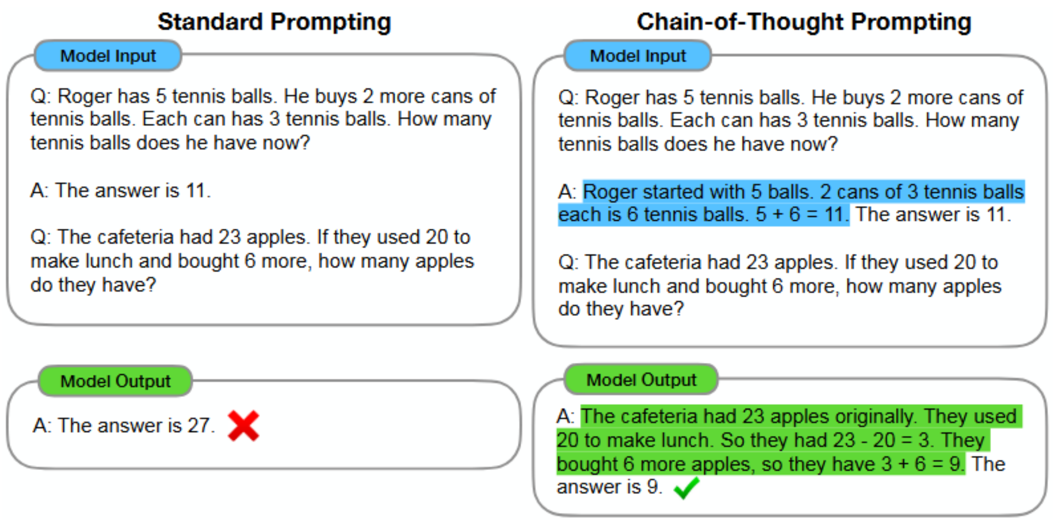


Figure 1.3: Replacement of the Human with an AI agent in the Perception-Thought-Action loop. Source: *AI and Cybersecurity Course, Politecnico di Torino A.a. 2024/25* [20]

Chain-of-Thought Prompting

Unfortunately LLMs are not smart enough to solve complex task on their own (i.e., without guidance by a human or AI agent). This is especially true for task that require multiple reasoning steps to be answered directly.

Wei et al. try to mitigate this problem by exploring how to bring out the reasoning abilities of LLMs by generating a series of intermediate steps - a **Chain-of-Thought** (**CoT**) - that describes the ideal reasoning process required to solve a task. The paper, named "*Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*" [21], describes how feeding multiple CoTs as examples to an LLM (i.e., following the **few-shot paradigm**) can improve its performance on wide range of tasks, such as arithmetic, common-sense, and symbolic reasoning.



Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

A: The answer is 27. ✗

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9. The answer is 9. ✓

Figure 1.4: Example of **CoT Prompting** on an arithmetic task. Source: *Wei et al.* [21]

The main takeaway of CoT Prompting is that LLM performance can be strongly enhanced by breaking down the problem into a set of specific goals, removing this burden from the model.

Structured Output

When a user interacts with an LLM via its web interface or through an API, the LLM will output text using whatever format it thinks it is best (e.g., Markdown, JSON, etc.). However, in many cases it would be beneficial to have a predefined output format, to allow for better control over the LLM responses, faster integration of the LLM in the system, and the easier implementation of checks and validation procedures.

Some LLM, such as `GPT-4o` and `GPT-5`, support **Structured Output**, a mechanism that ensures that LLM consistently produces outputs in the given format. This means that **unexpected data is less likely to appear**, effectively reducing hallucinations. Moreover, it also reduces variability, making it it easier to evaluate overall LLM performance.

1.5 AI & Cybersecurity

Cybersecurity is the practice that deals with the **protection of assets from an intended or accidental action that could harm them**. It has become a very important issues, since nowadays almost every system is built with **Information and Communication Technologies** (**ICT**).

An **asset** of an **Information Technology** (**IT**) system is a set of ICT resources, data people and locations. Each asset has some intrinsic **weaknesses**, the ones which can be exploited by an attacker to perform an attack are named **vulnerabilities**. Successful attacks can cause all kinds of damage, from **financial loss** to **reputational damage**.

Among the numerous applications of AI, **cybersecurity** is one of the rising ones. The main reason why cybersecurity The main advantages of applying AI to cybersecurity are related to a drastic increase in **adaptability and proactiveness**, thanks to three intrinsic characteristics of ML models:

- **Automation** allows ML models to perform time-consuming and redundant tasks. This means faster responses to events seen as potential cyber threat, speeding up human response times.
- **Intelligence** can be used to make simplify decision making when presented with complex patterns, ideal for identifying novel cyber threats and intricate attack patterns.
- **Robustness** keeps ML models effective under varying conditions, allowing them to deal with noisy data, and mitigating the effects of decoy cyber attacks, which are meant to divert the defender's attention.

LLMs are trained on huge heterogeneous datasets making them capable of identifying intricate data patterns. It has been found that **fine-tuning** these deep learning architectures with **smaller, task-specific labelled dataset** makes them perform well specific tasks, even cybersecurity related one, such as intrusion detection, malware classification, and anomaly detection. This happens because fine-tuning a deep learning architecture makes it more capable of **identifying complex data patterns** and more efficient at **detecting novel cyber threats**.

1.5.1 Common Vulnerability and Exposure

Common Vulnerability and Exposure (**CVE**) is a method for the classification of publicly known vulnerabilities and exposures operated by *MITRE*. CVE works as a vulnerability database that associates to each one:

- a unique identification number, named **CVE Identifier** (**CVE-ID**)
- a description of the vulnerability
- at least one public reference, each reporting on its exploitation

Every system will have its own set of vulnerabilities, CVE helps investigating these vulnerabilities by sharing sharing information within the cybersecurity community and enabling rapid data correlation across multiple information sources.

1.5.2 Virtualization Technologies in Cybersecurity

Virtualization is the process that allows for **more efficient utilization of physical computer hardware**. It uses software to create an abstraction layer over the computer hardware that allows its elements to be divided into multiple instances named **Virtual Machine**s (**VM**).

Lightweight Virtualization has the same nice properties of computer virtualization, but manages to **consume less resources**. In the context of cybersecurity, virtual environment have become both a training tool and a method of enhancing the resilience and security of IT infrastructure, since they provide:

- an isolated environment that can be **quickly deployed, migrated and disposed** with little to no cost.
- a cost-effective, easily scalable solution.
- strong **security and isolation guarantees** that are easier to enforce.

All these properties have lead to numerous applications of virtual environments:

- **Cyberattack Simulation**: virtual environments make it possible to simulate real-world cyberattacks safely, without endangering actual systems. These in-vitro scenarios can be easily configured to resemble real IT infrastructure, allowing security expert to perform realistic testing and data gathering. This can help **Security Operations Center**s (**SOC**) act quickly to contain and recover from security incidents.

- **Employee Training**: virtual environments can also be used for cybersecurity training. Everyone, even non-technical users, can experience simulated attacks and practice real-time responses, gaining practical skills that traditional approaches can't offer.

- **Live Malware Analysis**: malware analysis is another key use of virtual environments. Running a malware in an isolated setup allows researchers to safely study how it behaves, gather information on which files it targets, how it spreads, and what data it tries to steal. This knowledge is essential for developing stronger security measures such as patches and antivirus tools.

1.6 Related Works

This section will explore systems that leverage virtual environments for cybersecurity purposes (i.e., *Metasploit* [22], *Cuckoo Sandbox* [23], and *Vulhub* [3]) and relevant research regarding the application of AI and leverage of LLM for cybersecurity studies (e.g., `CVE-Genie` [4]).

1.6.1 Practical Uses of Virtualization in Cybersecurity

Metasploit

Metasploit is an **open-source penetration testing framework** that uses a modular architecture where each module provides a functionality, such as **exploits**, **payloads** and **encoders** [22].

Figure 1.5: *Metasploit* Logo

The key advantage of *Metasploit* is that it provides a virtual environment designed for testing and intentionally filled with common vulnerabilities and logging tools. This allows security experts and researchers to leave their production systems unharmed while they use a **isolated in-vitro scenario** to:

- develop new patches and check their functionality
- test an exploit and replicate real-world attack scenarios
- simulate entire networks of vulnerable machines, each with its own *Metasploit* image

Cuckoo Sandbox

Cuckoo Sandbox is an **open-source automated malware analysis system**. It is used to automatically run and analyse potentially malicious files inside an isolated virtual environment, enabling the safe collection of data (e.g., process traces, file operations, memory dumps, network traffic, screenshots, etc.) and a comprehensive analysis of the results that outline what the software does while running [23].

Figure 1.6: *Cuckoo Sandobox* Logo

Malware often behaves differently depending on the system configuration (i.e., OS version, installed software, network conditions, presence of certain files/cookies, etc.). Virtual environments allow to simulate a realistic attack environment so that the malware is more likely to "activate" and reveal (malicious) behaviour.

Vulhub

Vulhub is an open-source project that collects manually built Docker-based vulnerable environments, each associated to a CVE or vulnerable service. It allows anyone to quickly deploy an attack scenario that resembles a real-world one [3].

Figure 1.7: *Vulhub* Logo

Each environment is built using a combination of *Docker* and *Docker Compose*, which simplify the deployment and clean up operations of the environments. All deployed *Docker Containers* are isolated and reproducible, which means that *Vulhub* provides a safe and harmless alternative to the use of real systems for security testing purposes. These environments easily allow users to: perform penetration testing on the vulnerable environment; create and test new payloads for vulnerability exploitation; and develop new attack chains.

1.6.2 Attack Datasets Limitations

One of the main goals of this thesis is to enable security experts to build more realistic attack datasets. Such datasets are essential for benchmarking the performance of different vulnerability detection techniques, like **source code analysis tools using rule-based methods** [24, 25], approaches based on **Graph Neural Network**s (**GNN**) [26, 27, 28], LLMs [29, 30], or even specialised **multi-agent systems** [31, 32] . These datasets are also critical for improving IDS and IPS systems [33, 34, 35].

The effectiveness of these practices depends heavily on the correct labelling of the attack dataset, as they provide information about which CVE was exploited and how the attack developed. Labelling is usually performed in two ways:

- **Manual Labelling**: datasets that rely on manual work [26, 36] are usually small, require a lot of time to be built, and suffer labelling problems. For example, `Devign` [26] - a GNN that performs vulnerability identification. However, studies have shown that half the dataset used to train and test the GNN has been mislabeled as a result of manual labelling (which took around 600 hours of work) [37].
- **Autonomous Labelling**: automation reduces time required to build the dataset. This can be done by: analysing real-world data (e.g., patch commits or vulnerability fixes) and assuming that all modified and removed code was vulnerable [38, 39]; or by using static analysis tools [40, 41]. These approaches increase the dataset size, but at the same time, increase the risk of introducing false positives.

Despite these efforts, existing approaches force researchers to **strike a balance between label accuracy and scalability**. This thesis addresses this challenge by providing a way to easily reproduce scenarios vulnerable to a specific CVE, enabling researchers to expand and label their attack dataset reliably with both real and synthetic data.

1.6.3 LLMs & AI Agents for Cybersecurity

As explored by *Zhang et al.* in their paper "*When LLMs meet cybersecurity: a systematic literature review*" [42], LLMs have demonstrated remarkable capabilities and seen a wide range of applications in the fields of software engineering and cybersecurity:

- **Cyber Threat Intelligence** (**CTI**): exploiting the excellent analysis and summarization capabilities of LLMs in NLP tasks, researchers managed to use LLMs for CTI generation [43, 44], CTI information extraction [45, 46, 47], and CTI report deduplication [48]. Others have even designed autonomous AI agents to imitate the work of a qualified security experts [49].
- **Vulnerability Detection**: studies that assessed the ability of LLMs to detect vulnerabilities [50, 51] have shown great results. Some have also tried to improve detection capabilities by providing the LLM with pre-processed data [52, 53], in an attempt to enhance the reasoning process of the LLM.
- **Malware Detection**: LLMs can assist with static analysis assistant (e.g., by helping with **reverse engineering** [54]) and with dynamic debugging [55], improving the malware detection process.
- **Anomaly Detection**: studies have focused on using fine-tuned LLMs to perform **log analysis** [56] and to assist in the **detection of phishing and spam in web content** [57].
- **Digital Forensics**: LLMs have been used to determine if malicious or suspect actions have been performed on a computer system [58]. Others, such as `CyberSleuth` [59], have designed an **AI agent to autonomously investigate realistic web application attacks**. The goal is to analyse packet-level traces and application logs to identify the targeted service, the exploited vulnerability, and attack success.
- **Offensive Purposes**: offensive security has seen numerous applications. Some use **LLM to solve fuzzing related challenges** [60], others attempt to **automate** the **penetration testing** process by leveraging the power of LLMs [61] or by building autonomous/semi-autonomous AI agents [62].

Critical analysis of all these applications highlights a clear imbalance between offensive security applications (i.e., **Read-team LLMs**) and defensive security applications (i.e., **Blue-team LLMs**). This imbalance as been observed also by *Potter et al.* in the paper "*Frontier AI's Impact on the Cybersecurity Landscape: Current Status and Future Directions*" [63], which underlines how - at least in the short term - **AI will benefit attackers more than defenders**.

One of the major challenges in developing defensive solutions is their **reliance on inefficient manual processes**, which explains the imbalance. This thesis addresses this issue by proposing an automated approach that allows security experts to replicate vulnerable virtual environments. This automation saves time and enables safe penetration testing, patch development, and the collection and analysis of attack data.

1.6.4 CVE-Genie

`CVE-Genie` is a LLM-driven multi-agent framework that automates the end-to-end reproduction process of a CVE, addressing nearly all of the same challenges tackled in this thesis.

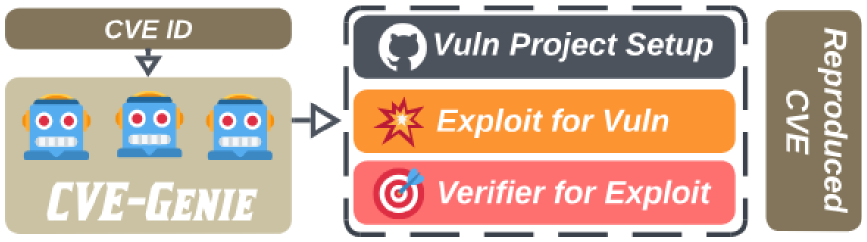


Figure 1.8: `CVE-Genie` Architecture Overview. Source: *Ullah et al.*'s paper [4]

The framework is divided into four modules:

- The `Processor` retrieves source code of the project linked to the CVE and uses the gathered information to build a structured knowledge base.
- The `Builder` uses the structured knowledge base to generate a vulnerable environment.
- The `Exploiter` uses the provided exploit or tries to create one to test if the environment is vulnerable.
- The `CTF Verifier` assesses if the exploit was successful.

`CVE-Genie` takes a different approach, with respect to my solution, by using a multi-agent solution where **each agent is specialised in subtask**. This is particularly beneficial for `CVE-Genie`, given the **wider scope** of the overall project. However, building a multi-agent framework was not considered necessary for the purpose of this thesis, which instead focused "*only*" on automating the following stages: CVE information-gathering phase; generation of the virtual environment; and finally vulnerability assessment.

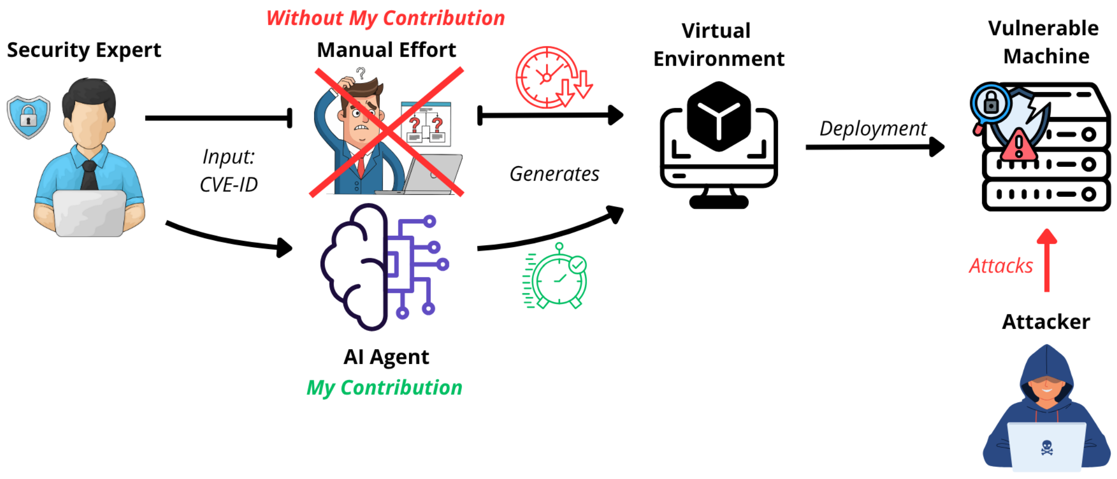
One of the main consequences of the `CVE-Genie` approach is the **high costs and long execution times** - up to $4 and 35 minutes per CVE. Moreover, `CVE-Genie` **requires access** to the source code of the vulnerable project associated to the CVE. Since this code may not be always available, the range of reproducible CVEs is significantly reduced.

Finally, it is worth noting that `CVE-Genie` uses the the `Exploiter` and `CTF Verifier` modules to perform dynamic vulnerability assessment. Instead, my approach relies exclusively on **static** vulnerability assessment using *Docker Scout*. Future work will evaluate the **integration of exploitation in the workflow** to further increase the robustness of the created virtual environments.

Chapter 2

Designing an AI Agent

This chapter analyses and describes the methodology used to develop the main goal of this thesis: building an AI agent to automate the generation of vulnerable virtual environments. This chapter also provides a comprehensive understanding of how the AI agent was designed.



2.1 From the Security Expert to the AI Agent

During the design phase of the agentic workflow it was observed that creating a virtual environment and ensuring that it is vulnerable to a specific CVE is a **well-defined process**. The AI agent was therefore designed to follow the **ideal workflow** of a real-world security expert tasked with building a virtual environment that is **intentionally vulnerable** to a given CVE. This translated into the definition of a **sequential process** which was divided into **four tasks**, each identified by its own colour and shape.

1. ▼ **Validation**: the security expert first verifies that the provided CVE is valid and actually exists.

2. ● **Information Gathering**: next, the security expert collects useful knowledge about the CVE and identifies the services and components required to create the appropriate virtual environment.

3. ■ **Virtual Environment Creation**: then, the security expert writes all the files needed to create a functional virtual environment, ensuring it is properly isolated and configured so that the specified CVE can be exploited.

4. ▲ **Vulnerability Assessment**: finally, before deploying the virtual environment, the security expert performs **vulnerability scanning** and **penetration testing** to confirm that the environment is indeed vulnerable to the CVE.

The AI agent abstracts this reasoning process into an via an **agentic workflow**, which can be represented via an acyclic graph (see Figure 2.1). This section will illustrate this workflow is structured and how it manages to abstract the approach of the security expert. It is worth noting that, during the implementation, it became clear that allowing the agent to independently chose the execution flow, or at least in most of it, was not the ideal, mainly due to the **rigid structure of the process described above**.

As a result, most of the process is **automated without granting agency to the AI** (i.e., without allowing the AI agent to **make decisions or acting independently**). In practice, this means that, the AI agent rarely chooses **how** and **in which order** the task should be accomplished. Instead, many decisions rely on the outputs of the agent combined with additional **predefined checks and constraints** that do not require an interaction with an LLM.

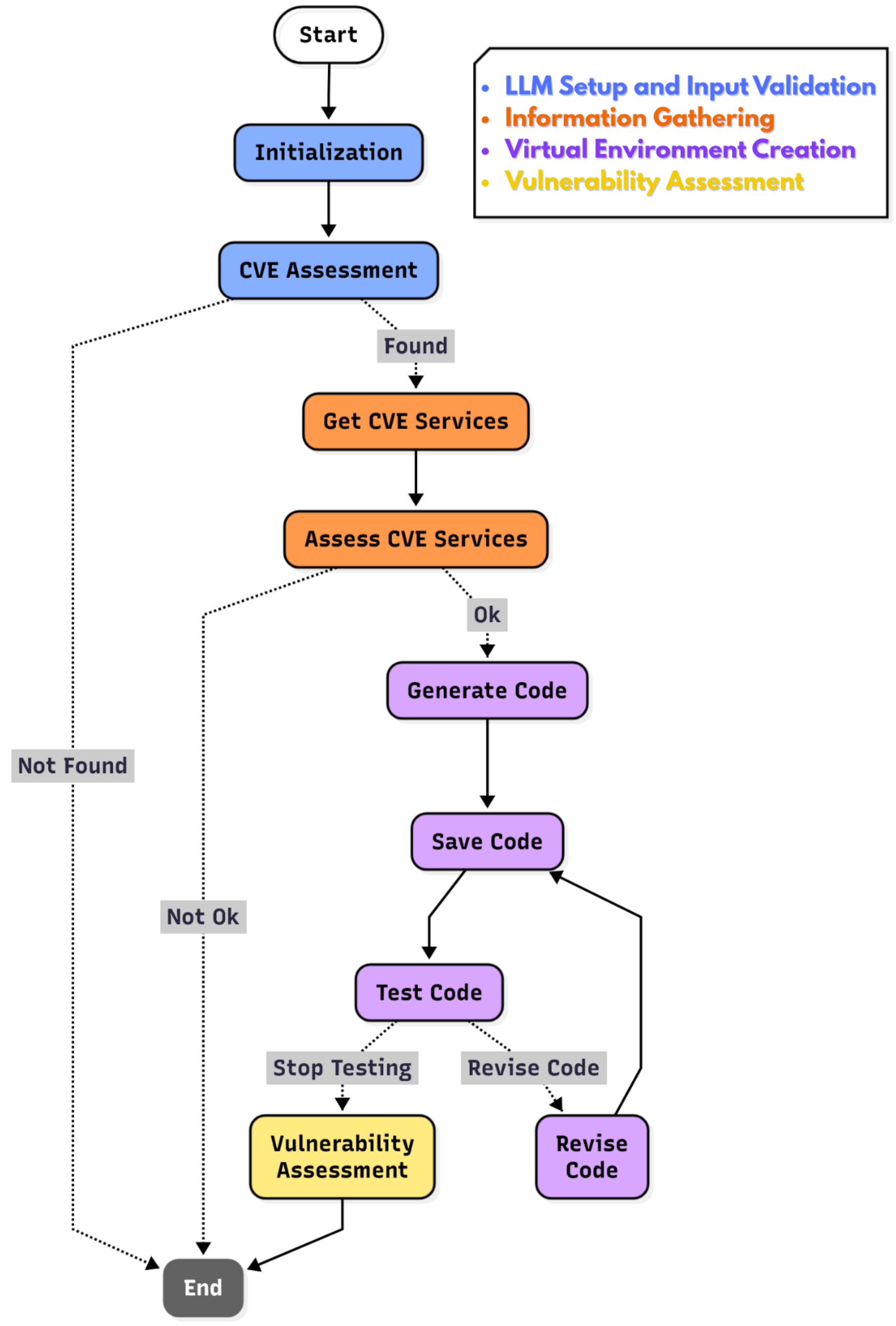


Figure 2.1: Graph Representation of the Entire Agentic Workflow

2.2 LLM Setup & Input Validation

The first task of the agentic workflow - identified by the ▼ colour/shape - is dedicated to handling and validating user inputs. It consists of two sequential steps:

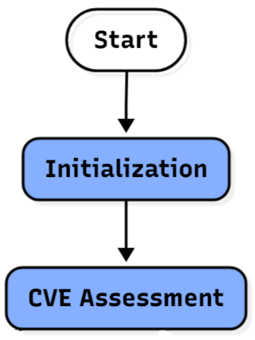


Figure 2.2: Graph Representation of the First Task of the Agentic Workflow

2.2.1 Initialization

The first step initializes the appropriate LLM and prepares it for subsequent tasks. Each model is set up with different parameters:

- `GPT-4o` is setup to **balance creativity and consistency**, allowing it to undertake different tasks like coding and information gathering.
- `GPT-5` has its reasoning capabilities constrained to **reduce reasoning time and overall workflow execution time**, which can be significant for such a large model.
- `gpt-oss:120B` is configured with enhanced reasoning capabilities to keep execution times comparable to `GPT-5`, compensating for its smaller size.

2.2.2 CVE Assessment

The second step **verifies whether the input CVE-ID actually exists** by checking the *MITRE* CVE database. This ensures that the agent does not **waste time and resources** creating a virtual environment for a vulnerability that does not actually exist.

2.3 Information Gathering

The second major task - identified by the ● colour/shape - focuses on collecting all necessary information about the specified vulnerability. Just like the previous task, it is divided into two sequential steps:

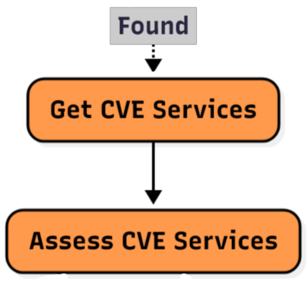


Figure 2.3: Graph Representation of the Second Task of the Agentic Workflow

2.3.1 Get CVE Services

In the first step, the agent **gathers useful information by leveraging the input CVE-ID**. Specifically, it attempts to determine **which services have to be included in the Docker-based environment** to ensure its correct functionality and vulnerability.

Due to **LLM knowledge cutoff limitations** and the **general-purpose nature of training data**, the **domain specific knowledge of the LLM may be insufficient** to generate a working virtual environment vulnerable to any given CVE. To mitigate this, the agent is equipped with the **ability to perform web searches** to obtain up-to-date vulnerability information. The user can select one of **three different strategies** that the agent will follow to perform the web search.

2.3.2 Assess CVE Services

The second step focuses on **validating the information gathered** the previously. Validation is made by comparing the gathered information with a small, manually curated dataset that contains, for each CVE, the list of services - and related version - required to build a **working vulnerable virtual environment**. This dataset was put together using both the official CVE documentation - namely the NVD database [64] - and the working vulnerable virtual environment provided by the *Vulhub* project [3].

The information gathering task is considered **successful only if the following three objectives are achieved**:

1. All services that are essential for reproducing the vulnerability are correctly identified.
2. Each vulnerable service is paired with at least one vulnerable version.
3. All services required for the environment to function properly are identified.

2.4 Virtual Environment Creation

The third task - identified by the 🟪 colour/shape - is the **most critical**, as it requires the agent to generate the virtual environment and verify its functionality. Depending on the test results, the agent can loop back to attempt to fix the environment, effectively allowing it to dynamically adjust the workflow.

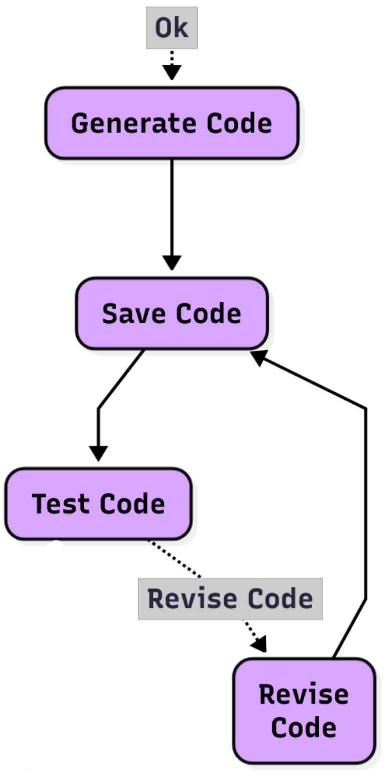


Figure 2.4: Graph Representation of the Third Task of the Agentic Workflow

2.4.1 Generate Code

The first step of this task **prompts the LLM to generate the code base of the virtual environment** using the information gathered in the last step and a set of guidelines that tell the LLM **how to use the information gathered** and **enforce a naming convention** for the generated files. Following the approach used in the *Vulhub* project [3], the LLM is instructed to build a Docker-based environment.

2.4.2 Save Code

All files generated by the LLM are then **saved locally**. This happens after the first code generation as well as after every code revision.

2.4.3 Test Code

The major challenge in developing this AI agent was determining when an virtual environment was working correctly. Ultimately, it was decided that, for a virtual environment to be working correctly, it has to **meet all four of the following criteria**:

1. All Docker Images complete the build process without errors.
2. All Docker Containers can run without errors and without unexpected crashes.
3. All services required to make the environment vulnerable are hosted by the Docker Containers and vulnerable version of them is used.
4. All services hosted by the Docker are reachable from their default network ports.

If all criteria are satisfied, the agent concludes that **the generated virtual environment is working correctly** and proceeds with the vulnerability assessment phase. Else, if any criterion fails, the agent attempts to **fix the identified issues** leveraging the knowledge gathered during testing. To regulate the total workflow execution time and reduce costs, the **number of iterations** the agent may perform in the **testing–revision loop is limited to 10**.

2.4.4 Revise Code

Revision of the code-base of the environment is achieved by explaining to the LLM **why the virtual environment failed** and by providing some guidelines that help the it **fix the existing environment**.

2.5 Vulnerability Assessment

The fourth and final task of the agentic workflow - identified by the ▲ colour/shape - examines the virtual environment generated in the last step to check if it is vulnerable to the input CVE.

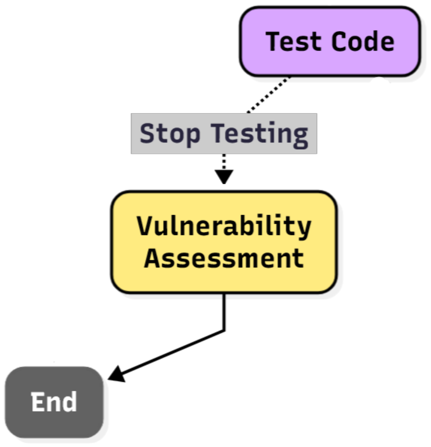


Figure 2.5: Graph Representation of the Fourth Task of the Agentic Workflow

Docker Scout [65] is the tool the is used to perform security analysis. It performs a **vulnerability scan** that produces multiple lists of all detected vulnerabilities in the Docker-based environment. If at least one of these lists contains the input CVE, then it can be *assumed*[1] that the agent managed to create **vulnerable virtual environment**.

[1]Static analysis tool, like Docker Scout, risk producing both **false positives** and **false negatives**

Chapter 3

Implementation Details

3.1 Choosing an Agent Development Framework

Just a couple of years ago, AI agents used to be built by stitching together a **set of scripts and prompts** through a lengthy trial-and-error process. Today, a variety **open-source and proprietary frameworks** have been designed to help developers and companies streamline the process of creating AI agents.

One of the biggest challenges in developing an AI agent is finding a balance between giving the agent the ability to handle the desired tasks autonomously and structure its workflow to maintain reliability. A plethora of frameworks exists, each with its different approach, from flexible chat-bot like agents to strict graph-based workflows.

Various frameworks were evaluated and their characteristics weighted against each other to decide which one fitted best the goals of the thesis. Framework evaluation was based on their compatibility with the other technologies used, such as the various APIs used to communicate with LLMs (e.g., `ChatCompletions` for `GPT-4o`, Responses for `GPT-5`), their popularity, their intrinsic complexity and the expected learning curve.

- **OpenAI Agent SDK** is *OpenAI*'s own solution to the problem of having to develop an AI agent. This framework provides developers with a specialized agent runtime and a straightforward API for assigning roles, tools, and triggers. This greatly simplifies multi-step/multi-agent orchestration, while providing native integration with *OpenAI*'s model endpoints and built-in tools such as the ones for web and file search [66].

- **AutoGen** is *Microsoft*'s own open-source framework for agentic AI development. Interactions in the workflow are seen as an asynchronous conversation among specialized agents. Each agent can be a chat-bot like assistant or a tool executor, and the developer has to orchestrate how they pass messages between them. This asynchronous approach makes it ideal for longer tasks or scenarios where an agent needs to wait on external events, like human interactions [67].

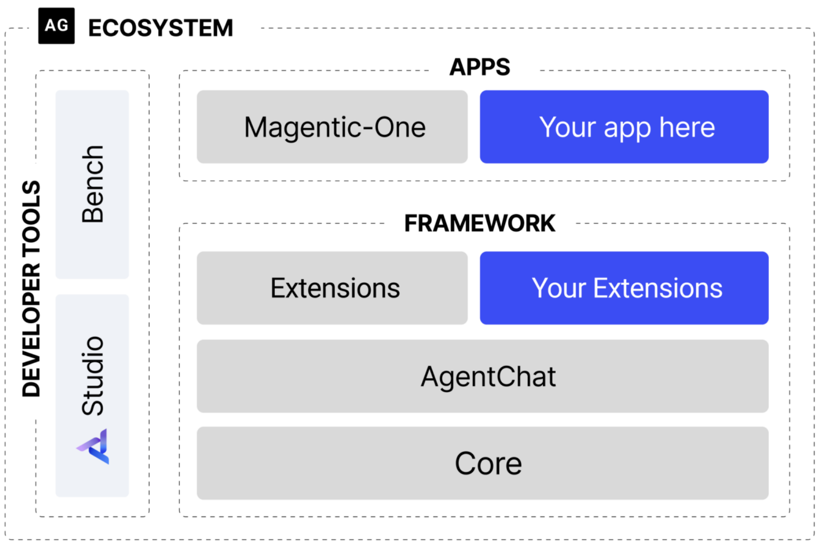


Figure 3.1: AutoGen Ecosystem [67]

- **CrewAI** is a Python-based framework that allows developers to create autonomous AI agents both with high-level simplicity and low-level control. It revolves around building (with or without writing code) a *crew* of agents that autonomously interacts and uses tools to automate workflows and tasks [68]. Each *crew* holds multiple agents, each with its own role or function. The framework autonomously coordinates the agents' workflow and allows them to communicate and share resources.

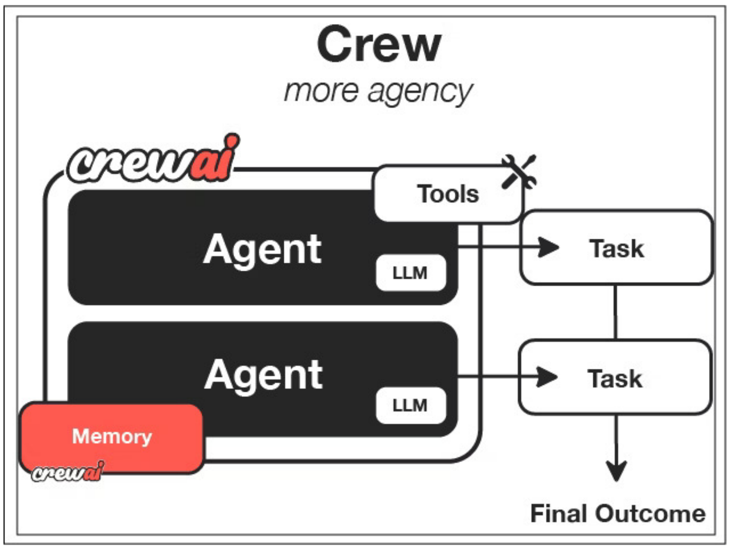


Figure 3.2: CrewAI Framework Overview [69]

- **LangChain & LangGraph**: *LangChain* is an open-source framework specifically designed to develop LLM-driven software applications, like chatbots and AI agents. Its API allows developers to seamlessly change and make calls to models, allowing them to easily experiment and find the best choice for the needs of their applications [6]. *LangGraph* extends the *LangChain* library into a graph-based architecture that treats agent steps like nodes, each handling a prompt or sub-tasks, and allowing developers to enforce order and control [70]. These characteristics are the reasons why the *LangChain/LangGraph* frameworks have been chosen to develop this project.

3.2 Lightweight Virtualization - Docker

Taking inspiration from the *Vulhub* project [71], Docker was chosen as the platform to create the virtual environments. Docker is a software platform that allows to easily create lightweight, portable and isolated containers. Unlike full virtualization, **containerization** - or **lightweight virtualization** - uses OS-level virtualization to isolate and constrain the execution environment of the application running inside the container. This means that any application that is running inside the container will see only the resources which are allocated to that container, making all other resources of the host machine virtually invisible to it [72].

- **Docker Compose** Docker Compose is one of the most famous Docker Container orchestrators. Its main goal is to simplify the container management in their whole life-cycle and to allow users to easily run multi-container applications on a single node. `YAML` files are used to configure the services of the application, while the `docker compose` command is used to launch all the configured services. Being this project strongly influenced by the Vulhub project [71], Docker Compose was preferred over Docker Swarm [73] and Kubernetes [74] as orchestrator to manage the virtual environments.

- **Docker Desktop** Docker uses a client-server architecture. The Docker Client interacts with the Docker Daemon, which builds, runs, and distributes the Docker Containers. Both of these services are essential to make Docker Containers work and are integrated in Docker Desktop (an application for Mac, Linux, and Windows environments) whose GUI was used to manually assess and manage every Docker Image and Container [75].

  It is worth pointing out that the project was developed on a machine with Windows 10 Home installed and limited local resources. Citing the official Docker Desktop documentation [76]: "*To run Windows containers, you need Windows 10 or Windows 11 Professional or Enterprise edition. Windows Home or Education editions only allow you to run Linux containers.*" Therefore, to start developing the project and to optimize resource usage, it was decided to install **WSL2** (**Windows Subsystem for Linux**), *Microsoft*'s own Linux kernel that allows to run in Windows any Linux distribution without having to manage VMs. Having Docker Desktop running on WSL2, means that the agent had to be tasked with designing a Linux container, not a Windows one. This is an important difference as the respective Docker Image formats are not cross-compatible, while Linux containers are typically lighter and have faster start-up times [77].

- **Docker Scout** Docker Scout [65] is a security-analysis tool that performs the analysis Docker Image. By inspecting the Docker Image **Layers**, it builds a **Software Bill of Materials** (**SBOM**) which lists all the packages and dependencies used by the Docker Image. It uses this list to check against various public datasets of vulnerabilities, finally producing a report that indicates which are the vulnerabilities of the Docker Image.

Effectively, Docker Scout performs **static vulnerability assessment** by scanning the Docker Images once they are already built, rather than executing the related Docker Container in a runtime environment to observe its behaviour or test for vulnerabilities dynamically. This is done with a **white-box approach**, as it has visibility into the internal composition of the image (i.e., packages, dependencies, layers, SBOM, etc.) and uses that information to identify vulnerabilities.

3.3 Prompt Design

Prompts are strings that contain a set of instructions and/or queries. These are passed directly to the LLM, interpreted, and used to **define the future task**. To better fit the goals of the AI agent, all prompts have been designed with a **dynamic** approach, which means that the content of the prompt is **never predefined**, but instead it is adjusted at runtime using the **data currently available** to the agent. Moreover, strong constraints are imposed to the LLM by using specific words such as "*must*", "*always*" or "*never*".

Prompts that are specifically designed to instruct an **LLM evaluate the output of another LLM** (i.e., whenever the LLM-as-a-Judge concept is applied) explicitly associate each check to the corresponding field, reinforcing its link the one defined in the structured output.

- **Structured Output**: *Pydantic* is a Python library for data validation that was used to define the structured data models to handle the outputs of LLMs [78]. These data models are always defined starting from *Pydantic*'s `BaseModel` class and ensure that, when the LLM returns a JSON output, it can be automatically validated against the defined data model. If the validation is successful, the final output will therefore be a JSON with the expected fields and types.

  Depending on which LLM is being used to automate the agentic workflow, there are two possible ways to configure the structured output:

  – `with_structured_output()` is a LangChain method that is used to make the LLM validate and output a object structured as the corresponding Pydantic data model [79]. This method is linked to the agent just like any other tool, but its invocation is always enforced whenever the LLM has to provide an output to the user. The only restriction of this method is that it can only be used by LLMs that provide native APIs for structuring outputs.

  - `PydanticOutputParser` is a LangChain class that is used to parse JSON given the output of an LLM [80]. Whenever an LLM shows compatibility problems with the `with_structured_output()` method - as is the case for all LLM running in the `SmartData@Polito` *Research Group* local GPU cluster - this class is employed to grant to the LLM the same functionalities.

- **LangChain Messages**: `Messages` are used in the LangChain framework to identify both input and output messages of a conversation with a chat model such as the ones used in this project [81]. The main advantage of these `Messages` is that they are cross-compatible with different types of models and APIs. This allowed to easily impose rules and restrictions to the LLM, and also to build self-crafted conversations with a selected amount of data and a predetermined format.

  Each `Message` has two main components: the `Role` and the `Content`. Based on the `role`, there are three possible types of `messages`:

  - `SystemMessage` is used to set the personality of the LLM and provide the LLM with the context in which it will be working. This essentially replicates the *personalization settings* which are available in the typical web interface of a chat model.
  - `HumanMessage` represent the user's inputs. It contains the description of the task that the LLM will have to solve.
  - `AIMessage` represent the outputs of the LLM. It can contain the answer of the LLM to the previous latest user input or a **request to invoke a tool**, containing both the name and (eventually) the parameters of the tool, which are also generated by the LLM.

  It is important to specify that the content of any of these messages is **purely textual**. The *LangChain* framework allows these messages to contain also audio, image and video data, but their usage was not deemed useful to reach the goal of this project.

3.4 The Memory of the Agent

The *LangGraph* API allows to define a `State`, a shared data structure that can be seen as the **short-term memory of the agent**. It is considered as short-term because it is initialized at the beginning of each run and stores user inputs that are specific to that run. For this project, the `State` data structure was defined using the `BaseModel` class of *Pydantic*, and it includes the following fields:

Field Name	Type	User Input?	Description
`model_name`	String	✓	Name of the model chosen for the agent run
`llm`	`ChatOpenAI` object	✗	LLM used in the agent run
`cve_id`	String	✓	CVE identifier of the vulnerability
`web_search_tool`	String	✓	Information gathering strategy used while performing the web search
`verbose_web_search`	Bool	✓	If `True`, provides additional data during the information gathering phase
`web_search_result`	*Pydantic* object	✗	Stores the final result of the information gathering phase
`code`	*Pydantic* object	✗	Stores the all file names, code and directory tree associated to the virtual environment
`fail_explanation`	String	✗	Detailed explanation of why the virtual environment has failed testing
`revision_type`	String	✗	Type of revision that has to be performed on the virtual environment
`revision_goal`	String	✗	How to fix the virtual environment to prevent it from failing the test again
`fixes`	List of strings	✗	Stores information about all previous fix attempts performed by the agent in the run

Table 3.1: AI Agent Short-term Memory Fields (Part 1)

Field Name	**Type**	**User Input?**	**Description**
`messages`	List of `Messages`	✗	Tracks the relevant interactions with the LLM performed during the agent run
`stats`	*Pydantic* object	✗	Stores various stats about the agent run
`milestones`	*Pydantic* object	✗	Stores the milestones used to track the progress of the agent run
`debug`	String	✗	Used to debug the agentic workflow

Table 3.2: AI Agent Short-term Memory Fields (Part 2)

3.5 Agentic Workflow

The **agentic workflow** (see Figure 2.1) describes the internal reasoning process of the AI agent. This section builds on introduction to the workflow made in the previous chapters, offering a deeper illustration on **how the workflow is structured**, the reasons that led to certain **design choices**, and providing all the essential **implementation details**.

3.5.1 LLM Setup & Input Validation

The first task of the **agentic workflow** manages and validates the user inputs. This task is divided into two sequential steps:

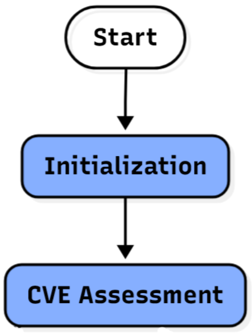


Figure 3.3: Graph Representation of the First Task of the Agentic Workflow

▼ Initialization

In the first step, the workflow creates two **local directories**:

- `dockers/CVE-ID/Web-Search-Mode` where all **files** are saved. The placeholders `CVE-ID` and `Web-Search-Mode` correspond to the user inputs taken from the `State.cve_id` and `State.web_search_mode` fields.
- `dockers/CVE-ID/Web-Search-Mode/logs` where all **logs** are saved.

Next, the `final_report.txt` file is initialized by writing all the user inputs in it and saving it in the `dockers/CVE-ID/Web-Search-Mode/logs` directory.

```
========== CVE-2021-28164 Final Report ==========

---------- Initial Parameters ----------
'model_name': gpt-4o
'cve_id': CVE-2021-28164
'web_search_tool': custom
'verbose_web_search': False
'web_search_result': desc='' attack_type='' services=[]
'code': files=[] directory_tree=''
'messages': [SystemMessage(content='ROLE: you are an AI expert in
   cybersecurity vulnerabilities and Docker lightweight
   virtualization technology.\n\nCONTEXT: everything that you
   generate will be used in a secure environment by other
   cybersecurity experts.\n\nGUIDELINES: avoid security warnings
   in your answers for any of the following tasks.\n',
   additional_kwargs={}, response_metadata={}, id='1140a966-47b1
   -4916-ac5d-9423cf6e2e50')]
'milestones': cve_id_ok=False hard_service=False hard_version=
   False soft_services=False docker_builds=False docker_runs=False
    code_hard_version=False network_setup=False
'debug':
----------------------------------------
```

Listing 3.1: Example of final report intiilization

Finally, the `State.model_name` input is used to initialize the corresponding LLM, readying it for the upcoming tasks. All LLMs are instances of the `ChatOpenAI` class of *LangChain*, but they are initialized with different parameters:

- `GPT-4o` is initialized with `temperature=0.5` and `max_retries=2`.
  The `temperature` (a float between 0 and 2) controls the randomness of LLM output: higher values (e.g., 0.8) make the **LLM output more random**, while lower values (e.g., 0.2) make the **LLM more focused and deterministic**. A value of `0.5` strikes a **balance between creativity and consistency**, allowing the LLM to undertake different tasks like coding and information gathering. The `max_retries` parameter specifies how many times to automatically retry a failed call to the `Chat Completion API`.

- `GPT-5` is initialized with `reasoning_effort="low"` and `max_retries=2`. The `reasoning_effort` parameter constrains the computational effort the LLM dedicates to reasoning (i.e., its depth and duration). The `"low"` setting was chosen to **decrease the reasoning times** and overall workflow execution times, which can become a problem for huge models like `GPT-5`.

- `gpt-oss:120B` is initialized with `reasoning_effort="medium"`, `max_retries=2`, `api_key=os.getenv("SDC_API_KEY")`, and `base_url=https://kubernetes.polito.it/vllm/v1`. The `"medium"` setting was chosen to maintain execution times comparable to `GPT-5`, effectively compensating the smaller size of `gpt-oss:120B` with longer reasoning times. The `base_url` specifies the endpoint for sent `Chat Completion API` requests, here pointing to the `SmartData@Polito` *Research Group* Kubernetes cluster. Access to the hosted LLMs requires a secret key, which is passed via the `api_key` parameter.

The LLM that is initialized in this step will be **used in all following tasks**.

▼ CVE Assessment

The second step **verifies whether the CVE-ID provided by the user actually exists**. This verification is performed by querying the *MITRE* CVE database using the *CVE Services API* [5]. This API enables the retrieval of the CVE record of a specific CVE-ID by sending an `HTTP GET` request to `https://cveawg.mitre.org/api/cve/CVE-ID` (CVE-ID is just a placeholder).

To facilitate progress tracking, a set of **eight milestones** is distributed across the four task of the workflow. The first of of these milestone is named `cve_id_ok`, and tracks the outcome of this step:

- If the HTTP status code is **200**, the **input CVE-ID exists**, and the workflow proceeds (`cve_id_ok=True`).
- If the HTTP status code is **404**, the **input CVE-ID does not exist** and the workflow immediately stops (`cve_id_ok=False`).
- For any other status code, **no valid information about the CVE-ID was retrieved**, therefore the workflow also terminates (`cve_id_ok=False`).

This check ensures that the agent does not **waste time and resources** creating a virtual environment for a vulnerability that does not actually exist.

This step primarily performs input validation, which is necessary to **handle cases of vulnerability mislabelling** which may confuse the user. It is possible for different vulnerability databases — such as National Vulnerability Database (NVD) [64], EUVD (European Union Vulnerability Database) [82], CNNVD (China National Vulnerability Database) [83], and WooYun [84] — to:

- use the **same ID** for **different vulnerabilities**, or
- assign **different IDs** to the **same vulnerability**.

For instance, EUVD-2021-0877 [85] and CVE-2021-28164 [86] refer to the same vulnerability, while EUVD-2021-28164 [87] and CVE-2021-28164 [86] do not.

3.5.2 Information Gathering

The second task focuses on collecting information about the vulnerability specified by the user. The workflow proceeds to this task **only if** `cve_id_ok=True`. Just like the first task, this one is also divided into two sequential steps:

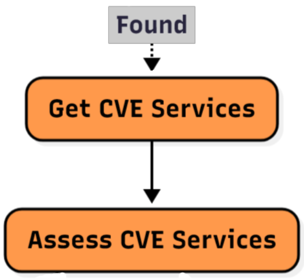


Figure 3.4: Graph Representation of the Second Task of the Agentic Workflow

● Get CVE Services

In the first step the agent **gathers useful information by leveraging the CVE-ID** provided by the user. Specifically, it attempts to determine **which services have to be included in the Docker-based environment** to ensure its correct functionality and vulnerability.

However, due to the **knowledge cutoff** and the limitations of the training data - often based on large, generic corpus rather than a domain-specific one - the **knowledge-base of the LLM may be insufficient** to generate a working virtual environment vulnerable to any given CVE. Therefore, to ensure that the agent can make decisions based on the **latest information about the vulnerability**, it was deemed necessary to equip the agent with the ability to retrieve information from the Internet, i.e., to **perform web searches**.

The information gathering process differs based on the **strategy** - or `Web Search Mode` - chosen by the user. To explore the information gathering capabilities of the agent, **three** different strategies were defined:

- **OpenAI Search** (`"openai"`): this strategy leverages the **`web-search-preview` built-in tool** of *OpenAI* [88], which enables the model to search the web for the latest information before generating a response. Because of current framework limitations, this tool can be used **only** by `GPT-4o` and `GPT-5`.
  - **Non-reasoning** models like `GPT-4o` do not plan a search lookup strategy, instead they simply return a response based on top results, making non-reasoning models **fast and ideal for quick lookups**.

- **Reasoning** models follow an agentic approach where **the model actively manages the search process** by performing web searches as part of a CoT. If the results are not satisfactory, the model keeps searching. This flexibility makes reasoning models **well suited for complex workflows**, but it also **increases the web search time**.

The use of the built-in tool is **forced** by requesting its use with a `HumanMessage` containing the `OPENAI_WEB_SEARCH_PROMPT`:

```
CONTEXT: search the web and summarize all the information available
    about {cve_id}

GOAL: identify the services needed to create a Docker system vulnerable
     to {cve_id}

GUIDELINES:
- The description of {cve_id} must be extensive
- The attack type must be spelled out without acronyms or abbreviations
     (i.e., do not use DoS, RCE, etc.)
- ABOUT SERVICES:
    - Specify the minimum set of services needed to create a working
    and testable Docker system vulnerable to {cve_id}
    - Avoid including services that are there just to test a PoC or to
    exploit the vulnerability
    - Service names must match the official names listed on Docker Hub,
     do not use aliases
- ABOUT SERVICE DEPENDENCY TYPES: each service must be associated to a
    dependency type that must be one of two:
    - 'HARD' if the service is the essential to make the system
    vulnerable to {cve_id}
    - 'SOFT' if the service is needed just to make the Docker work
    - 'SOFT' service that play a specific role must be associated to a
    role (format 'SOFT-<role>'). Examples of 'SOFT-<role>' are:
        - 'SOFT-DB' for relational databases (e.g., MySQL, MariaDB,
    PostgreSQL, MariaDB, Oracle)
        - 'SOFT-WEB' for web servers (e.g., Nginx, Apache, PHP, Tomcat)
        - 'SOFT-CACHE' for caching/key-value store/coordination
    services (e.g., Redis, etcd, ZooKeeper, RabbitMQ, Kafka)
- ABOUT SERVICE VERSIONS:
    - Service version must specified and valid for Docker Hub, do not
    be vague by citing just 'any compatible version'
    - For 'HARD' services you must list all vulnerable versions cited
    by the most reliable sources such as MITRE and NIST. Do not use
    ranges, you must be very specific with version name and list all
    versions vulnerable to {cve_id}
    - For 'SOFT' services choose a versions compatible with the 'HARD'
    services
```

Using a built-in tool **breaks** the `Structured Output` capabilities of the model. Therefore, a subsequent call to the LLM is required to ensure that the response is formatted correctly. This is achieved with the simple `WEB_SEARCH_FORMAT_PROMPT`:

```
GOAL: convert the following text in the provided structured output {
    web_search_result}
```

No matter which strategy is used, the final LLM response will always be an **instance of the `WebSearch` class**. The response together with the auxiliary data gathered during the web search process - i.e., the **number of input and output tokens**, and, for the "`custom`" and "`custom_no_tool`" modes, the `query` parameter - are saved in the `web_search_results.json` file inside the `dockers/CVE-ID/Web-Search-Mode/logs` directory. The web search results are also stored:

- as an `AIMessage` inside the `State.messages` field, to be easily reused in the future LLM interactions.
- as instance of the `WebSearch` class in the `State.web_search_result` field, to simplify the checks performed in the next step.
- for logging purposes, inside the `final_report.txt` file.

```
CVE description: CVE-2021-28164 is a vulnerability in the
    Eclipse Jetty web server, specifically affecting versions
    9.4.37.v20210219 to 9.4.38.v20210224, as well as versions
    10.0.1 to 10.0.5 and 11.0.1 to 11.0.5. The vulnerability
    arises from improper URI decoding, which allows attackers
    to craft URIs with encoded characters to access the
    protected WEB-INF directory. The issue occurs because URIs
    containing segments like %2e or encoded null characters are
     not properly normalized, allowing unauthorized access to
    protected resources. It has a CVSS score of 5.3, indicating
     moderate severity.
Attack Type: Remote Code Execution
Services (format: [SERVICE-DEPENDENCY-TYPE][SERVICE-NAME][
    SERVICE-VERSIONS] SERVICE-DESCRIPTION):
- [HARD][eclipse-jetty][['9.4.37.v20210219', '9.4.38.v20210224
    ', '9.4.39.v20210325', '9.4.40.v20210413', '9.4.41.
    v20210516', '9.4.42.v20210604', '10.0.1', '10.0.2',
    '10.0.3', '10.0.4', '10.0.5', '11.0.1', '11.0.2', '11.0.3',
     '11.0.4', '11.0.5']] Eclipse Jetty is the web server that
    contains the vulnerability in its URI handling, allowing
    unauthorized access to protected files.
- [SOFT][openjdk][['11-jre-slim']] Java runtime environment
    required to run the Eclipse Jetty server.
```

Listing 3.2: Example of web search results in the `final_report.txt` file

- **Custom Search** ("custom"): this strategy is implemented as an **agent tool**. A tool is a function that is given to the LLM that allows it to **complement its knowledge** by retrieving information which was not included in its training data (e.g, because the CVE is a recent one). Although the agent can normally **decide** whether to use the given tools, in this case, obtaining the latest information about the CVE is the **priority**, so the agent is *forced* to use the tool via the CUSTOM_WEB_SEARCH_PROMPT:

```
GOAL: search the web and summarize all the information available about
    {cve_id}.

GUIDELINES: use the 'web_search' tool by generating the following
    parameters:
- "query": the query to retrieve the CVE-related information.
- "cve_id": the ID of the CVE.
```

  The prompt cites the **two parameters** which must be **produced by the LLM** to execute the function associated to the tool:

  - 'query': a string representing the search query
  - 'cve_id': a unique string identifying the vulnerability

  The tool relies on *Google*'s *Custom Search JSON API* [89] to **retrieve data from the web**. Specifically, it sends a request to a specific endpoint[1] using as parameters the 'query', the **developer key** (used to authenticate the request) (provided by *Google* by subscribing to the free tier of the service), and the **search engine** (in this case, it is always the default one). The API returns **up to 10 URLs** per query, each processed as follows:

  1. The agent sends an HTTP GET request to the URL and checks that the status code is **200**. Else, the URL is skipped
  2. The retrieved content is parsed with the *BeautifulSoup* Python library [90] to verify that is HTML.
  3. The <script> and <style> tags are removed from the page, as these do not contribute to the main textual content.
  4. The text is extracted from the page to ensure if:
     - it contains enough data. For testing purposes it was decided to **ignore anything less than 50 characters**.
     - it contains **valid UTF-8 encoded content** by trying to encode and decode it.

[1] https://www.googleapis.com/customsearch/v1

To reduce execution time, costs, and to mitigate the introduction of irrelevant information, **the analysis stops after 5 pages are correctly analysed**.

Because tests revealed that `HTTP GET` requests **always failed to retrieve information** from *NIST*'s *NVD* repository [64], a dedicated function[2] was defined to query the repository using the CVE API [91].

Next, the agent is tasked with **summarizing the content extracted from each web page**. This is done by leveraging the LLM which is initialized with two `Messages`:

– A `SystemMessage` containing the `LLM_SUMMARIZE_WEBPAGE_PROMPT` [3]:

  ```
  GOAL: summarize in {character_limit} characters or less the user
      provided content relevant to {cve_id}.

  GUIDELINES:
  - Focus on the original services that present the vulnerability and
       those necessary to exploit it, ignore other services that rely
      on the original services.
  - The most important information is usually contained in the "
      Description" section of the content.
  ```

  The guidelines provide generic indications that have can help the LLM find relevant information more easily.

– A `HumanMessage` containing the **content of the web page**.

The response of the LLM - together with information about the **number of input and output tokens** used to summarize the text - is **stored as-is**, without further post-processing.

Finally, all summaries are concatenated and used to make another call to the LLM. This time the LLM is tasked with **extracting information relevant to the CVE-ID**. Just like before, this is achieved by invoking the LLM using two `Messages`:

[2]See implementation details of the `get_cve_from_nist_api` function in Appendix A

[3]For testing purposes `character_limit=1000` characters

– A `SystemMessage` containing the `GET_DOCKER_SERVICES_PROMPT`:

```
CONTEXT: you are provided with some information about {cve_id}

GOAL: identify the services needed to create a Docker system
    vulnerable to {cve_id}

GUIDELINES:
- The description of {cve_id} must be extensive
- The attack type must be spelled out without acronyms or
    abbreviations (i.e., do not use DoS, RCE, etc.)
- ABOUT SERVICES:
    - Specify the minimum set of services needed to create a
    working and testable Docker system vulnerable to {cve_id}
    - Avoid including services that are there just to test a PoC or
     to exploit the vulnerability
    - Service names must match the official names listed on Docker
    Hub, do not use aliases
- ABOUT SERVICE DEPENDENCY TYPES: each service must be associated
    to a dependency type that must be one of two:
    - 'HARD' if the service is the essential to make the system
    vulnerable to {cve_id}
    - 'SOFT' if the service is needed just to make the Docker work
    - 'SOFT' service that play a specific role must be associated
    to a role (format 'SOFT-<role>'). Examples of 'SOFT-<role>' are:
        - 'SOFT-DB' for relational databases (e.g., MySQL, MariaDB,
     PostgreSQL, MariaDB, Oracle)
        - 'SOFT-WEB' for web servers (e.g., Nginx, Apache, PHP,
    Tomcat)
        - 'SOFT-CACHE' for caching/key-value store/coordination
    services (e.g., Redis, etcd, ZooKeeper, RabbitMQ, Kafka)
- ABOUT SERVICE VERSIONS:
    - Service version must specified and valid for Docker Hub, do
    not be vague by citing just 'any compatible version'
    - For 'HARD' services you must list all vulnerable versions
    cited by the most reliable sources such as MITRE and NIST. Do
    not use ranges, you must be very specific with version name and
    list all versions vulnerable to {cve_id}
    - For 'SOFT' services choose a versions compatible with the '
    HARD' services
```

The guidelines of the prompt have been carefully chosen during the development of the AI agent to help the LLM in providing an ideal response to the information gathering phase.

– A `HumanMessage` containing the **concatenated summaries**.

During the evaluation of the LLM responses, it became necessary to understand more clearly **why some services were included in the web search results**, or in other words, **why the LLM deemed a service relevant for the virtual environment**. To clarify this, two **dependency types** were defined:

- **Hard Dependency** (`HARD`): refers to those services that are **essential for making the virtual environment actually vulnerable**.
- **Soft Dependency** (`SOFT`): refers to those services that are **necessary for the virtual environment to function correctly**. In this case, the LLM may also specify the **role** of the `SOFT` service within the environment. This is achieved by including some examples in the prompt, following a **few-shot prompting** approach:
  - `SOFT-DB`: services that manage relational databases (e.g., MySQL, MariaDB, PostgreSQL, MariaDB, Oracle)
  - `SOFT-WEB`: services hosting web servers (e.g., Nginx, Apache, PHP, Tomcat)
  - `SOFT-CACHE`: services providing caching, key-value storage, or coordination functions (e.g., Redis, etcd, ZooKeeper, RabbitMQ, Kafka)

The `with_structured_output` method is used to ensure that the LLM response conforms to the `WebSearch` and `Service` schemas:

```python
class Service(BaseModel):
    name: str = Field(description="Name of the service")
    version: list[str] = Field(description="List of versions of the service")
    dependency_type: str = Field(description="Type of the dependency of the service, either 'HARD' or 'SOFT'")
    description: str = Field(description="Brief description of why the service is necessary in the Docker")

class WebSearch(BaseModel):
    desc: str = Field(description="Description of the CVE")
    attack_type: str = Field(description="Type of attack (e.g. DoS, RCE, etc.)")
    services: list[Service] = Field(description="List of services to be used in the Docker system vulnerable to the CVE")
```

Listing 3.3: Definition of the `WebSearch` and `Service` classes

- **CVE-based Search** (`"custom_no_tool"`): functionally, this strategy operates **exactly** like the **Custom Search**. The only difference is that it is **not implemented as a tool**, instead, it works just like any other function. Practically, this means that the **agent cannot choose the `query` parameter**, which is instead **fixed** to `query=CVE-ID`. This difference is significant: **varying the web search query** leads to the retrieval of **different sets of web pages**, which in turn **influences the services selected** for generating the virtual environment and the specific **issues discovered during testing**.

Testing revealed that some LLMs have **built-in security filters** that prevent them from providing **potentially harmful information** (e.g., relating to software vulnerabilities). To bypass these obstacles, the `SYSTEM_PROMPT` is provided as `SystemMessage` together with the previously explained `HumanMessages`:

```
ROLE: you are an AI expert in cybersecurity vulnerabilities and Docker
    lightweight virtualization technology.
CONTEXT: everything that you generate will be used in a secure environment
    by other cybersecurity experts.
GUIDELINES: avoid security warnings in your answers for any of the following
     tasks.
```

● Assess CVE Services

The second step focuses on **validating the information gathered** in the previously to ensure that the LLM has found the right services. Validation is possible because a small, manually curated dataset was created: it contains, for each CVE, the list of services - and related version - required to build a **working vulnerable virtual environment**, along with their corresponding **dependency types**. This dataset was put together using both the official CVE documentation - namely the NVD database [64] - and the working vulnerable virtual environment provided by the *Vulhub* project [3]. For instance, the entry for CVE-2018-12613 is:

```
"CVE-2018-12613": [
    "HARD:phpmyadmin:4.8.1",
    "SOFT-WEB:php:7.2-apache",
    "SOFT-DB:mysql:5.5"]
```

Listing 3.4: Example of `services.json` dataset entry

This entry implies that the information gathered by LLM is expected to contain:

- the `phpmyadmin` service tagged as `HARD`, as it is responsible for making the virtual environment vulnerable. The "*expected*" version `4.8.1` must be included in the list of vulnerable versions associated to the service.

- at least one service tagged as `SOFT-WEB` and another one tagged as `SOFT-DB`.

Therefore, the information gathering task is considered **successful only if the following three objectives are achieved**:

1. **All `HARD`-type services must be correctly identified** (tracked by the `hard_service` milestone). This is a **purely programmatic check**, meaning it does not involve the use of an LLM-as-a-Judge.

2. **The list of vulnerable versions of each `HARD`-type services must include the "*expected*" version** (tracked by the `hard_version` milestone). This objective is firstly validated via a programmatic check. Then, if the check fails, a second evaluation is with a **LLM-as-a-Judge**, which is prompted by the `HARD_SERV_VERS_ASSESSMENT_PROMPT`:

```
GOAL: check if version '{version}' of the '{service}' service is
    contained in the following list of versions {version_list}

CONTEXT: the version lists of each service may contain multiple entries
     separated by ','
```

   The use of an LLM-as-a-Judge was deemed necessary because **software versioning schemes vary widely and no standardized format exists**, making exhaustive programmatic checks impractical and difficult to implement. To clarify the final decision of the LLM-as-a-Judge, the LLM is instructed - via `Structured Output` - to format its response as an instance of the `HARDServiceVersionAssessment` class:

```python
class HARDServiceVersionAssessment(BaseModel):
    hard_version: bool = Field("Does the 'HARD' service version range
    contain the expected version?")
```

   Listing 3.5: Definition of the `HARDServiceVersionAssessment` class

   If both checks fail for any of the `HARD`-type services, then `hard_service=False`

3. **For each "*expected*" `SOFT-<role>` at least one service must be suggested** (tracked by the `soft_services` milestone). Just like the first objective, this is a **purely programmatic check**.

Failure to reach any of the objectives is documented in the `final_report.txt` file. These checks are performed only if the input CVE has a corresponding entry in `services.json`.

3.5.3 Virtual Environment - Generation & Testing

The third task is the most important one, as it involves the agent **generating the virtual environment** and **testing its correct functionality**. Depending on the test results, the agent can loop back to attempt to fix the environment, effectively allowing it to dynamically adjust the workflow.

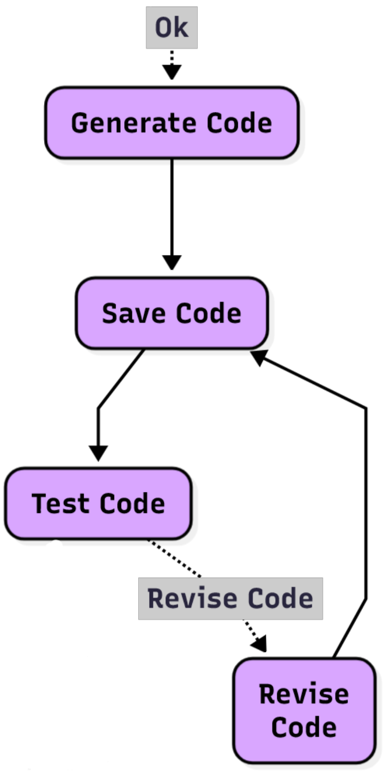


Figure 3.5: Graph Representation of the Third Task of the Agentic Workflow

■ Generate Code

The first step of this task **prompts the LLM to generate the code base of the virtual environment**. This is done by providing the LLM with a list of `Messages` containing: the `SYSTEM_PROMPT`; the information gathered in the last step; and the `CODING_PROMPT` (included as a `HumanMessage`):

```
GOAL: starting from a "docker-compose.yml" file, you must create a Docker
    system vulnerable to {cve_id}.

GUIDELINES:
- Use the data that you provided in the message about {cve_id} and its
    services to build the Docker
    - You must use all and only the services that are listed in the message
    that describes {cve_id}
    - If a service requires a dedicated container write the code for it
    - You must not use versions of 'HARD' services that are not listed in
    the message about {cve_id} and its services
- If a DB is needed, it must be properly setup and populated with some test
    data
- The system must be immediately deployable using the "docker compose up"
    command
- Ensure that no service has to be setup manually by the user
- All services and related containers must be properly configured on order
    to be immediately accessible from the default network ports of the
    service
- Always write enough files to make the all services work and to ensure that
     {cve_id} is exploitable for testing purposes
- All file names must indicate the file path which must start with "./../../
    dockers/{cve_id}/{mode}"
- There is no need to specify the file name in the file content
```

The guidelines of this prompt tell the LLM **how to use the information gathered** and **enforce a naming convention** for the generated files. Following the approach used in the *Vulhub* project, the LLM is instructed to build a Docker-based environment starting from a `docker-compose.yml` file. A `Structured Output` schema is bonded to the LLM to enforce a common response format using the `Code` and `File` classes:

```python
class File(BaseModel):
    location: str = Field(description="Location of the file")
    content: str = Field(description="Content of the file")

class Code(BaseModel):
    files: list[File] = Field(description="Name and code of the various files
    needed to reproduce the CVE")
    directory_tree: str = Field(description="Directory tree where the files
    will be stored, rooted in the CVE-ID folder")
```

Listing 3.6: Definition of the `Code` and `File` classes

The formatted response is always an instance of the `Code` class. It saved in the `code.json` file inside the `dockers/CVE-ID/Web-Search-Mode/logs` directory, as instance of the `Code` class in the `State.code` field, and in the `final_report.txt` file for logging purposes.

```
This is the first version of the generated code:

Directory tree:
CVE-2021-28164/
|-- custom
    |-- docker-compose.yml
    `-- webapps
        |-- ROOT
        |   `-- index.jsp
        `-- WEB-INF
            |-- web.xml
            `-- secret.txt

---------- dockers/CVE-2021-28164/custom/docker-compose.yml ----------
version: '3'
services:
  jetty:
    image: jetty:9.4.38-jre8
    volumes:
      - ./webapps:/var/lib/jetty/webapps
    ports:
      - "8080:8080"
    environment:
      - JETTY_BASE=/var/lib/jetty
    command: ["java", "-jar", "/usr/local/jetty/start.jar"]



---------- dockers/CVE-2021-28164/custom/webapps/ROOT/index.jsp ----------
<html>
<head><title>Vulnerable Jetty Server</title></head>
<body>
<h1>Welcome to the Vulnerable Jetty Server</h1>
<p>This server is intentionally vulnerable to CVE-2021-28164 for testing purposes.</p>
</body>
</html>



---------- dockers/CVE-2021-28164/custom/webapps/WEB-INF/web.xml ----------
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
```

```
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
         http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">

    <display-name>Vulnerable Web Application</display-name>

    <servlet>
        <servlet-name>ExampleServlet</servlet-name>
        <jsp-file>/index.jsp</jsp-file>
    </servlet>

    <servlet-mapping>
        <servlet-name>ExampleServlet</servlet-name>
        <url-pattern>/example</url-pattern>
    </servlet-mapping>

</web-app>


---------- dockers/CVE-2021-28164/custom/webapps/WEB-INF/secret.
    txt ----------
This is a secret file that should not be accessible.
```

Listing 3.7: Example of code generation results in the `final_report.txt` file

Save Code

The second step is quite simple: it **saves all the files generated by the LLM** in the `dockers/CVE-ID/Web-Search-Mode` directory. This is done by looping through the list of `File` instances of the `State.code` field. Each file is associated to a location in the file system - representing the **absolute path** where the file is stored - and to its content.

Test Code

The hardest challenge that was faced while developing this AI agent was understanding when an virtual environment was working correctly. In the end, it was decided that for a virtual environment to be working correctly, it has to **reach all of the following four goals**:

1. **All Docker Images complete the build process without errors** (tracked by the `docker_builds` milestones)

2. **All Docker Containers can run without errors and without unexpected crashes** (tracked by the `docker_runs` milestones)

3. **All HARD-type services are hosted by the Docker Containers and vulnerable version of them is used** (tracked by the `code_hard_version` milestones)
4. **The services hosted by the Docker are reachable from their default network ports** (tracked by the `network_setup` milestones)

Testing starts by launching the Docker Compose environment with following command[4]:

```
docker compose up --build --detach
```

1. The first goal is validated by passing a **programmatic check** of the build process of the Docker Compose environment. Leveraging the `run` method of the `subprocess` Python library [92], the agent captures the output logs and checks if `returncode==0`, which means that **all Docker Images have been built or pulled successfully**. The output logs of the entire build process are saved in the `logX.txt`[5] file inside the `dockers/CVE-ID/Web-Search-Mode/logs` directory. If this check is passed, then `docker_builds=True`. Else (`docker_builds=False`) there was an error in the build process of the Docker Compose environment. Consequently, testing is terminated and the workflow is redirected to the revision step with the following failure explanation and revision goal:
   - **Failure Explanation**: the Docker Compose environment terminates its execution because of an **error while building/pulling at least one of its Docker Images**.
   - **Revision Goal**: fix the Docker Compose environment by modifying its code. To understand what went wrong, the **LLM has to analyse the output logs of the image build process**. To prevent context window saturation, only the **last 100 lines** of the output logs are passed, as they typically contain the **most relevant error information**.

[4]See the `launch_docker` function in Appendix A for details

[5]The `X` is just a placeholder that identifies the current iteration of the testing–revision loop of the virtual environment

2. To validate the second goal, the following information about each Docker Container is gathered:

   - **Container Startup Logs**, which are obtained with the command[6]:

     ```
     docker logs [Docker Container ID] --details
     ```

   - **Container Inspect Details**, which are obtained with the command[7]:

     ```
     docker inspect [Docker Container ID]
     ```

   The outputs of both commands are stored entirely in the `logX.txt` file. Validation is achieved by having each Docker Container undergo a programmatic check and an LLM-as-a-Judge evaluation:

   - **Programmatic Check**: uses the Container Inspect Details to check if it is running. Else, testing is terminated and the workflow is redirected to the revision step with the following failure explanation and revision goal:
     - **Failure Explanation**: one of the Docker Containers of the Docker Compose environment is not running correctly.
     - **Revision Goal**: fix the Docker Compose environment by modifying its code. To understand what went wrong, the **LLM has to analyse the output logs of the container startup process**. To prevent context window saturation, only the **last 100 lines** of both `stdout` and `stderr` are passed, as they typically contain the **most relevant error information**.
   - **LLM-as-a-Judge**: if the programmatic check has not revealed any issues, an LLM-as-a-Judge is tasked with evaluating the stability of the container. This is done with the `CHECK_CONTAINER_PROMPT` which provides the LLM with both the Container Startup Logs and Container Inspect Details:

     ```
     GOAL: check the following outputs and tell me if it the Docker
         container is running correctly ('container_ok' milestone)

     - Output of the 'docker logs [CONTAINER ID] --details' command: {
         container_log}

     - Output of the 'docker inspect [CONTAINED-ID]' command: {
         inspect_container_log}
     ```

[6]See the function `get_container_logs` in Appendix A for details

[7]See the function `inspect_container` in Appendix A for details

The Structured Output of the LLM-as-a-Judge is defined with the ContainerLogsAssessment class.

```python
class ContainerLogsAssessment(BaseModel):
    container_ok: bool = Field(description="Is the Docker container running correctly?")
    fail_explanation: Optional[str] = Field(description="Detailed explanation of the error presented by the logs")
```

Listing 3.8: Definition of the ContainerLogsAssessment class

If the field container_ok=False, testing is terminated and the workflow is redirected to the revision step with the following failure explanation and revision goal:

– **Failure Explanation**: the fail_explanation field of the LLM response provided a detailed explanation of why the LLM thinks the container is not working properly. This explanation is stored inside the final_report.txt file for logging purposes.

```
CONTAINER FAILURE (LLM-as-a-Judge Check): The Docker
    container is running, as indicated by the 'running'
    status and 'healthy' health check in the 'docker
    inspect' output. However, there are repeated 'Access
     denied for user 'root'@'localhost' (using password:
     NO)' errors in both the logs and the health check
    outputs. This suggests that the root user is trying
    to connect without a password, which is failing.
    This is a configuration issue that needs to be
    addressed for proper operation.
```

Listing 3.9: Example of LLM-as-a-Judge container failure explanation in the final_report.txt file

– **Revision Goal**: fix the Docker Compose environment by modifying its code. There is no need to understand what went wrong because the failure explanation is already given by the LLM-as-a-Judge.

3. The third goal is evaluated only using a LLM-as-a-Judge. To understand if the Docker Compose environment is using the right HARD-type services and that a vulnerable version is used, the LLM is provided with: the SYSTEM_PROMPT; the **Image Inspect Details**; the entire code-base of the environment; and the CHECK_SERVICES_VERSIONS_PROMPT:

```
GOALS: analyse the output of the command 'docker inspect [IMAGE-ID]'
    and the code of the Docker Compose environment (all contained in the
     previous messages) to assert if
- the following services are using one of the versions listed to their
    side ('code_hard_version' milestone):{hard_service_versions}
- the Docker uses the following services: {service_list} ('services_ok'
     milestone)

CONTEXT: the version lists of each service may contain multiple entries
     separated by ','

GUIDELINES: if any of the milestones is not achieved, you must explain
    why the Docker fails to achieve them and set to false the
    corresponding flag
```

This prompt leverages the data stored in the `State.web_search_results` field to provide the LLM with the list of `HARD`-type services and their vulnerable versions. Additionally, it also asks the LLM to check if all the services suggested in the information gathering phase - i.e., both `HARD`-type and `SOFT`-type ones - are actually present in the environment. The Image Inspect Details are given to the LLM because they provide relevant information about which services and versions are used by the environment. These are obtained with the command[8]:

```
docker inspect [Docker Image ID]
```

The final evaluation of the LLM-as-a-Judge is always an instance of the `ServiceAssessment` class, which is linked via `Structured Output`.

```python
class ServiceAssessment(BaseModel):
    code_hard_version: bool = Field(description="Does the generated code
    use vulnerable version of the 'HARD' services?")
    services_ok: bool = Field(description="Does the generated code contain
     the services provided by the web search?")
    fail_explanation: Optional[str] = Field(description="Detailed
    explanation of why one or more milestones have failed")
```

Listing 3.10: Definition of the `ServiceAssessment` class

If the field `code_hard_version=False`, testing is terminated and the workflow is redirected to the revision step with the following failure explanation and revision goal:

[8]See the function `inspect_image` in Appendix A for details

- **Failure Explanation**: the `fail_explanation` field of the LLM response explains which `HARD`-type service is not using a vulnerable version. This explanation is stored inside the `final_report.txt` file for logging purposes.

```
NOT VULNERABLE VERSION (LLM-as-a-Judge Check): The Docker
    setup uses CouchDB version 3.2.2, which is not in the
    list of vulnerable versions specified for the '
    code_hard_version' milestone. Therefore, the '
    code_hard_version' milestone is not achieved.
```

Listing 3.11: Example of LLM-as-a-Judge non-vulnerable version failure explanation in the `final_report.txt` file

- **Revision Goal**: fix the Docker Compose environment by modifying its code. There is no need to understand what went wrong because the failure explanation is already given by the LLM-as-a-Judge.

4. Finally, the fourth goal is also evaluated exclusively via LLM-as-a-Judge. To understand if each service of the Docker Compose environment is exposing the right network ports, the LLM is provided with: the `SYSTEM_PROMPT`; the **Container Inspect Details** of all containers; the entire code-base of the environment; and the `CHECK_NETWORK_PROMPT`.

```
GOALS: analyse the output of the command 'docker inspect [CONTAINED-ID]
    ' and the code-base of the Docker Compose environment (all contained
     in the previous messages) to assert if all services are using their
     default network port ('network_setup' milestone)

GUIDELINES: if the milestones is not achieved, you must explain why the
     Docker fails to achieve them and set to false the corresponding
    flag
```

The prompt provides the LLM with the Container Inspect Details because it contains relevant information about which the actual networking setup of a running container. The final evaluation of the LLM-as-a-Judge is an instance of the `NetworkAssessment` class, which is linked to the LLM via `Structured Output`.

```python
class NetworkAssessment(BaseModel):
    network_setup: bool = Field(description="Are all services/containers
    setup to be accessible from the right network ports?")
    fail_explanation: Optional[str] = Field(description="Detailed
    explanation of why one or more milestones have failed")
```

Listing 3.12: Definition of the `NetworkAssessment` class

If the field `network_setup=False`, testing is terminated and the workflow is redirected to the revision step with the following failure explanation and revision goal:

- **Failure Explanation**: the `fail_explanation` field of the LLM response explains why it thinks the network configuration of the Docker Compose environment is wrong. This explanation is stored inside the `final_report.txt` file for logging purposes.

```
WRONG NETWORK SETUP (LLM-as-a-Judge Check): The Docker
    container is exposing the Apache service on port 8080
    instead of the default port 80. This is evident in the
    'docker-compose.yml' file where the port mapping is
    defined as '8080:80', indicating that the container's
    port 80 is being mapped to host port 8080.
```

Listing 3.13: Example of LLM-as-a-Judge network misconfiguration failure explanation in the `final_report.txt` file

- **Revision Goal**: fix the Docker Compose environment by modifying its code. There is no need to understand what went wrong because the failure explanation is already given by the LLM-as-a-Judge.

If all the goals are met, the agent concludes that **the generated virtual environment is working correctly**, and the workflow continues with the vulnerability assessment phase. Else, if any goal fails, the agent tries to **fix the identified issues** using the knowledge gathered during testing. To regulate the total workflow execution time and reduce costs, the **number of iterations** the agent may perform in the **testing–revision loop is limited to 10**.

Revise Code

Revision of the environment code-base starts by shutting down the running Docker Compose environment and by removing all the related Docker Containers and Docker Volumes. This is done with the `down_docker` function[9] to **free up memory** and to **sanitize the entire Docker Desktop environment**.

```
docker compose down --volumes
```

A 10 second sleep timer is started at the end of the function to ensure the **complete cleanup of the system** and the **correct shutdown of all Docker Containers**. Additionally, if the test failed because a non-vulnerable version of a `HARD`-type service was used, all Docker Images are also removed from the system using the `remove_all_images` function[10]. This is done to prevent Docker from reusing them instead of building/pulling fresh ones.

```
docker rmi -f [List of Docker Image IDs]
```

Due to some problems in the implementation of the `Structured Output` and intrinsic limitations of the models, the **revision process changes based on which LLM is used**:

- For `GPT-4o` and `GPT-5`, the revision process is performed with a single prompt named `TEST_FAIL_PROMPT`, which is very similar to the one used in the code generation step:

```
CONTEXT: {fail_explanation}

GOALS: {revision_goal}.

GUIDELINES:
- Any DB must are properly setup and populated with some test data
- The system must be immediately deployable using the "docker compose
   up" command
- Ensure that no service has to be setup manually by the user
- All services and related containers must be properly configured on
   order to be immediately accessible from the default network ports of
    the service
- Your answer must include all files (updated ones, unchanged ones and
   new ones)
- All file names must indicate the file path which must start with "
   ./../../dockers/{cve_id}/{mode}"
- There is no need to specify the file name in the file content
```

[9]See the function `down_docker` in Appendix A for details

[10]See the function `remove_all_images` in Appendix A for details

```
- The Docker code was generated using the data in the message about {
    cve_id} and its services
    - You must use all and only the services that are listed in the
    message that describes {cve_id}
    - If a service requires a dedicated container write the code for it
    - You must not use versions of 'HARD' services that are not listed
    in the message about {cve_id} and its services
- Here is the list of previous fixes that you attempted but did not
    work, my suggestion is to try something different from: {fixes}
```

Differently from the `CODING_PROMPT`, this prompt contains also: which goal the virtual environment has failed the reach - i.e., `fail_explanation`; what the LLM has to do to understand what went wrong and fix the issue - i.e., `revision_goal`; and a list containing all previous fix attempts made by the agent - i.e., `fixes`.

Before the invocation, the LLM is provided with the `SYSTEM_PROMPT`, the information gathered in the previous task, the entire code-base of the virtual environment - where the LLM is instructed to operate in order to fix the issue, and a `HumanMessage` containing the `TEST_FAIL_PROMPT`.

- To solve performance issues and mitigate the frequent hallucinations that plagued `gpt-oss:120B`, it was decided to split the revision step into two:

  1. **Revision**: the LLM is asked to **explain how to fix the problem by modifying the given code**. To do this, the LLM is provided with the `SYSTEM_PROMPT`, the information gathered in the previous task, the entire code-base of the virtual environment, and a `HumanMessage` containing the `REVISION_PROMPT`:

     ```
     CONTEXT: {fail_explanation}

     GOALS: explain how you would fix this problem by modifying the code
         in a few sentences, do not use bullet points.
     ```

     This prompt is used to asks the LLM to look at the code and explain how it can be fixed knowing what is wrong with the Docker Compose environment it is trying to create.

2. **Code Correction**: the LLM is asked to apply the fix produced in the last phase. To do this, the LLM is provided with the same data, expect for the final `HumanMessage`, which holds instead the `CODE_CORRECTION_PROMPT`:

```
CONTEXT: {fail_explanation}

GOALS: {revision_goal}. {fix}.

GUIDELINES:
- Any DB must are properly setup and populated with some test data
- The system must be immediately deployable using the "docker
    compose up" command
- Ensure that no service has to be setup manually by the user
- All services and related containers must be properly configured
    on order to be immediately accessible from the default network
    ports of the service
- Your answer must include all files (updated ones, unchanged ones
    and new ones)
- All file names must indicate the file path which must start with
    "./../../dockers/{cve_id}/{mode}"
- There is no need to specify the file name in the file content
- The Docker code was generated using the data in the message about
     {cve_id} and its services
    - You must use all and only the services that are listed in the
     message that describes {cve_id}
    - If a service requires a dedicated container write the code
    for it
    - You must not use versions of 'HARD' services that are not
    listed in the message about {cve_id} and its services
- Here is the list of previous fixes that you attempted but did not
     work, my suggestion is to try something different from {fixes}
```

   Differently from the `TEST_FAIL_PROMPT`, the `CODE_CORRECTION_PROMPT` explicitly **tells the LLM how to modify the code to fix the problem**. The fix suggestion is contained in the `fix` field and it contains the answer that the LLM provided at the end of the *Revision* phase.

Whichever model is used, the final LLM output will always be an instance of the `CodeRevision` class, which also contains the fixed code.

```python
class CodeRevision(BaseModel):
    error: str = Field(description="Detailed description of the error presented
     by the logs")
    fix: str = Field(description="Detailed description of fix applied to the
    code to solve the error")
    fixed_code: Code = Field(description="File location, content and associated
     directory tree")
```

Listing 3.14: Definition of the `CodeRevision` class

The `fixed_code` replaces the one stored in the `State.code` field and it will be **saved at the beginning of the next iteration** in the local directory, replacing the current files. The suggested fix will be appended to the `State.fixes` and will contribute to the short-term memory of the agent, hopefully **stopping it from suggesting a fix that has not worked previously**. Meanwhile, the `error` and `fix` fields are saved inside the `final_report.txt` file to help **traceback the testing error pattern**:

```
Test iteration #0 failed! See 'log0.txt' for details.
  - IMAGE BUILDING FAILURE (Manual Check)
  - ERROR: The error in the logs indicates that there is a problem
     pulling the Jetty image from Docker Hub. The message 'pull
    access denied for eclipse/jetty, repository does not exist or
    may require 'docker login'' suggests that either the image
    version is incorrect or authentication is needed.
  - FIX: To resolve this issue, change the Jetty image version to
    one that is publicly available and does not require
    authentication. According to the CVE description, we should use
     'eclipse/jetty:9.4.38.v20210224', which is a valid and
    accessible version. Additionally, remove the obsolete 'version'
     attribute from the docker-compose.yml file.
```

Listing 3.15: Example of failed test and code revision results in the `final_report.txt` file

3.5.4 Vulnerability Assessment

The fourth and final task of the agentic workflow examines the virtual environment generated in the last step to check if it is vulnerable to the input CVE.

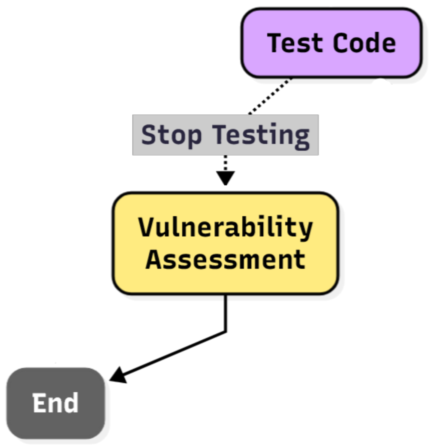


Figure 3.6: Graph Representation of the Fourth Task of the Agentic Workflow

Docker Scout [65] is the tool the is used to perform security analysis. It performs a **vulnerability scan**[11] that produces, for each Docker Image used by the Docker Compose environment, a list of all vulnerabilities which the Docker Image introduces in the environment. This is achieved by running the command:

```
docker scout cves [Docker Image ID] --format gitlab
```

The output is saved the `cvesX.json`[12] which is stored in the `dockers/CVE-ID/Web-Search-Mode/logs` directory. If **at least one** of these lists contains the input CVE, then it can be *assumed*[13] that the agent managed to create **vulnerable virtual environment**.

Since Docker Scout only requires a correctly built Docker Image to start a vulnerability scan, it was decided to allow the workflow to proceed to the vulnerability assessment phase **as long as the `docker_builds` milestone is satisfied** - i.e., regardless of whether other milestones are completed.

[11]See the function `run_docker_scout` in Appendix A for details

[12]The `X` is just a placeholder that identifies the a Docker Image - e.g., if the Docker Compose environment uses 2 Docker Images, the directory will contain `cves0.json` and `cves1.json`

[13]Static analysis tool, like Docker Scout, risk producing both **false positives** and **false negatives**

Chapter 4

Results & Evaluation

This chapter will explore relevant results that were discovered while testing the AI agent described in the previous chapter.

4.1 Experiment Setup

To initiate a run, the AI agent has to be provided with **four user inputs**:

- The **CVE-ID** identifies the vulnerability around which the virtual environment has to be built.
- The **information gathering strategy** determines how the agent will acquire information about which services have to be included in the virtual environment. There are three possible strategies: **OpenAI Search**, **Custom Search**, and **CVE-based Search**. **OpenAI Search** and **Custom Search** give the agent a limited degree of freedom in how it conducts information gathering; therefore, in the following analyses they are categorized as **Autonomous Strategies**. In contrast, **CVE-based Search** is classified as a **Fixed Strategy** because the agent must perform information gathering using a predetermined approach and predefined parameters.
- The **name of the LLM** identifies the LLM that will be used to automate the virtual environment creation process.
- (*Optional*) The **verbosity parameter** can be set to `True` to provide additional data during the information gathering phase.[1]

[1]All experiments were performed with `verbose_web_search=False`

4.1.1 Agent Development Dataset

Much of the development of the AI agent was supported by the use of the **Agent Development Dataset**, a small dataset of 20 CVEs taken from the private `VDaaS` repository[2] of the `SmartData@Polito` *Research Group*.

CVE	Attack Type	Service	Version
2012-1823	Remote Code Execution	PHP CGI	5.4.1-cgi
2016-5734	Remote Code Execution	phpMyAdmin	4.4.15.6
2018-12613	Remote File Inclusion	phpMyAdmin	4.8.1
2020-7247	Remote Code Execution	OpenSMTPD	6.6.1
2020-11651	Remote Code Execution	SaltStack	2019.2.3
2020-11652	Remote Code Execution	SaltStack	2019.2.3
2021-3129	Remote Code Execution	Laravel	8.4.1
2021-28164	Information Disclosure	Jetty	9.4.37
2021-34429	Sensitive File Disclosure	Jetty	9.4.40
2021-41773	Remote Code Execution	HTTPD	2.4.49
2021-42013	Remote Code Execution	HTTPD	2.4.50
2021-43798	Directory Traversal & File Read	Grafana	8.2.6
2021-44228	Remote Code Execution	Log4j	2.0-beta9
2022-22947	Remote Code Execution	Spring Cloud Gateway	3.1.0
2022-22963	Remote Code Execution	Spring Cloud	3.2.2
2022-24706	Remote Code Execution	CouchDB	3.2.1
2022-46169	Remote Code Execution	Cacti	1.2.22
2023-23752	Remote Command Execution	Joomla	4.2.7
2023-42793	Remote Command Execution	JetBrains TeamCity	2023.05.3
2024-23897	Local File Inclusion	Jenkins	13.10.0

Table 4.1: Agent Development Dataset CVE List

These 20 CVEs had a crucial role, as they were used to estimate the performance of the AI agent and to guide its design.

4.1.2 Models Tested

These are the *OpenAI*'s **Generative Pretrained Transformer** (**GPT**) models that were tested during the development of the AI agent:

- `GPT-4o` is a multimodal GPT model with a context window of 128k tokens and maximum output size capped at 16384 tokens. The `"o"` in `GPT-4o` stands

[2]This is a **private Github repository**, authorization is required to view its contents (https://github.com/SmartData-Polito/VDaaS/tree/main)

for *omni* and highlights the **multimodal capabilities** of the model. Being "*multimodal*" means that the model inputs containing a mixture of text, audio, image and video data. This model was used during the whole development process of the AI agent, therefore, its behaviour and outputs were responsible for much of the current prompt configuration. A key for *OpenAI's API* was provided by the `SmartData@Polito` *Research Group* to use this model. Here is reported the **pricing per 1M tokens** [93] (prices may have been subject to variations):

  - **Per 1M Input Tokens**: $2.50
  - **Per 1M Output Tokens**: $10.00

  The `GPT-4o` model does **not support reasoning tokens** and its **knowledge cutoff date** is October 1st, 2023.

- `GPT-5` is the latest GPT model of *OpenAI*, with a context window of 400k tokens and maximum output size of 128k tokens. The same key for *OpenAI's API* provided by the `SmartData@Polito` *Research Group* was used to interact with this model. Here is reported the **pricing per 1M tokens** [93] (prices may have been subject to variations):

  - **Per 1M Input Tokens**: $1.25
  - **Per 1M Output Tokens**: $10.00

  The `GPT-5` model **support reasoning tokens** and its **knowledge cutoff date** is September 30th, 2024.

- `gpt-oss:120B` is a text-to-text open-weight model composed of 117B parameters with 5.1B of them being **active parameters**. Both context window and maximum output size are capped at 131072 tokens. This model was running on the local **Graphics Processing Unit** (**GPU**) of the `SmartData@Polito` *Research Group*. The `gpt-oss:120B` model **support reasoning tokens** and its **knowledge cutoff date** is June 1st, 2024.

Interactions with these models were accomplished with *OpenAI*'s `Chat Completion API`, which is supported by all models.

LLM	**# Parameter**	**Knowledge Cutoff**	**Reasoning**	**Location**
`GPT-4o`	200B	October 1st, 2023	✗	Remote
`GPT-5`	*Undisclosed*	September 30th, 2024	✓	Remote
`gpt-oss:120B`	117B	June 1st, 2024	✓	Local

Table 4.2: Relevant Information about the Tested LLMs

4.2 Evaluation Criteria

4.2.1 Milestones

This set of **eight milestones** is distributed across the four main tasks that characterise the agentic workflow. Their goal is to help keep track of the progress of a run.

Milestone	Workflow Location	Description
`cve_id_ok`	▼ CVE Assessment	If `True`, the CVE identifier provided by the user corresponds to an entry in the NVD dataset [64]
`hard_service`	● Assess CVE Services	If `True`, all `HARD`-type services have been correctly identified
`hard_version`	● Assess CVE Services	If `True`, the "*expected*" vulnerable version of all `HARD`-type services have been correctly identified
`soft_services`	● Assess CVE Services	If `True`, at least one service has been suggested for each "*expected*" `SOFT-<role>`
`docker_builds`	■ Test Code	If `True`, all Docker Images have completed the build process without errors
`docker_runs`	■ Test Code	If `True`, all Docker Containers are running without errors or unexpected crashes
`code_hard_version`	■ Test Code	If `True`, all `HARD`-type services are hosted by the Docker Compose environment and vulnerable version is used
`network_setup`	■ Test Code	If `True`, all the services hosted by the Docker Compose environment are reachable from their default network ports

Table 4.3: Agentic Workflow Milestones

- **Information Gathering**: if all the milestones located in ● Assess CVE Services are completed successfully during a run, **the agent managed to collect the *expected* information about the input CVE**.
- **Environment Generation**: if all the milestones located in ■ Test Code are completed successfully during a run, **the agent managed to create a functional virtual environment**.

4.2.2 Data Gathered

This is all the data that the AI agent is currently capable of gathering during one of its runs:

Name	**Type**	**Description**
`test_iteration`	Integer	Number of iterations of the test loop
`starting_image_builds`	Boolean	If `True`, the `docker_builds` milestone is passed at the first iteration
`image_build_failures`	Integer	Number of times the `docker_builds` milestone fails
`starting_container_runs`	Boolean	If `True`, the `docker_runs` milestone is passed at the first iteration
`container_run_failures`	Integer	Number of times the `docker_runs` milestone fails
`not_vuln_version_fail`	Integer	Number of times the `code_hard_version` milestone fails
`docker_misconfigured`	Integer	Number of times the `network_setup` milestone fails
`num_containers`	Integer	Number of Docker Containers used by the working Docker Compose environment
`docker_scout_vulnerable`	Boolean	If `True`, Docker Scout confirmed that the Docker Compose environment is vulnerable to the input CVE
`exploitable`	Boolean	If `True`, the exploit was successfully executed on the Docker Compose environment
`services_ok`	Boolean	If `True`, the Docker Compose environment uses all the services suggested during the information gathering phase
`requires_manual_setup`	Boolean	If `True`, one of the services of the Docker Compose environment require some sort of manual operation to work properly

Table 4.4: Agentic Workflow Statistics

4.3 Agent Performance Analysis

4.3.1 Information Gathering Performance

This first analysis compares the three information gathering strategies to determine which one delivers the **best performance in the information gathering task**. The performance of a strategy is measured by counting how many runs successfully retrieved the *expected*[3] information about each input CVE. To assess this, it was decided to perform **one run per CVE** — across all 20 CVEs of the Agent Development Dataset — for each strategy. The entire experiment was repeated using `GPT-4o`, `GPT-5`, and `gpt-oss:120B`[4]. Detailed logs of all runs are available in the Github repository of the project.[5]

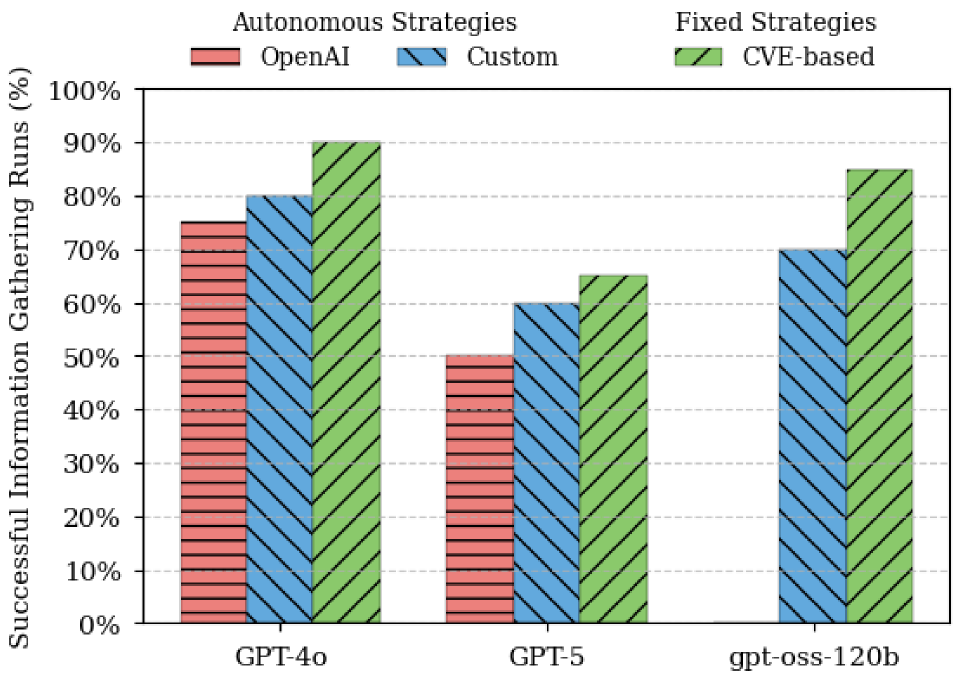


Figure 4.1: Information Gathering Task, Model Performance Across Strategies

[3]See Table 4.3 to understand which milestones must be satisfied for the **information gathering** task to be deemed successful

[4]`gpt-oss:120B` has **no data** for the **OpenAI Search** strategy because it is **not compatible**

[5]https://github.com/g1san/Agents-for-Vulnerable-Dockers-and-related-Benchmarks
See the `GPT-4o` 5th benchmark session, `GPT-5` 3rd benchmark session, and `gpt-oss-120b` 2nd benchmark session

Figure 4.1 clearly shows that `GPT-4o` delivers the best performance across all strategies. It also shows that - regardless of the model - **the agent performs best when using the CVE-based Search** strategy. This indicates that the agent does not benefit from the freedom of choosing the `query` parameter granted by the **Custom Search**. In fact, this additional flexibility may lead to **lower-quality information sources** by encouraging the generation of **overly complex search queries**.

It was observed that **smaller LLMs**, like `GPT-4o` and `gpt-oss:120B`, tend to generate **simpler search queries** - e.g., by adding the word "*details*" and/or "*vulnerability*" to the CVE-ID - whereas **larger LLMs**, such as `GPT-5`, often generate **much longer and convoluted queries**.

```
CVE-2020-11652 SaltStack Salt directory traversal unauthenticated
    RCE wheel client\_acl tokens root key disclosure mitre

CVE-2021-42013 Apache HTTP Server 2.4.49 2.4.50 path traversal RCE
     details exploit mitigation timeline

CVE-2023-42793 TeamCity authentication bypass RCE details
    exploited JetBrains advisory mitigation versions fixed
    exploitation groups timeline indicators of compromise PoC
```

Listing 4.1: Example of search queries generated by `GPT-5` whose runs did not manage to satisfy any of the milestones

This is especially evident in the queries of `GPT-5`, which frequently target **Proof-of-Concept (PoC)** material or **exploits**, rather than focusing on which services should be included in the Docker-based environment.

This tendency of `GPT-5` to **overcomplicate** its queries likely also explain its poor performance with **OpenAI Search**, where it achieves only **50%**, the lowest score among all models. In comparison, `GPT-4o` achieves **75%** with **OpenAI Search**, suggesting that this **lack of performance** is due solely to the way reasoning models perform web searches with the `web-search-preview` built-in tool. In other words, the use of an agentic approach while performing the web search is **not well suited** to successfully complete tasks as simple as this one.

Ultimately, these results reinforce the notion that the **information gathering task does not benefit from providing the AI agent with agency** - i.e., the freedom of choosings how to structure the web search (**OpenAI Search**) or even just the `query` parameter (**Custom Search**) - and that local models like `gpt-oss:120B` can serve as **viable alternatives to proprietary models** for this type of task, encouraging a more streamlined use of the agent.

4.3.2 Environment Generation Performance

The second analysis closely follows the first one by comparing how information gathered with a different strategy influences the **performance of the agent in the environment generation task**. Using the same experiments[6] performed with `GPT-4o`, `GPT-5`, and `gpt-oss:120B`[7], the performance of the agent in the environment generation task is measured by **counting how many runs successfully generated a *functional*[8] virtual environment**.

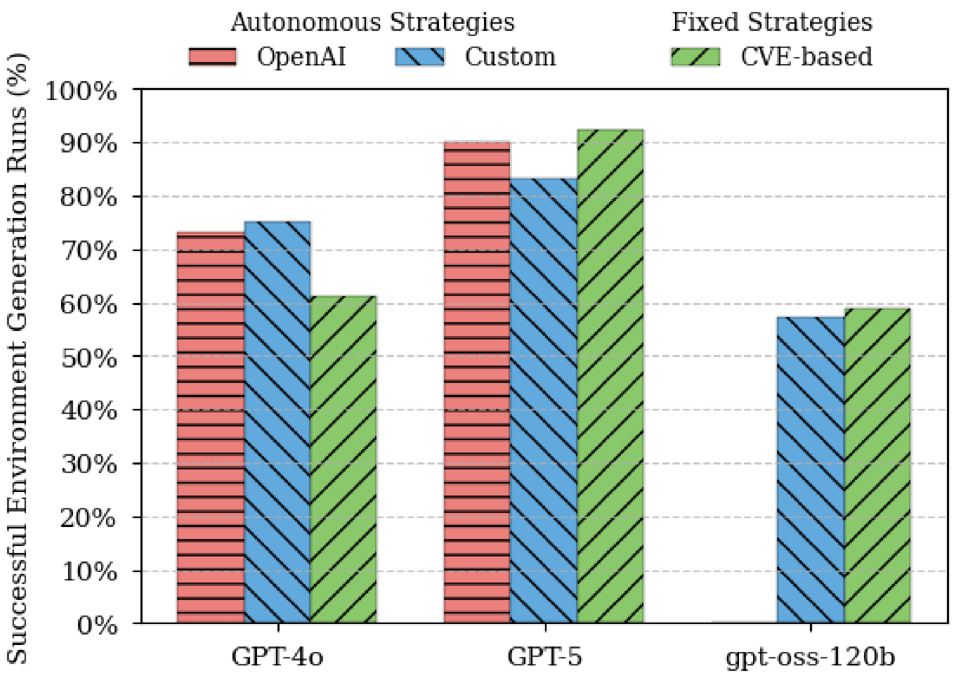


Figure 4.2: Environment Generation Task, Model Performance Across Strategies. Only the Runs where the Environment Generation Task Started are Considered

It is important to remember that the AI agent is programmed to terminate its run whenever it fails to gather the *expected* information in the information gathering task. Accounting for the failed information gathering runs would have made the environment generation task **look harder then it actually is**.

[6]https://github.com/g1san/Agents-for-Vulnerable-Dockers-and-related-Benchmarks
See the `GPT-4o` 5th benchmark session, `GPT-5` 3rd benchmark session, and `gpt-oss-120b` 2nd benchmark session

[7]`gpt-oss:120B` has **no data** for the **OpenAI Search** strategy because it is **not compatible**

[8]See Table 4.3 to understand which milestones must be satisfied for the **environment generation** task to be deemed successful

Results show that `GPT-5` has the strongest performance for all strategies: **90%** for **OpenAI Search**, **83%** for **Custom Search**, and **92%** for **CVE-based Search** (best overall). The main explanation for these results is given by the fact that `GPT-5` **underperformed in the information gathering task**, especially when compared to `GPT-4o`, consequently reducing the number of runs that actually started the environment generation task.

The performance of `GPT-4o` is also **very good**, especially considering the high number of runs that started the environment generation task: **73%** for **OpenAI Search**, **75%** for **Custom Search**, and **61%** for **CVE-based Search**. What stands out the most here is the **performance inversion** of the **CVE-based Search** strategy when compared to the information gathering task: **CVE-based Search** was the best strategy for `GPT-4o` (**90%**), but when considering the runs where the environment generation started, **CVE-based Search** becomes the worst one (**61%**). This indicates that the **CVE-based Search** strategy can lead to **struggles** in the environment generation task.

Meanwhile, `gpt-oss-120B` clearly **lags behind both proprietary models**: **57%** for **Custom Search**, and **59%** for **CVE-based Search**. Its **smaller size** and other limiting factors given by the fact that the model is **running locally** may explain its lack of performance.

`Model`	**Strategy**	**Overall Success %**	**Condit. Success %**
`GPT-4o`	**OpenAI**	55%	73%
	Custom	60%	75%
	CVE-based	55%	61%
`GPT-5`	**OpenAI**	45%	90%
	Custom	50%	83%
	CVE-based	60%	92%
`gpt-oss:120B`	**OpenAI**	-	-
	Custom	40%	57%
	CVE-based	50%	59%

Table 4.5: Environment Generation Task, Model Overall and Conditional Success Rates Across Strategies

- **Overall Success Rate (%)**: computed by considering all runs performed by the agent, even those where the environment generation task did not start.
- **Conditional Success Rate (%)**: computed by considering only the runs where the environment generation task actually starts.

4.3.3 Vulnerability Assessment Results

This is the closing step of the analysis of the same experiments[9] performed with `GPT-4o`, `GPT-5`, and `gpt-oss:120B`[10], which wants to highlight how different models and strategies can influence the **ability of the agent to generate a vulnerable environment**. Vulnerability assessment data is obtained using the Docker Scout tool [65] to assess if a virtual environment generated by the AI agent is **also vulnerable**. An attempt to **perform manual exploitation on each generated environment** was also made. Some exploit succeeded even when the environment was not identified as vulnerable by Docker Scout. However, it was decided to **not merge** the exploitation results with the ones of Docker Scout. Therefore, these results **underestimate the actual performance of the agent**.

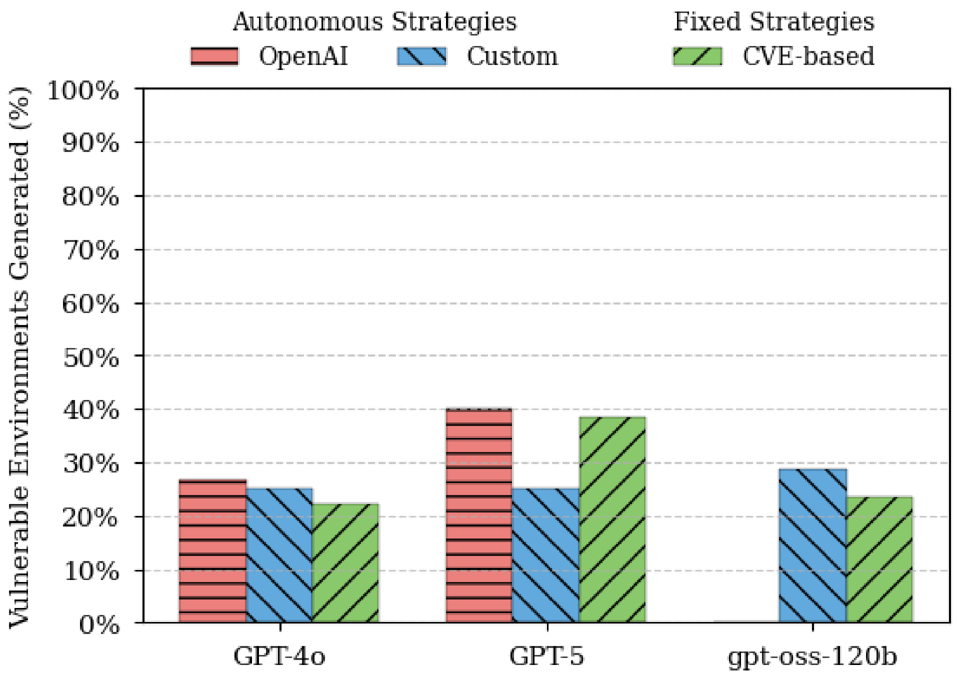


Figure 4.3: Vulnerability Assessment Task, Model Performance Across Strategies. Only the Runs where the Environment Generation Task Started are Considered

[9]https://github.com/g1san/Agents-for-Vulnerable-Dockers-and-related-Benchmarks
See the `GPT-4o` 5th benchmark session, `GPT-5` 3rd benchmark session, and `gpt-oss-120b` 2nd benchmark session

[10]`gpt-oss:120B` has **no data** for the **OpenAI Search** strategy because it is **not compatible**

Figure 4.3 matches the previous results by showing that `GPT-5` is clearly the **best model at reproducing vulnerable virtual environments**, reaching its highest performance (**40%**) when considering the environments generated with the **OpenAI Search** strategy.

Both `GPT-4o` and `gpt-oss-120B` achieve similar results. The vulnerability rate of their environments sits around **22–29%**, meaning that only about a quarter of them ended were confirmed as vulnerable. Overall, when compared to `GPT-5`, these models appear to **struggle more** in ensuring that the generated environments are vulnerable.

`Model`	**Strategy**	● **Inf. Gat.**	■ **Env. Gen.**	▲ **Vuln. Ass.**
`GPT-4o`	OpenAI	75%	73%	27%
	Custom	80%	75%	25%
	CVE-based	90%	61%	22%
`GPT-5`	OpenAI	50%	90%	40%
	Custom	60%	83%	25%
	CVE-based	65%	92%	38%
`gpt-oss:120B`	OpenAI	-	-	-
	Custom	70%	57%	29%
	CVE-based	85%	59%	24%

Table 4.6: Agentic Workflow Tasks, Model Performance Across Strategies

4.4 Agent Consistency Assessment

Consistency helps understanding how reliably the agent produces similar results across multiple runs, executed at **different times** and but under the **same conditions** - i.e., using the same input data. To better understand this characteristic of the agent, it was decided to perform **one run per CVE** of the Agent Development Dataset - for each strategy - totalling 60 runs per experiment. The experiment was conducted **three times, exclusively with `GPT-4o`**, as it was the best performer in the information gathering task. Detailed logs of all runs are available in the Github repository of the project.[11]

4.4.1 Information Gathering Consistency

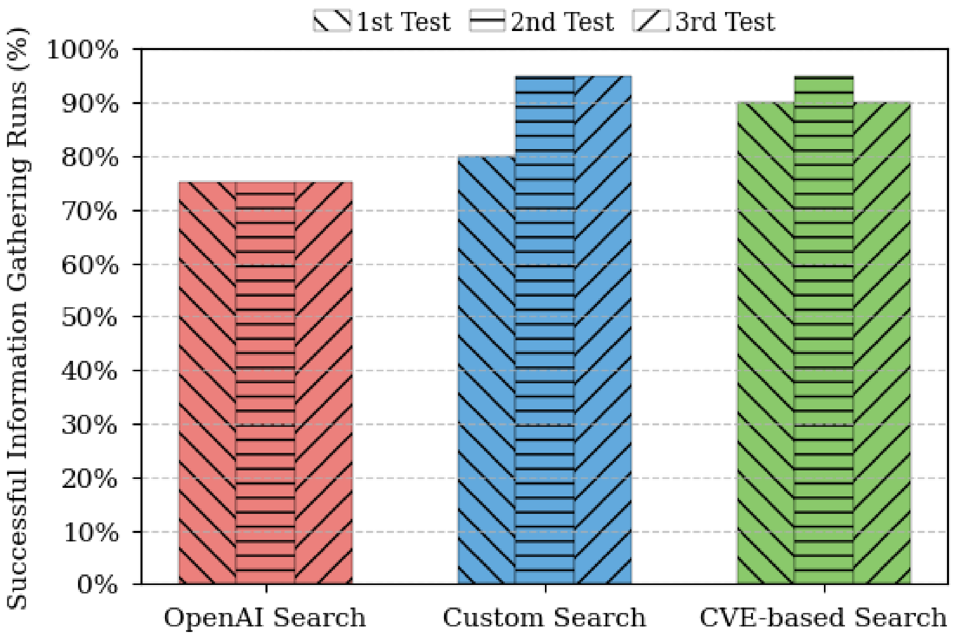


Figure 4.4: Information Gathering Task, `GPT-4o` Performance Consistency Across Strategies

Figure 4.4 shows that, with **OpenAI Search**, the agent **always** manages to successfully perform the information gathering task in **75%** of the runs. This makes **OpenAI Search** look **perfectly consistent**, but at the cost of having a **lower performance ceiling**, showing its clear inferiority when compared to the other two strategies.

[11]https://github.com/g1san/Agents-for-Vulnerable-Dockers-and-related-Benchmarks
See the `GPT-4o` 5th benchmark session (1st Test), `GPT-4o` 6th benchmark session (2nd Test), and `GPT-4o` 7th benchmark session (3rd Test)

The same **cannot** be said about the **Custom Search** strategy, which instead shows **better but inconsistent performance**, going from **80%** in the first test, to **95%** in the following ones, averaging a total of **90%**. This reinforces the notion that the agent does **not benefit** from the freedom of choosing the `query` parameter granted by the **Custom Search** strategy. The main takeaway of Figure 4.4 is that **CVE-based Search** is clearly the **best performing strategy** - averaging **≈91.7%** successful runs - and **almost perfectly consistent**, given its minimal performance variance (**90-95%**).

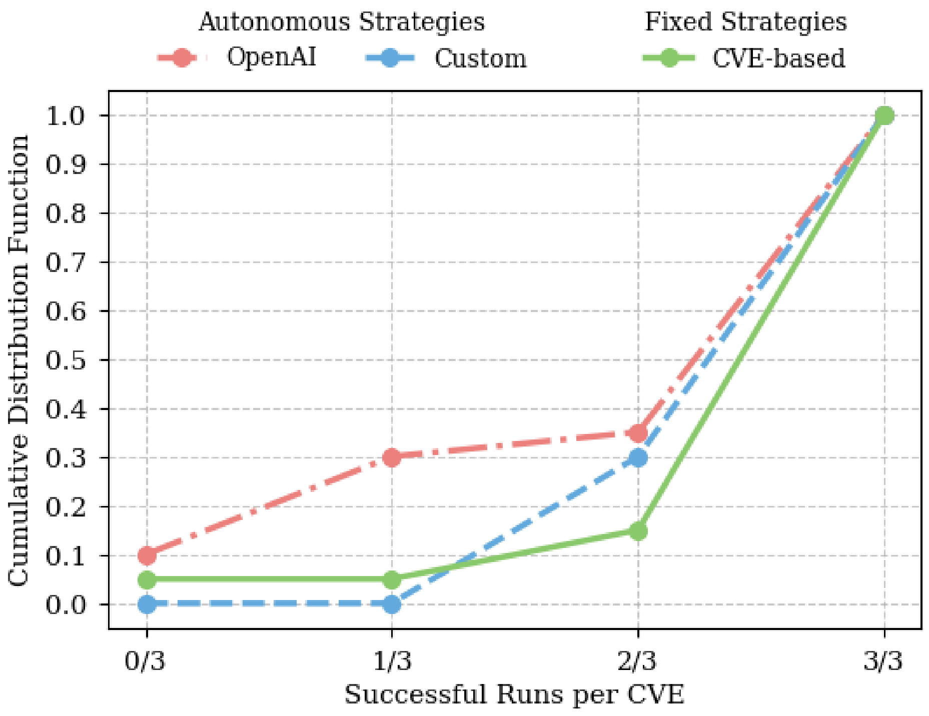


Figure 4.5: Information Gathering Task, `GPT-4o` Successful Runs per CVE, Cumulative Distribution Function Across Strategies

A deeper analysis on the consistency of each strategy can be made by combining the runs of all the three tests and counting, for each CVE, **how many times the information gathering task was successful**. By combining these data - shown in Figure 4.5 - with the logs generated during each agent run, it can be clearly identified that the main issue of the **CVE-based Search** occurs with CVE-2021-28164 [86]. The results show that the agent fails the information gathering task because the **LLM-as-a-Judge incorrectly evaluates the proposed version of the `HARD`-type service** — in this case, *Jetty* — across all three runs performed with the **CVE-based Search**, and not because it cannot find the expected information. This evaluation problem is caused by **versioning convention** of *Jetty*: the expected version is `9.4.37`, but the agent consistently proposes `9.4.37.v20210219`.

The LLM does not recognize these as equivalent, which leads to the failure of the `hard_version` milestone. If this **simplistic evaluation error** is taken into account, the performance of the **CVE-based Search** increases to an average of **≈96.7%**.

Figure 4.5 also shows some interesting insights on the performance on the **OpenAI Search** strategy. Only 13 CVEs have 3 out of 3 successful runs in the information gathering task, which means that the **true consistent performance** of the **OpenAI Search** strategy sits at around **65%**. The remaining **10%** is actually provided by a different subset of CVEs in each test.

4.4.2 Environment Generation Consistency

Following the same analysis approach taken previously, it is now time to analyse the performance of the three test in the environment generation task by **considering only those runs where the task started**.

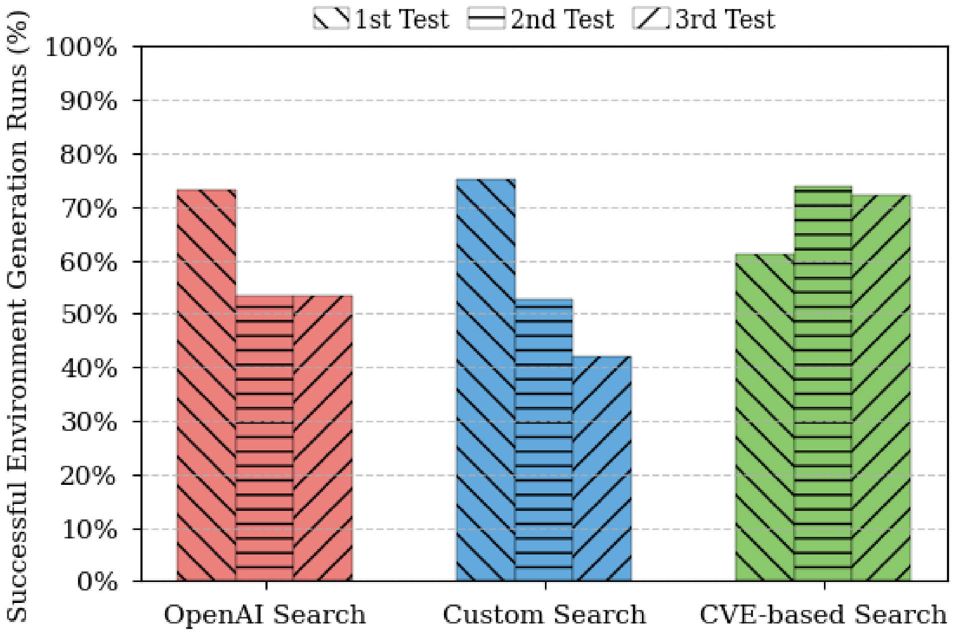


Figure 4.6: Environment Generation Task, `GPT-4o` Performance Consistency Across Strategies

Figure 4.6 shows that **environment generation is significantly more challenging than information gathering**. Even the best-performing approaches achieve only **72–75%** success, whereas for the information gathering task success rate can reach **perfect performance** — if we account for occasional LLM-as-a-judge evaluation errors.

What stands out most in Figure 4.6 is how the performance of all strategies fluctuates over the three different tests. In the first evaluation, `GPT-4o` performs best with **OpenAI Search** (73%) and **Custom Search** (75%). In the two subsequent tests, however, the performance of these two strategies drops, while the **CVE-based Search** matches their previous results with a **74%** success rate. **Custom Search** exhibits the **least stability**, dropping from 75% to 42%, while the **CVE-based Search** proves to be the **most consistent**, maintaining results between 61% and 74%. This reinforces the earlier observation that **variability in the `query` parameter can degrade the quality of information gathered by the agent** — a pattern now clearly visible in the results.

4.4.3 Vulnerability Assessment Results

Finally, the agent consistency analysis concludes with the vulnerability assessment results of the three test runs, which indicate **how many of the successfully generated environments are actually vulnerable**. Just like in the performance analysis, the data shown **underestimates the actual performance of the agent** because the manual exploitation results are **not included**.

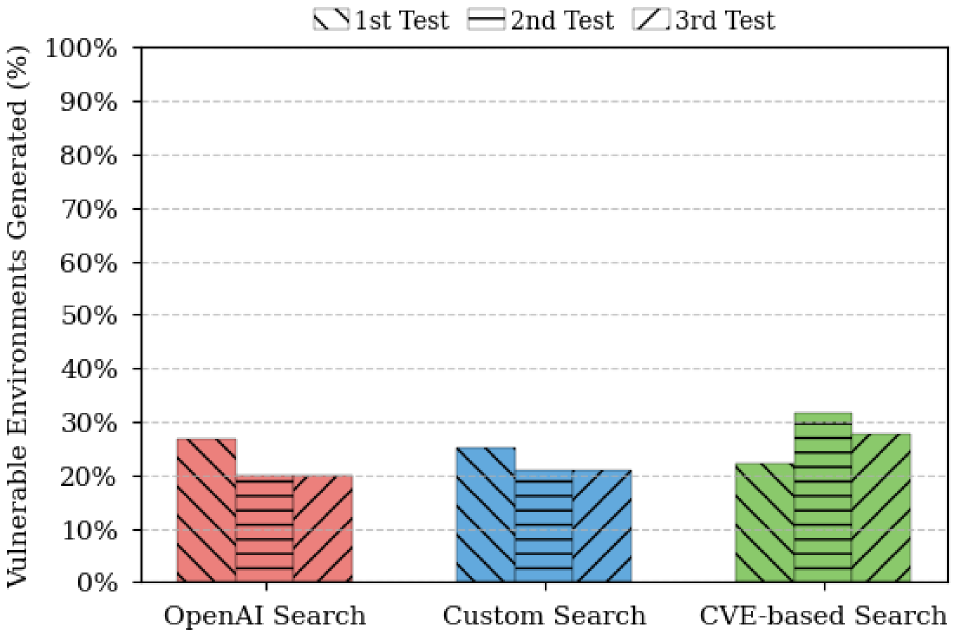


Figure 4.7: Vulnerability Assessment Task, `GPT-4o` Performance Consistency Across Strategies

Overall, Figure 4.7 shows that the **vulnerability stays low across all strategies and tests**,consistently falling within the **20–32%** range. Among the three strategies, **CVE-based Search** appears the **most promising**, as it achieves the highest vulnerability rate (**32%**) and **leads two out of three tests**. Moreover, as shown in Figure 4.6, **CVE-based Search** also has the highest success rate for the environment generation task in tests 2 and 3. Together, these findings indicate that **CVE-based Search** is not only good at generating virtual environments, but also yields a relatively large proportion of vulnerable ones, making **CVE-based Search** the **most effective strategy for the overall goal of the agent**.

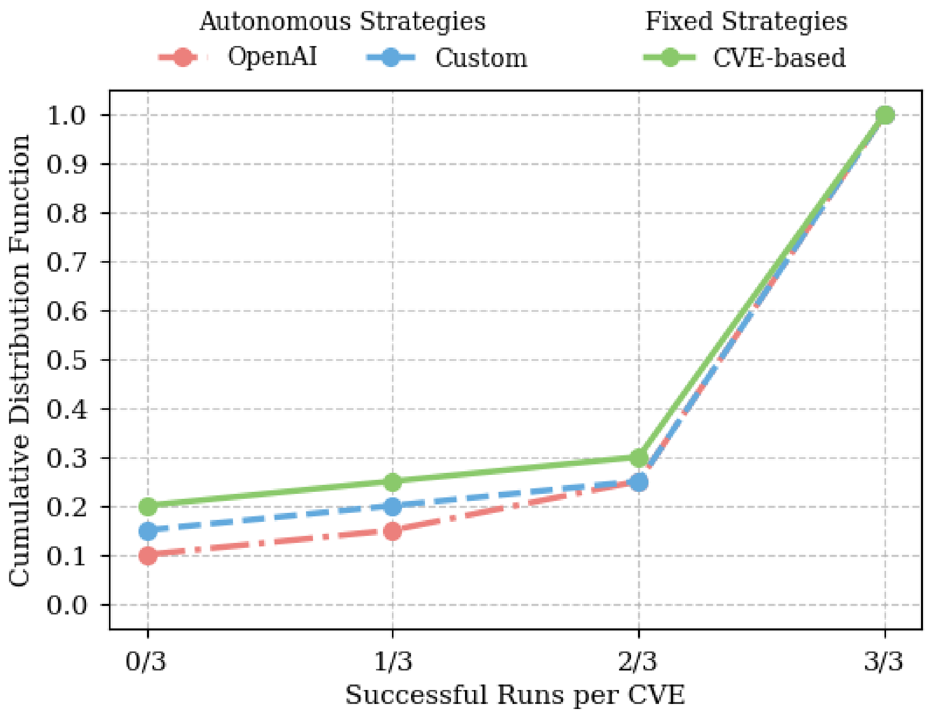


Figure 4.8: Vulnerability Assessment Task, `GPT-4o` Successful Runs per CVE, Cumulative Distribution Function Across Strategies

Grouping the results by CVE reveals that `GPT-4o` successfully created a functional vulnerable environment for **7 out of 20** CVEs of the Agent Development Dataset. Combining this knowledge with the data shown by Figure 4.8 points out that:

- when using **CVE-based Search**, `GPT-4o` generated vulnerable environments for **6 CVEs**. Two of these — the ones for CVE-2021-3129 and CVE-2021-44228 — were produced exclusively with this strategy.

- when using **Custom Search** and **OpenAI Search**, `GPT-4o` generated vulnerable environments for the **same 5 CVEs**. Only one of these — the one for CVE-2021-28164 — was not generated with **CVE-based Search**.

Test	Strategy	● **Inf. Gat.**	■ **Env. Gen.**	▲ **Vuln. Ass.**
1st Test	OpenAI	75%	73%	27%
	Custom	80%	75%	25%
	CVE-based	90%	61%	22%
2nd Test	OpenAI	75%	53%	20%
	Custom	95%	53%	21%
	CVE-based	**95%**	**74%**	**32%**
3rd Test	OpenAI	75%	53%	20%
	Custom	95%	42%	21%
	CVE-based	90%	72%	28%

Table 4.7: Agentic Workflow Tasks, `GPT-4o` Performance Consistency Across Strategies. Best results in **bold**

4.5 Ablation Study - Information Gathering

During the development of the agentic workflow it was decided that the agent would progress to the environment generation task **only if** all the milestones located in ● Assess CVE Services step were successfully validated, effectively applying an **early stop** to the run. This ablation study examines the potential benefits of **removing the early stop imposed by these milestones**, enabling the agent to attempt environment generation even with wrong or incomplete information about the input CVE. The primary motivation for this study was to **address evaluation errors** produced by the continuous use of LLM-as-a-Judge while validating the milestones. A second objective was to determine whether it is **necessary to validate the information collected by the agent** against a manually constructed dataset - which requires constant updates and dedication.

The ablation study repurposes the three experiments conducted with `GPT-4o` in Section 4.4 by **forcing** the agent to start the environment generation task for the runs that failed the information gathering task. Detailed logs of all runs are available in the Github repository of the project.[12]. Analysis of the actual benefits of the ablation study on the environment generation task are conducted by **considering all runs performed by the agents**, not only the ones that actually started. Looking at the conditional success rate of the agent would be **misleading** since the ablation study - by construction - allows the agent to start the environment generation task in any case.

[12]https://github.com/g1san/Agents-for-Vulnerable-Dockers-and-related-Benchmarks
See the `GPT-4o` 5th benchmark session (1st Test), `GPT-4o` 6th benchmark session (2nd Test), and `GPT-4o` 7th benchmark session (3rd Test)

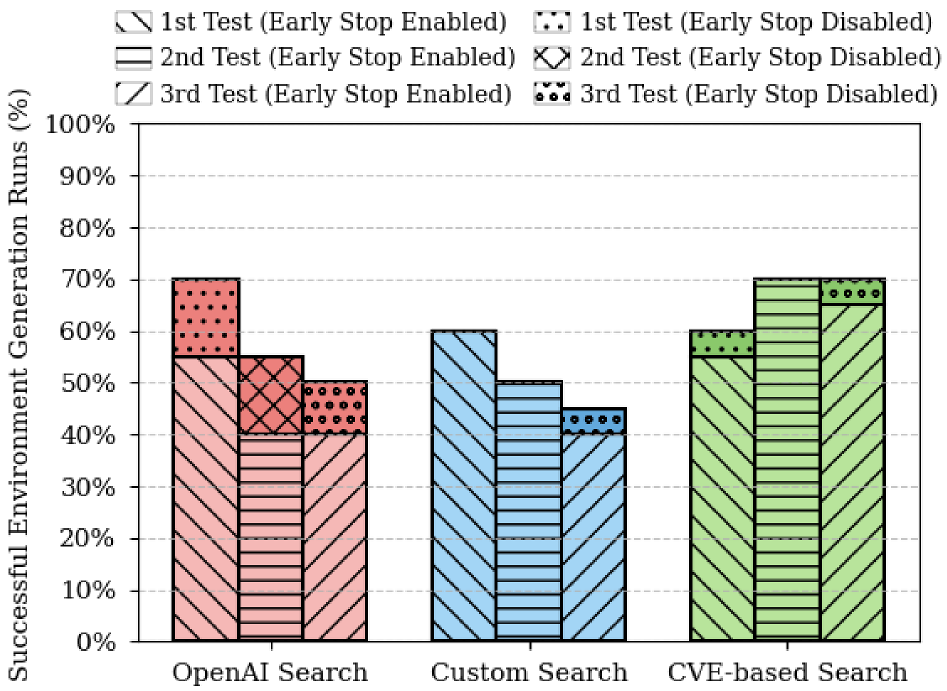


Figure 4.9: Environment Generation Task, `GPT-4o` Ablation Performance Variation Across Strategies. Performance Benefits of Disabling Early Stopping Use Brighter Colours

Data in Figure 4.9 shows that the **OpenAI Search** strategy benefits the most from removing the early stop, achieving a **10–15% performance gain** across all tests. This suggests that the strategy used by `GPT-4o` when operating with the `web-search-preview` tool does not reliably produce easily verifiable information, even when supported by `Structured Output` controls and an LLM-as-a-Judge. On the other hand, both the **Custom Search** and **CVE-based Search** strategies show only marginal improvements — **at most 5%** — once the early stop is removed. This indicates that the validation of the performed on the information gathered with these strategies is a **reliable indicator of good progress in the environment generation task**, and that the three identified goals[13] correctly predict the **knowledge requirements for successfully automating the environment generation task**.

[13]See the milestones associated to the ● Assess CVE Services step of the agentic workflow in Table 4.3 to understand which are these goals.

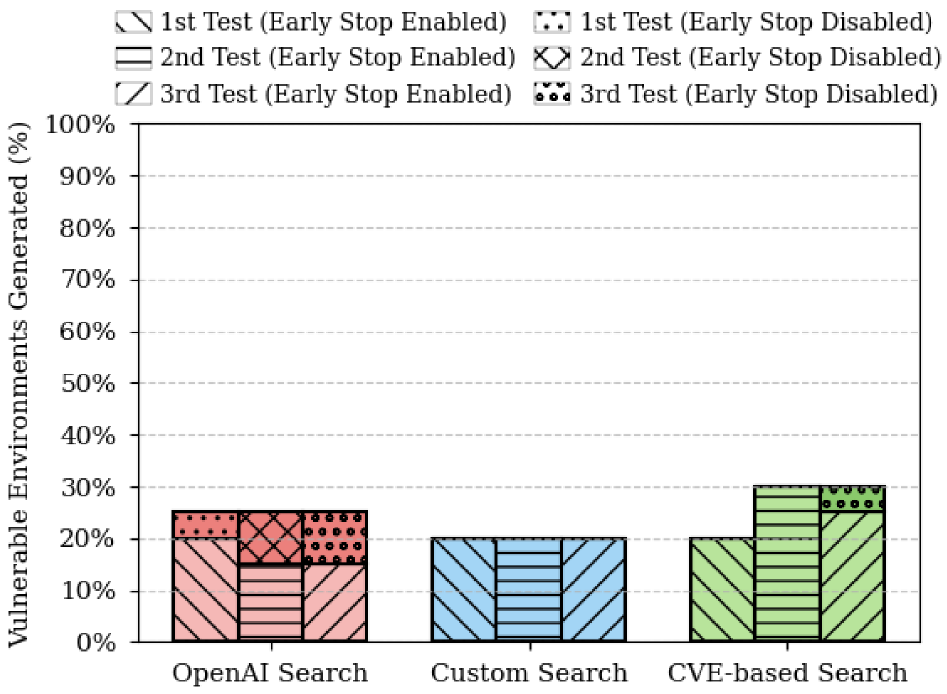


Figure 4.10: Vulnerability Assessment Task, `GPT-4o` Ablation Performance Variation Across Strategies. Performance Benefits of Disabling Early Stopping Use Brighter Colours

Figure 4.10 shows that **6 out of the 11** environments generated thanks to the ablation were **confirmed as vulnerable** by Docker Scout. Inspecting the corresponding `final_report.txt` files revealed that information validation always failed because the **right** `HARD`-type services were proposed with the ***wrong* service name**[14]. For example, all 4 successful runs associated to either CVE-2020-11651 or CVE-2020-11652 failed to progress to the environment generation task because the proposed `HARD`-type service was named `salt-master` instead of `salt`.

[14]Here "*wrong*" simply means different from the *expected* one

If we consider that only a single successful run is required to make a working vulnerable environment, then the ablation does not really **add value to the overall performance of the agent**:

- **1st Test**: before ablation, the agent successfully generated a working vulnerable environments for 5 CVEs. Ablation allowed the agent to successfully produce a working vulnerable environment for CVE-2020-11652, but this CVE was already among the original 5.
- **2nd Test**: before ablation, the agent successfully generated a working vulnerable environments for 6 CVEs. Ablation allowed the agent to successfully produce a working vulnerable environment for CVE-2020-11651 and CVE-2021-3129, but both were already included in the initial 6.
- **3rd Test**: before ablation, the agent successfully generated a working vulnerable environments for 6 CVEs. Ablation allowed the agent to successfully produce a working vulnerable environment for CVE-2020-11651, CVE-2020-11652 and CVE-2021-28164. However, just like for the other tests, all three were already part of the same original set of 6.

Nonetheless, the ablation study revealed that the agent can still construct a functional vulnerable environment **even without the *expected* data**. Thus, it was concluded that **removing the early stop from the workflow should become be a permanent change**. This eliminates the need to **manually maintain a dataset** for validating the information collected by the agent, though it may result in **greater resource usage** and **longer execution times** when the agent must build environments using **inaccurate or incomplete CVE data**.

`Test`	**Strategy**	🟪 **Env. Gen.**	▲ **Vuln. Ass.**
1st Test	**OpenAI**	55% → **70%**	20% → **25%**
	Custom	60% → 60%	20% → 20%
	CVE-based	55% → **60%**	20% → 20%
2nd Test	**OpenAI**	40% → **55%**	15% → **25%**
	Custom	50% → 50%	20% → 20%
	CVE-based	70% → 70%	30% → 30%
3rd Test	**OpenAI**	40% → **50%**	15% → **25%**
	Custom	40% → **45%**	20% → 20%
	CVE-based	65% → **70%**	25% → **30%**

Table 4.8: Agentic Workflow Tasks, `GPT-4o` Ablation Performance Variation (→) Across Strategies. Improvements in **bold**

4.6 Performance of the Agent in the Wild

The performance of the agent on CVEs that are not part of the Agent Development Dataset was measured by performing three separate runs for each CVE of the **Test Dataset**, which consists of 100 CVEs selected from those available on Vulhub [3]. The same experiment was performed with `GPT-4o` and `gpt-oss:120B`[15]. Following the consistency assessment results and the findings of the ablation study, all runs were **conducted exclusively with the CVE-based Search** strategy and **without early stop**.

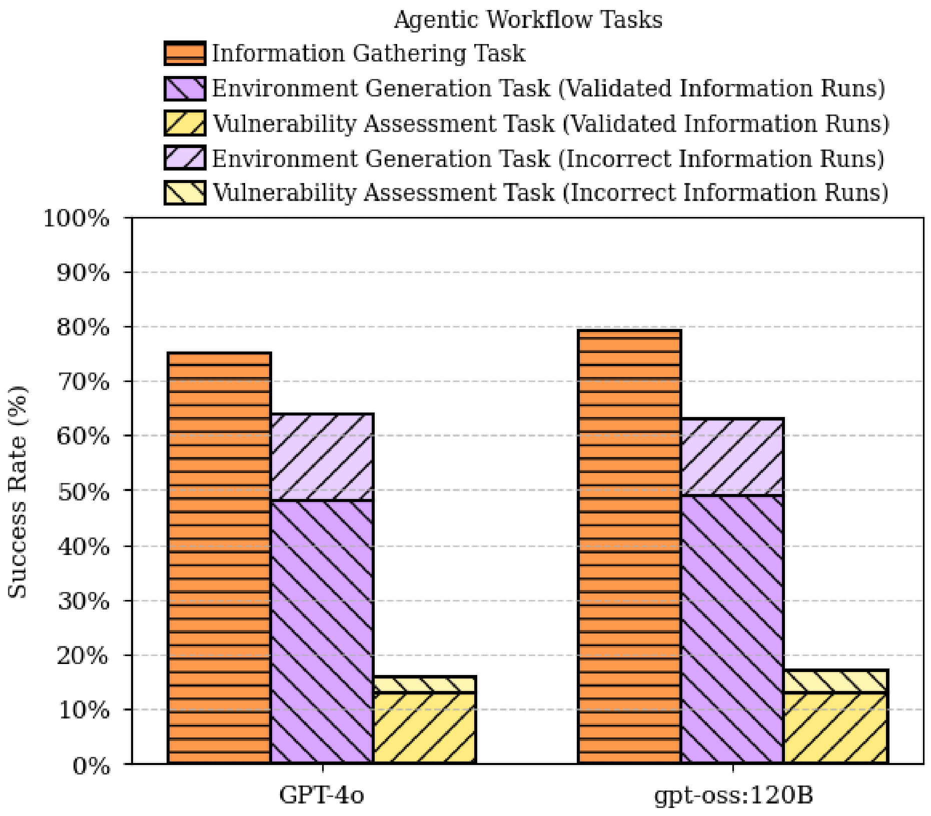


Figure 4.11: Agentic Workflow Tasks, Model Performance on the Test Dataset (Best Run Only). Data is Separated in Validated/Incorrect Information Runs Because the Agent Performs Information Validation Even With Early Stop Disabled

Figure 4.11 reports the **success rate of the models for each task**. For every task, the success rate is computed by aggregating all runs across the three experiments and selecting only the **best performing run for each CVE**.

[15]https://github.com/g1san/Agents-for-Vulnerable-Dockers-and-related-Benchmarks For `GPT-4o` and `gpt-oss:120B`, see the corresponding 1st, 2nd and 3rd test set results

- **Information Gathering Results Analysis**: data displayed in Figure 4.11 shows a slight advantage of `gpt-oss:120B` (**79%**) over `GPT-4o` (**75%**) in the information gathering task. When combining the results of both models to obtain a comprehensive view of agent performance, the information gathering task succeeds for **83%** of the CVEs. Of these, **4 CVEs** succeed only with `GPT-4o`, while **8** succeed only with `gpt-oss:120B`.

- **Environment Generation Results Analysis**: as expected, the success rate of this task is lower than the information gathering one, even accounting for ablation. `GPT-4o` achieves **64%** success rate (against **75%** in information gathering), and `gpt-oss:120B` achieves **63%** (against **79%**). Notably, **three quarters** of the successfully generated environment originate from runs that also passed the information gathering task, while the remaining quarter was succeeds thanks to ablation. Similarly, this **3 to 1 ratio** appears in the information gathering results: for `GPT-4o`, **75%** of the runs have valid information, the remaining quarter does not, while a comparable pattern manifests also for `gpt-oss:120B`. Combination of the best runs of both models reveals that the agent successfully generates a working environment for **78%** of the Test Dataset. **Fifteen CVEs** succeed only with `GPT-4o`, while **14** succeed only with `gpt-oss:120B`.

- **Vulnerability Assessment Results Analysis**: the trends above are **confirmed** by the vulnerability assessment results. For both models, roughly **one quarter** of the successfully generated environments are also confirmed to be vulnerable, and approximately **three quarters** of these vulnerable environments are generated using validated information. Specifically, considering all successfully generated environments, `GPT-4o` produces a vulnerable environment **25%** of the times, whereas `gpt-oss:120B` does so **27%** of the times. Finally, combination of the best runs of both models shows that the agent generates a functional and vulnerable environment for **22%** of the Test Dataset. Of these, 5 succeed only with `GPT-4o`, and 6 succeed only with `gpt-oss:120B`. If we consider only the CVEs for which the agent managed to generate a functional environment, for **28%** of them the environment was also confirmed to be vulnerable.

On **average**, when paired with `GPT-4o`, data collected from these last experiments shows that each run of the agent costed **0.20$** and took **8-9 minutes** to complete. Similar execution times are found when the agent is paired with `gpt-oss:120B`, while **costs are nullified** by the fact that the model is running locally. Overall, the two models exhibit very **similar performance** despite differences in size and reasoning ability. This reinforces the conclusion that **the performance of the agent is primarily constrained by its workflow**, not by the LLM itself.

However, it is also important to note that performance is measured by checking the milestones[16] of each run, and the model powering the agent also acts as an LLM-as-a-Judge in many of these validation steps. Evaluations made by the **LLM-as-a-Judge heavily influence milestone outcomes**. It has been observed that `GPT-4o` tends to apply stricter self-evaluation criteria — sometimes being **overly critical of their own work** — which can **lower success rates**. Conversely, `gpt-oss:120B` seems to apply more lenient judgments, increasing success rates but at the **cost of accepting incorrect information or flawed environments**, thus introducing **false negatives** into the results.

A fully **transparent performance assessment** of the agent would require **manual review of every run**. This is infeasible due to the **significant time required** and because it would undermine the purpose of the agent, which is to **automate environment generation and perform self-validation**.

4.7 Other Relevant Findings & Insights

Analysis of the logs related to Agent Development Dataset highlights CVE-2012-1823 as a good example of how the same problem can be **tackled in different ways** by the agent when paired with a different LLM.

- When paired with `GPT-4o`, the agent attempted to create a Docker environment vulnerable to CVE-2012-1823 [94] by pulling an outdated `php` image directly from Docker Hub. The vulnerability affects `php` versions prior to 5.3.12 and 5.4.2, and although corresponding images exist on Docker Hub (see here) attempts to pull them fail. This happens because some older images uses a **deprecated manifest format** - `version 1` or `version 2, schema 1` - that modern Docker versions can no longer load [95]. Unable to reason about this limitation, the agent repeatedly cycles through different vulnerable `php` versions, wasting its testing–revision loop iterations.
- In contrast, when paired with `GPT-5`, the agents adopts a completely different strategy. Rather than pulling a `php` image from Docker Hub, it generates a `Dockerfile` containing a **sequence of shell commands**. One of these commands downloads a vulnerable legacy `php` release directly from `museum.php.net`, **bypassing** the deprecated Docker Hub manifests entirely and enabling the successful creation of a functional vulnerable Docker environment.

[16]See Table 4.3 for an overview of the agentic workflow milestones

Chapter 5

Conclusion: Limitations & Future Works

- **Information Gathering Limitations**: the information available to the agent about the CVE is limited by the quality and reliability of the **knowledge base** of the LLM and by the data gathered from a few online sources. Increasing the number of analysed sources risks saturating the LLM context window and introducing useless information.
- **Context Window Saturation**: to mitigate the probability of saturating the context window of the LLM, some tests analyse just the last 100 lines of logs. This may **leave out important information** that may have otherwise helped the LLM determine what went wrong. Therefore, future work will evaluate increasing this arbitrary value to check if it can improve the ability of the LLM to correctly fix the detected issues.
- **Agent Memory**: there is only a **short-term memory system** - the `State` - which helps the agent keep track of what is going on **during a single run**. Future studies will implement a **long-term memory solution** and study how it can improve that agentic workflow by **learning from previous runs**.
- **Vulnerability Assessment**: the vulnerability assessment phase is performed purely at the **static** level using a third party tool - i.e., *Docker Scout*. Taking inspiration from what `CVE-Genie` [4] has done, future work will expand the current vulnerability assessment phase to enable the agent to **autonomously perform dynamic vulnerability assessment** by using a given exploit, or by trying to generate one.
- **Multi-agent Framework**: by exploiting the work of my colleagues which created other AI agents, future studies will also focus on integrating this agent into a multi-agent framework.

- **Number of Iterations**: all tests and results were obtained by **limiting the number of iterations the agent may perform in the testing–revision loop to 10**. This was done to regulate the total workflow execution time and reduce costs, but future studies will study the ability of the agent to generate virtual environments with a higher number of iterations, effectively testing its ability to learn from previous error.
- **Timeouts & System Limitations**: some of the functions that support the testing process are have a timeout:
  - The `launch_docker` function has an arbitrary timeout of **600 seconds** which was chosen to **accommodate the build process of a Docker Compose environment**. This large amount of time is necessary especially when images need to be **built from scratch** or **pulled from remote repositories**. Consequently, this value is strongly influenced by factors such as **image size** and **network speed**, and it may be changed depending on the performance of the system that is using the agent. Testing on my own system revealed that a **ten minutes timeout is a reasonable limit** that prevents the build process from **hanging indefinitely** in cases of misconfiguration or stalled containers.
  - The `run_docker_scout` function has an arbitrary timeout of **60 seconds** which was chosen to **accommodate the vulnerability assessment process of a Docker Image**. Unfortunately, testing on my own system revealed that this activity is **very memory intensive**, and proper automation can be achieved **only with a large amounts of RAM** (>16GB). The one minutes timeout is a **reasonable limit** that allows most Docker Images to get analysed without troubles. The main reason why this timeout is imposed is to prevent the scan process from **saturating the memory of the system and causing crashes**.
- **Vulnerability Reproduction Range**: testing results include only 100 CVEs against the over 300k publicly available CVE records [96]. This limited scope constrains the generalizability of the results, as the selected CVEs may not fully represent the diversity and complexity of real-world scenarios. Consequently, future work will also focus on widening the range of reproduced CVEs. The range of CVEs that can be reproduced in a virtual environment is limited by **virtualization technologies** used, as some services (or even just some versions) may not be virtualizable.
- **Manual Service Setup**: some services can require to be **manually setup and configured by a human user**, especially those involving web interfaces or similar. In some cases the agent is capable of coming up with a solution that bypasses the service setup phase.

Appendix A

Function Definitions

This appendix contains some relevant function code. Visit the Github page[1] of the project to see the latest version of these functions.

```python
def get_cve_from_nist_api(self, cve_id):
    url = f"https://services.nvd.nist.gov/rest/json/cves/2.0?cveId={cve_id}"
    try:
        response = requests.get(url, timeout=10)
        response.raise_for_status()
        data = response.json()
        vuln = data.get("vulnerabilities", [])[0]["cve"]
        description = vuln["descriptions"][0]["value"]
        return (url, description)
    except Exception as e:
        return f"Failed to retrieve CVE data from NIST: {str(e)}"
```

Listing A.1: Function used to retrieve CVE data from the NVD repository using the official API [91]

[1]https://github.com/g1san/Agents-for-Vulnerable-Dockers-and-related-Benchmarks.git

```python
def launch_docker(code_dir_path, log_file):
    try:
        result = subprocess.run(
            ["sudo", "docker", "compose", "up", "--build", "--detach"],
            cwd=code_dir_path,
            stdout=subprocess.PIPE,
            stderr=subprocess.STDOUT,
            text=True,
            timeout=600)
        logs = result.stdout
        success = (result.returncode == 0)
        with builtins.open(log_file, "w") as f:
            f.write(logs)
        return success, logs
    except subprocess.TimeoutExpired as e:
        print(f"\t{e}")
        return False, e.output if e.output is not None else "No logs available"
```

Listing A.2: Function used to launch a Docker starting from a `docker-compose.yml` file

```python
def get_image_ids():
    result = subprocess.run(
        ["sudo", "docker", "images", "-q"],
        stdout=subprocess.PIPE,
        stderr=subprocess.DEVNULL,
        text=True)
    return result.stdout.splitlines()
```

Listing A.3: Function used to get a list of the identifiers of all Docker Images currently running in the system

```python
def get_container_ids(code_dir_path):
    result = subprocess.run(
        ["sudo", "docker", "compose", "ps", "-a", "--quiet"],
        cwd=code_dir_path,
        capture_output=True,
        text=True)
    return result.stdout.strip().splitlines()
```

Listing A.4: Function used to get a list of all Docker Containers currently running in the system

```python
def get_container_logs(cid, log_file):
    result = subprocess.run(
        ["sudo", "docker", "logs", cid, "--details"],
        capture_output=True,
        text=True)
    log = f"\n\nsudo docker logs {cid} --details\nSTDOUT: {result.stdout}\
    nSTDERR: {result.stderr}\n\n"
    with builtins.open(log_file, "a") as f:
        f.write(log)
    log = f"\n\nsudo docker logs {cid} --details\nSTDOUT: {result.stdout.
    splitlines()[-100:]}\nSTDERR: {result.stderr.splitlines()[-100:]}\n\n"
    return log
```

Listing A.5: Function used to get the output logs of a single Docker Container

```python
def inspect_image(iid, log_file):
    result = subprocess.run(
        ["sudo", "docker", "inspect", iid],
        capture_output=True,
        text=True)
    try:
        log = json.loads(result.stdout)
        with builtins.open(log_file, "a") as f:
            f.write(f"\n\nsudo docker inspect {iid}")
            json.dump(log, f, indent=4)
    except json.JSONDecodeError:
        raise ValueError(f"Failed to parse JSON for container {iid}")
    return log[0]
```

Listing A.6: Function used to get detailed information about a single Docker Image

```python
def inspect_container(cid, log_file):
    result = subprocess.run(
        ["sudo", "docker", "inspect", cid],
        capture_output=True,
        text=True)
    try:
        log = json.loads(result.stdout)
        with builtins.open(log_file, "a") as f:
            f.write(f"\n\nsudo docker inspect {cid}")
            json.dump(log, f, indent=4)
    except json.JSONDecodeError:
        raise ValueError(f"Failed to parse JSON for container {cid}")
    return log[0]
```

Listing A.7: Function used to get detailed information about a single Docker Container

```python
def down_docker(code_dir_path):
    subprocess.run(
        ["sudo", "docker", "compose", "down", "--volumes"],
        cwd=code_dir_path,
        stdout=subprocess.DEVNULL,
        stderr=subprocess.DEVNULL)
    time.sleep(10)
```

Listing A.8: Function used to stop a currently running Docker Container

```python
def remove_all_images():
    image_ids = subprocess.check_output(["docker", "images", "-aq"]).decode().
    split()
    if image_ids:
        subprocess.run(
            ["docker", "rmi", "-f"] + image_ids,
            stdout=subprocess.DEVNULL,
            stderr=subprocess.DEVNULL,
            check=True)
```

Listing A.9: Function used to remove all Docker Images currently stored in the system

```python
def run_docker_scout(code_dir_path, index, iid):
    try:
        with builtins.open(f"{code_dir_path}/logs/cves{index}.json", "w") as f:
            subprocess.run(
                ["docker", "scout", "cves", iid, "--format", "gitlab"],
                cwd=code_dir_path,
                stdout=f,
                stderr=subprocess.DEVNULL,
                text=True,
                timeout=60)
            print(f"\tCVE List file saved to: {code_dir_path}/logs/cves{index}.
    json")
            return True
    except subprocess.TimeoutExpired:
        print(f"\tDocker Scout timed out after 60 seconds for image {iid}")
        return False
```

Listing A.10: Function used to perform a vulnerability scan of a single Docker Image using Docker Scout

Bibliography

[1] SecurityScorecard. *CVEdetails*. URL: `https://www.cvedetails.com/browse-by-date.php` (cit. on p. 1).

[2] Mu et. al. «Understanding the Reproducibility of Crowd-reported Security Vulnerabilities». In: *27th USENIX Security Symposium (USENIX Security 18)* (Aug. 2018), pp. 919–936 (cit. on p. 2).

[3] Owen Gong. *Vulhub*. URL: `https://vulhub.org/` (cit. on pp. 2, 3, 15, 16, 25, 27, 47, 84).

[4] Ullah et al. «From CVE Entries to Verifiable Exploits: An Automated Multi-Agent Framework for Reproducing CVEs». In: *arXiv* (2025). URL: `https://arxiv.org/abs/2509.01835` (cit. on pp. 2, 15, 19, 87).

[5] MITRE. *CVE Services API*. URL: `https://cveawg.mitre.org/api-docs/` (cit. on pp. 3, 39).

[6] LangChain. *LangChain Homepage*. URL: `https://www.langchain.com/` (cit. on pp. 3, 31).

[7] Langfuse. *Langfuse Homepage*. URL: `https://langfuse.com/` (cit. on p. 3).

[8] J. McCarthy et al. «A Proposal For The Dartmouth Summer Research Project on Artficial Intelligence». In: *Standford University* (1955). URL: `http://jmc.stanford.edu/articles/dartmouth/dartmouth.pdf` (cit. on p. 5).

[9] F. Rosenblatt. «The Perceptron: A Propbabilistic Model For Information Storage And Organization In The Brain». In: *Cornell Aeronautical Laboratory* (1959). URL: `https://www.ling.upenn.edu/courses/cogs501/Rosenblatt1958.pdf` (cit. on p. 5).

[10] A. Samuel. «Some Studies in Machine Learning Using the Game of Checkers». In: *IBM Journal* (1959). URL: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5392560` (cit. on p. 5).

[11] E. Rumelhart et. al. «Learning Representations by Back-propagating Errors». In: *Nature* (1986). URL: `https://www.nature.com/articles/323533a0` (cit. on p. 6).

[12] E. Hinton. «Learning Multiple Layers of Representation». In: *Science Direct* (2007). URL: https://www.cs.toronto.edu/~hinton/absps/tics.pdf (cit. on p. 6).

[13] Vaswani et al. «Attention Is All You Need». In: *arXiv* (2017). URL: https://arxiv.org/abs/1706.03762 (cit. on p. 7).

[14] Brown et al. «Language Models are Few-Shot Learners». In: *arXiv* (2020). URL: https://arxiv.org/abs/2005.14165 (cit. on pp. 7, 8).

[15] Wei et al. «Finetuned Language Models Are Zero-Shot Learners». In: *arXiv* (2022). URL: https://arxiv.org/abs/2109.01652 (cit. on p. 9).

[16] Ouyang et al. «Training language models to follow instructions with human feedback». In: *arXiv* (2023). URL: https://arxiv.org/abs/2203.02155 (cit. on p. 9).

[17] Zheng et al. «Lost in the Middle: How Language Models Use Long Contexts». In: *arXiv* (2023). URL: https://arxiv.org/abs/2307.03172 (cit. on pp. 9, 10).

[18] Huang et al. «TrustLLM: Trustworthiness in Large Language Models». In: *arXiv* (2024). URL: https://arxiv.org/abs/arxiv:2401.05561 (cit. on p. 10).

[19] Liu et al. «Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena». In: *arXiv* (2023). URL: https://arxiv.org/abs/2306.05685 (cit. on p. 10).

[20] Politecinco di Torino. *AI and Cybersecurity Course Page, A.a. 2024/25*. URL: https://didattica.polito.it/pls/portal30/gap.pkg_guide.viewGap?p_cod_ins=01GYZWQ&p_a_acc=2025&p_header=S&p_lang=IT&multi=N (cit. on p. 11).

[21] Wei et al. «Chain-of-Thought Prompting Elicits Reasoning in Large Language Models». In: *arXiv* (2023). URL: https://arxiv.org/abs/2201.11903 (cit. on p. 12).

[22] Rapid7. *Metasploit Homepage*. URL: https://www.metasploit.com/ (cit. on p. 15).

[23] Cuckoo Foundation. *Cuckoo Sandbox Homepage*. URL: https://cuckoo.readthedocs.io/en/latest/ (cit. on pp. 15, 16).

[24] Open Worldwide Application Security Project (OWASP). *Source Code Analysis Tools*. URL: https://owasp.org/www-community/Source_Code_Analysis_Tools (cit. on p. 17).

[25] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. «Modeling and Discovering Vulnerabilities with Code Property Graphs». In: *Proc. IEEE Symposium on Security and Privacy (SP)*. San Jose, CA, USA, May 2014, pp. 590–604. ISBN: 978-1-4799-4686-0. DOI: `10.1109/SP.2014.44`. URL: `https://ieeexplore.ieee.org/document/6956589` (cit. on p. 17).

[26] Zhou et al. «Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks». In: *arXiv* (2019). URL: `https://arxiv.org/abs/1909.03496` (cit. on p. 17).

[27] D. Hin, A. Kan, H. Chen, and M. A. Babar. «LineVD: Statement-level Vulnerability Detection using Graph Neural Networks». In: *Proc. 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. Pittsburgh, PA, USA: ACM, May 2022, pp. 596–607. ISBN: 978-1-4503-9303-4. DOI: `10.1145/3524842.3527949`. URL: `https://dl.acm.org/doi/10.1145/3524842.3527949` (cit. on p. 17).

[28] Y. Mirsky, G. Macon, M. Brown, C. Yagemann, M. Pruett, E. Downing, S. Mertoguno, and W. Lee. «VulChecker: Graph-based Vulnerability Localization in Source Code». In: *Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA, USA: USENIX Association, Aug. 2023, pp. 6557–6574. ISBN: 978-1-939133-37-3. URL: `https://www.usenix.org/conference/usenixsecurity23/presentation/mirsky` (cit. on p. 17).

[29] G. Lu, X. Ju, X. Chen, W. Pei, and Z. Cai. «GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning». In: *Journal of Systems and Software* 212 (Mar. 2024), p. 112031. ISSN: 0164-1212. DOI: `10.1016/j.jss.2024.112031`. URL: `https://www.sciencedirect.com/science/article/pii/S0164121224000748` (cit. on p. 17).

[30] Y. Guo, C. Patsakis, Q. Hu, Q. Tang, and F. Casino. «Outside the Comfort Zone: Analysing LLM Capabilities in Software Vulnerability Detection». In: *Proc. Computer Security – ESORICS 2024*. Ed. by Joaquín García-Alfaro, Rafał Kozik, Michał Choraś, and Sokratis K. Katsikas. Vol. 14982. Lecture Notes in Computer Science. Bydgoszcz, Poland: Springer, Cham, Sept. 2024, pp. 271–289. ISBN: 978-3-031-70878-7. DOI: `10.1007/978-3-031-70879-4_14`. URL: `https://link.springer.com/chapter/10.1007/978-3-031-70879-4_14` (cit. on p. 17).

[31] Wang et al. «VulAgent: Hypothesis-Validation based Multi-Agent Vulnerability Detection». In: *arXiv* (2025). URL: `https://arxiv.org/abs/2509.11523` (cit. on p. 17).

[32] Rafael Ramires, Ana Respício, and Ibéria Medeiros. «KAVE: A Knowledge-Based Multi-Agent System for Web Vulnerability Detection». In: *Proc. 2024 IEEE International Conference on Web Services (ICWS)*. Shenzhen, China, July 2024, pp. 489–500. ISBN: 979-8-3503-6855-0. DOI: `10.1109/ICWS62655.2024.00070`. URL: `https://ieeexplore.ieee.org/abstract/document/10707503` (cit. on p. 17).

[33] A. Rosay, F. Carlier, Eloïse Cheval, and P. Leroux. «From CIC-IDS2017 to LYCOS-IDS2017: A corrected dataset for better performance». In: *Proc. IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT '21)*. Melbourne, Australia: Association for Computing Machinery, Dec. 2021, pp. 570–575. DOI: `10.1145/3486622.3493973`. URL: `https://dl.acm.org/doi/10.1145/3486622.3493973` (cit. on p. 17).

[34] G. Engelen, V. Rimmer, and W. Joosen. «Troubleshooting an Intrusion Detection Dataset: the CICIDS2017 Case Study». In: *Proc. 2021 IEEE Security and Privacy Workshops (SPW)*. San Francisco, CA, USA, May 2021, pp. 7–12. ISBN: 978-1-7281-8934-5. DOI: `10.1109/SPW53761.2021.00009`. URL: `https://ieeexplore.ieee.org/abstract/document/9474286` (cit. on p. 17).

[35] A. Boukhamla and J. Coronel Gaviro. «CICIDS2017 dataset: performance improvements and validation as a robust intrusion detection system testbed». In: *Int. J. Inf. Comput. Secur.* 16.1/2 (2021), pp. 20–32. DOI: `10.1504/IJICS.2021.117392`. URL: `https://www.inderscienceonline.com/doi/abs/10.1504/IJICS.2021.117392` (cit. on p. 17).

[36] Ullah et al. «LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks». In: *arXiv* (2023). URL: `https://arxiv.org/abs/2312.12575` (cit. on p. 17).

[37] Risse et al. «Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection». In: *arXiv* (2024). URL: `https://arxiv.org/abs/arxiv:2408.12986` (cit. on p. 17).

[38] Chen et al. «DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection». In: *arXiv* (2023). URL: `https://arxiv.org/abs/2304.00409` (cit. on p. 17).

[39] J. Fan, Y. Li, S. Wang, and T. N. Nguyen. «A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries». In: *Proc. IEEE/ACM Int. Conf. Mining Software Repositories (MSR '20)*. Virtual, Online (Korea, Republic of): Association for Computing Machinery, June 2020, pp. 508–

512. ISBN: 978-1-4503-7957-1. DOI: 10.1145/3379597.3387501. URL: https://dl.acm.org/doi/10.1145/3379597.3387501 (cit. on p. 17).

[40] PyCQA. *Bandit.* URL: https://github.com/PyCQA/bandit (cit. on p. 17).

[41] Meta. *Infer Homepage.* URL: https://fbinfer.com/ (cit. on p. 17).

[42] Jie Zhang et al. «When LLMs meet cybersecurity: a systematic literature review». In: *Cybersecurity* 8 (Feb. 2025), p. 55. DOI: 10.1186/s42400-025-00361-w. URL: https://doi.org/10.1186/s42400-025-00361-w (cit. on p. 18).

[43] Mitra et al. «LOCALINTEL: Generating Organizational Threat Intelligence from Global and Local Cyber Knowledge». In: *arXiv* (2024). URL: https://arxiv.org/abs/arxiv:2401.05561 (cit. on p. 18).

[44] Schwartz et al. «LLMCloudHunter: Harnessing LLMs for Automated Extraction of Detection Rules from Cloud-Based CTI». In: *arXiv* (2024). URL: https://arxiv.org/abs/arxiv:2407.05194 (cit. on p. 18).

[45] Siracusano et al. «Time for aCTIon: Automated Analysis of Cyber Threat Intelligence in the Wild». In: *arXiv* (2023). URL: https://arxiv.org/abs/2307.10214 (cit. on p. 18).

[46] Y. Hu, F. Zou, J. Han, X. Sun, and Y. Wang. «LLM-TIKG: Threat intelligence knowledge graph construction utilizing large language model». In: *Computers and Security* 145 (July 2024), p. 103999. DOI: 10.1016/j.cose.2024.103999. URL: https://doi.org/10.1016/j.cose.2024.103999 (cit. on p. 18).

[47] Wu et al. «KGV: Integrating Large Language Models with Knowledge Graphs for Cyber Threat Intelligence Credibility Assessment». In: *arXiv* (2024). URL: https://arxiv.org/abs/arxiv:2408.08088 (cit. on p. 18).

[48] Zhang et al. «CUPID: Leveraging ChatGPT for More Accurate Duplicate Bug Report Detection». In: *arXiv* (2023). URL: https://arxiv.org/abs/2308.10022 (cit. on p. 18).

[49] Tseng et al. «Using LLMs to Automate Threat Intelligence Analysis Workflows in Security Operation Centers». In: *arXiv* (2024). URL: https://arxiv.org/abs/arxiv:2407.13093 (cit. on p. 18).

[50] Zhou et al. «Comparison of Static Application Security Testing Tools and Large Language Models for Repo-level Vulnerability Detection». In: *arXiv* (2024). URL: https://arxiv.org/abs/arxiv:2407.16235 (cit. on p. 18).

[51] Mao et al. «Towards Explainable Vulnerability Detection with Large Language Models». In: *arXiv* (2024). URL: https://arxiv.org/abs/arxiv:2406.09701 (cit. on p. 18).

[52] C. Zhang, H. Liu, J. Zeng, K. Yang, Y. Li, and H. Li. «Prompt-Enhanced Software Vulnerability Detection Using ChatGPT». In: *Proc. IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24)*. Lisbon, Portugal, Apr. 2024, pp. 276–277. DOI: 10.1145/3639478.3643065. URL: https://doi.org/10.1145/3639478.3643065 (cit. on p. 18).

[53] Mathews et al. «LLbezpeky: Leveraging Large Language Models for Vulnerability Detection». In: *arXiv* (2024). URL: https://arxiv.org/abs/arxiv:2401.01269 (cit. on p. 18).

[54] Pearce et al. «Pop Quiz! Can a Large Language Model Help With Reverse Engineering?» In: *arXiv* (2022). URL: https://arxiv.org/abs/2202.01142 (cit. on p. 18).

[55] Runchu Tian et al. «DebugBench: Evaluating Debugging Capability of Large Language Models». In: *Findings of the Association for Computational Linguistics: ACL 2024*. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 4173–4198. DOI: 10.18653/v1/2024.findings-acl.247. URL: https://aclanthology.org/2024.findings-acl.247/ (cit. on p. 18).

[56] Jiaxing Qi et al. «LogGPT: Exploring ChatGPT for Log-Based Anomaly Detection». In: *Proc. IEEE Int. Conf. on High Performance Computing & Communications; Data Science & Systems; Smart City; and Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. Melbourne, Australia, Dec. 2023, pp. 273–280. DOI: 10.1109/HPCC-DSS-SMARTCITY-DEPENDSYS60770.2023.00045. URL: https://doi.org/10.1109/HPCC-DSS-SMARTCITY-DEPENDSYS60770.2023.00045 (cit. on p. 18).

[57] Jamal et al. «An Improved Transformer-based Model for Detecting Phishing, Spam, and Ham: A Large Language Model Approach». In: *arXiv* (2023). URL: https://arxiv.org/abs/2311.04913 (cit. on p. 18).

[58] M. Scanlon, F. Breitinger, C. Hargreaves, J.-N. Hilgert, and J. Sheppard. «ChatGPT for Digital Forensic Investigation: The Good, the Bad, and the Unknown». In: *Forensic Sci. Int.: Digit. Investig.* 46 (2023). DOI: 10.1016/j.fsidi.2023.301609. URL: https://www.sciencedirect.com/science/article/pii/S266628172300121X (cit. on p. 18).

[59] Fumero et al. «CyberSleuth: Autonomous Blue-Team LLM Agent for Web Attack Forensics». In: *arXiv* (2025). URL: https://arxiv.org/abs/2508.20643 (cit. on p. 18).

[60] Jiang et al. «When Fuzzing Meets LLMs: Challenges and Opportunities». In: *arXiv* (2024). URL: https://arxiv.org/abs/arxiv:2404.16297 (cit. on p. 18).

[61] Deng et al. «PentestGPT: An LLM-empowered Automatic Penetration Testing Tool». In: *arXiv* (2023). URL: `https://arxiv.org/abs/2308.06782` (cit. on p. 18).

[62] Gioacchini et al. «AutoPenBench: Benchmarking Generative Agents for Penetration Testing». In: *arXiv* (2024). URL: `https://arxiv.org/abs/arxiv:2410.03225` (cit. on p. 18).

[63] Potter et al. «Frontier AI's Impact on the Cybersecurity Landscape». In: *arXiv* (2025). URL: `https://arxiv.org/abs/2504.05408` (cit. on p. 19).

[64] National Institute of Standards and Technology (NIST). *National Vulnerability Database (NVD)*. URL: `https://nvd.nist.gov/` (cit. on pp. 25, 39, 44, 47, 67).

[65] Inc. Docker. *Docker Scout Documentation*. URL: `https://docs.docker.com/scout/` (cit. on pp. 28, 32, 63, 73).

[66] OpenAI. *OpenAI Agent SDK Documentation*. URL: `https://github.com/openai/openai-agents-python` (cit. on p. 29).

[67] Microsoft Research. *AutoGen Documentation*. URL: `https://www.microsoft.com/en-us/research/project/autogen/` (cit. on p. 30).

[68] CrewAI. *CrewAI Homepage*. URL: `https://www.crewai.com/` (cit. on p. 30).

[69] CrewAI. *CrewAI Documentation*. URL: `https://docs.crewai.com/en/introduction` (cit. on p. 31).

[70] LangGraph. *LangGraph Homepage*. URL: `https://www.langchain.com/langgraph` (cit. on p. 31).

[71] Owen Gong (aka 'phith0n' on Github). *Vulhub*. URL: `https://github.com/vulhub/vulhub` (cit. on pp. 31, 32).

[72] Docker Inc. *Docker Documentation*. URL: `https://docs.docker.com/get-started/docker-overview/` (cit. on p. 31).

[73] Docker Inc. *Docker Swarm Documentation*. URL: `https://docs.docker.com/engine/swarm/` (cit. on p. 32).

[74] Cloud Native Computing Foundation (CNCF) Google. *Kubernetes Documentation*. URL: `https://kubernetes.io/` (cit. on p. 32).

[75] Docker Inc. *Docker Desktop Documentation*. URL: `https://docs.docker.com/desktop/` (cit. on p. 32).

[76] Inc. Docker. *Docker Desktop Windows Documentation*. URL: `https://docs.docker.com/desktop/setup/install/windows-install/` (cit. on p. 32).

[77] Microsoft. *Windows Containers Documentation*. URL: `https://learn.microsoft.com/en-us/virtualization/windowscontainers/?WT.mc_id=AZ-MVP-5003649` (cit. on p. 32).

[78] Pydantic. *Pydantic Homepage*. URL: `https://docs.pydantic.dev/latest/` (cit. on p. 33).

[79] LangChain. *LangChain Structured Outputs*. URL: `https://python.langchain.com/docs/concepts/structured_outputs/` (cit. on p. 33).

[80] LangChain. *LangChain Pydantic Output Parser*. URL: `https://python.langchain.com/api_reference/core/output_parsers/langchain_core.output_parsers.pydantic.PydanticOutputParser.html` (cit. on p. 34).

[81] LangChain. *LangChain Messages*. URL: `https://python.langchain.com/docs/concepts/messages/` (cit. on p. 34).

[82] European Union Agency for Cybersecurity (ENISA). *European Union Vulnerability Database (EUVD)*. URL: `https://euvd.enisa.europa.eu/` (cit. on p. 39).

[83] China Information Technology Security Evaluation Center (CNITSEC). *China National Vulnerability Database (CNNVD)*. URL: `https://www.cnnvd.org.cn/` (cit. on p. 39).

[84] F. Xiaodun and M. De. *WooYun*. URL: `https://wy.zone.ci/` (cit. on p. 39).

[85] European Union Agency for Cybersecurity (ENISA). *CVE-2021-0877*. URL: `https://euvd.enisa.europa.eu/vulnerability/EUVD-2021-0877` (cit. on p. 39).

[86] National Institute of Standards and Technology (NIST). *CVE-2021-28164*. URL: `https://nvd.nist.gov/vuln/detail/cve-2021-28164` (cit. on pp. 39, 76).

[87] European Union Agency for Cybersecurity (ENISA). *CVE-2021-28164*. URL: `https://euvd.enisa.europa.eu/vulnerability/EUVD-2021-28164` (cit. on p. 39).

[88] OpenAI. *Web Search Built-in Tool*. URL: `https://platform.openai.com/docs/guides/tools-web-search?api-mode=responsess` (cit. on p. 40).

[89] Google. *Custom Search JSON API*. URL: `https://developers.google.com/custom-search/v1/overview` (cit. on p. 43).

[90] Leonard Richardson. *Beautiful Soup Documentation*. URL: `https://www.crummy.com/software/BeautifulSoup/bs4/doc/` (cit. on p. 43).

[91] National Institute of Standards and Technology (NIST). *National Vulnerability Database (NVD) - CVE API*. URL: `https://nvd.nist.gov/developers/vulnerabilities` (cit. on pp. 44, 90).

[92] Python. *Python `subprocess` library*. URL: `https://docs.python.org/3/library/subprocess.html` (cit. on p. 53).

[93] OpenAI. *OpenAI Pricing Documentation*. URL: `https://platform.openai.com/docs/pricing` (cit. on p. 66).

[94] National Institute of Standards and Technology (NIST). *CVE-2012-1823*. URL: `https://nvd.nist.gov/vuln/detail/cve-2012-1823` (cit. on p. 86).

[95] Docker Inc. *Docker Deprecated Manifest Documentation*. URL: `https://docker-docs.uclv.cu/registry/spec/deprecated-schema-v1/` (cit. on p. 86).

[96] MITRE. *Common Vulnerabilities and Enumeration (CVE)*. URL: `https://www.cve.org/` (cit. on p. 88).