# LLM agents for vulnerability identification and verification of CVEs

Tadesse ZeMicheal[1,*], Hsin Chen[1], Shawn Davis[1], Rachel Allen[1], Michael Demoret[1] and Ashley Song[1]

[1]NVIDIA

## Abstract

Vulnerability management in containerized systems is a labor-intensive and time-consuming process, particularly when dealing with many containers. This process involves the collection, comprehension, and synthesis of various pieces of information to ascertain whether immediate remediation is necessary upon the identification of a new common vulnerability and exposure (CVE). If analysts conclude remediation is not required, they assign an exemption justification status category from the standardized Vulnerability Exploitability eXchange (VEX) reasoning. This is a manual and time-consuming task. To address this issue, we propose a multi-component system using Large Language Models (LLM) that automates vulnerability management, verification, and VEX justification. The system uses a Plan-and- Execute-style LLM system for vulnerability impact analysis. The process begins with an LLM planner that generates a context-sensitive task checklist with up-to-date CVE intel. This checklist is then executed by an LLM agent equipped with Retrieval-Augmented Generation (RAG) capabilities and tool usage. The gathered information and the agent's findings are subsequently summarized and categorized by additional LLMs to provide a final verdict. The system eliminates the need for manual verification of CVEs in target containers by leveraging container Software Bill of Materials (SBOM), source code, and documentation as input. Experimental results on both synthetic and real-world datasets demonstrate that the proposed system achieves high accuracy rates in capturing false-triggered CVEs, and final justification summary in par with human labeled justifications, indicates the effectiveness of the approach in streamlining vulnerability analysis tasks. We release our code and blueprint reference at github.com/NVIDIA-AI-Blueprints/vulnerability-analysis

## Keywords

Vulnerability assessment, LLM, LLM agent

## 1. Introduction

Modern enterprise applications have complex software dependencies, forming an interconnected web that provides unprecedented functionality, but with the cost of exponentially increasing complexity. Patching software security issues is becoming progressively more challenging as the number of reported security flaws in the common vulnerabilities and exposures (CVE) database hit a record high in 2022, according to the CVE database [1]. The National Vulnerability Database (NVD) reported 17% yearover-year increase in vulnerabilities, with over two hundred thousand cumulative vulnerabilities reported as of the end of 2023 [2]. It is clear that a traditional approach to scanning and patching has become unmanageable. Large Language Models can improve vulnerability remediation while decreasing the load on security teams. While some organizations have begun to explore generative AI to help automate this process, doing so at enterprise scale requires the collection, comprehension, and synthesis of many pieces of information.

In recent years, LLM agents have gained attention, due to the capability of performing complex tasks autonomously. For example, tool assisted LLM agents are now capable of performing complex software engineering tasks, such as user interface design, code generation, test executor [3, 4] and even assisting in scientific investigations [5, 6]. A crucial factor enabling these advanced capabilities is the ability to utilize tools. LLM agents exhibit a wide range of capabilities in terms of tool usage and feedback response. In the domain of cybersecurity, LLM usage for various cybersecurity applications has been a new trend [7].

CAMLIS'24: Conference on Applied Machine Learning for Information Security, October 24–25, 2024, Arlington, VA

[*]Corresponding author.

tzemicheal@nvidia.com (T. ZeMicheal)

The NVD defines vulnerabilities as a “weakness in the computational logic (e.g., code) found in software and hardware components that, when exploited, results in a negative impact to confidentiality, integrity, or availability” [8]. Vulnerability, characterized by a weakness in security system, implies that a hypothetical attacker could potentially leverage a misconfiguration to escalate privileges. In contrast, exploitability denotes the existence of a specific attack vector that can be utilized to gain unauthorized access to sensitive information. Although a vulnerability may be theoretically exploitable, it does not necessarily imply the presence of a feasible exploitation path, which is a critical distinction in vulnerability assessment and risk management. It is crucial for analysts to verify the exploitability nature of flagged vulnerabilities from CVEs. For example, these could include whether the vulnerable packages are updated to the latest patch or upgraded to recommended versions.

This work introduces techniques and tools for automating software vulnerability verification and identification on large scale containers via LLM agent system. Our focus lies in leveraging all available input artifacts used for creating Docker containers to address CVE reported by vulnerability scanners. These input artifacts include Software Bill of Material (SBOM), source code, and developer documentation. In these settings, a container scanner, such as Anchore scanner[9], performs a security scan against the container using the latest reported CVEs. Based on the output of the scanner report and a container information dataset, we demonstrate an LLM agent powered system can gradually reduce the effort required for cybersecurity analysis and verification triggered by CVEs.

To achieve this, we propose a multi-component LLM system comprising an agentic-style LLM guided by sets of checklists generated from scanner reports and CVE intel. By harnessing the proposed system, organizations can significantly reduce the amount of human effort needed to analyze multiple containers and CVEs simultaneously, thereby enhancing the efficiency and effectiveness of vulnerability verification processes.

In the remainder of this work, we provide background on CVE analysis and AI agents; later, we dive into details of the proposed framework implementation and finally, benchmark results on both synthetic data and human labeled data

## 2. Related Works

### 2.1. LLMs for Vulnerability Detection

The application of LLMs to vulnerability detection has been extensively explored, with a focus on fine-tuning LLMs to identify vulnerable code fragments. Recent studies have showed fine-tuning LLMs for binary classification of source code fragments to detect vulnerabilities [10, 11, 12]. To evaluate the effectiveness of LLMs in vulnerability detection, Gao et al. developed comprehensive benchmarks such as VulBench [13]. More recently, prompt-based approaches utilizing GPT-3.5 and GPT-4 have been employed to improve vulnerability detection accuracy [14, 15, 16]. For example, Purba et al. work demonstrated the application of several GPT models for vulnerability detection, primarily relying on source code representations, and prompt engineering [14]. While these methods have shown promising results, they are not without limitations, including high false positive detection rates.

### 2.2. LLM Agents and Security

Several recent studies have shown LLM agent capability at various tasks [3, 4]. LLM agents in cybersecurity have been shown to enhance security capabilities as a knowledge assistant [17, 18, 19]. These use cases often rely on the use of tools [20, 21] to guide LLM agents [22, 23]. In contrast, our work combines the application of LLM agents in vulnerability exploitability verification. Building on previous studies, we extend the scope of LLM agent-based vulnerability remediation by incorporating environmental data, in addition to source code, to assess the exploitability of CVEs for target containers. Our approach offers a new perspective on the intersection of LLM agents and vulnerability research, highlighting the potential for LLMs to improve the efficiency and effectiveness of vulnerability exploitability verification.

# 3. Proposed Model

## 3.1. Problem Statement

Container vulnerability scanning typically involves generating a scan report using a vulnerability scanner, which produces a list of potential CVEs for the target container [6]. However, this raises critical questions:

- Given a CVE description, is my container vulnerable to the specified vulnerability?
- If I have a specific vulnerable package, under what conditions is the container exploitable?
- Are all detected CVEs present in my container?

To address these questions, we break down the problem into three sub-task objectives:

**Sub-Task 1: Optimal Checklist Generation**, for a given CVE, what is the optimal list of vulnerability and exploitability checks required to determine if the container is vulnerable?

**Sub-Task 2: Vulnerability Determination**, based on the generated checklist, do any of the checks satisfy the conditions to guarantee the container as "vulnerable" or "exploitable"?

**Sub-Task 3: Vulnerability Summarization and Justification**, based on the workflow output, can we categorize the models output into justifiable standard advisory format, such as VEX?

To tackle Sub-Task 1, we propose a novel approach using an LLM as a checklist generator. By crafting well-designed prompts, we task the LLM to generate validation task for a given CVE, ensuring a comprehensive evaluation of the container's vulnerability. To address Sub-Task 2, we build upon the success of LLM agents and propose an LLM agent with plan and execute capabilities [22, 23]. Leveraging access to various container information, including SBOM, source codes, and documentation, our proposed agent incorporates chain of thought-reasoning capabilities to determine the container's exploitability and vulnerability. Finally, for Sub-Task 3, we propose to prompt a pretrained Llama3-70B model [24], to categorize the output of agent response into subclass category of VEX justification format.

## 3.2. System Workflow

The system has three major components: a checklist generator, an LLM agent planner, and justification and summary components. We show an overall architectural diagram in Figure 1.

A checklist generator is initiated when a vulnerability scan event triggers the workflow by passing on a list of CVEs detected in the container. These results are combined with upto- date vulnerability and threat intelligence to provide the workflow with real-time information on the specific CVEs and their known exploitation status. This creates a list of plans and forwards the information to the LLM agent component.

The LLM agent uses the checklist to query for verification and exploitability checks. The LLM agent uses container data, such as SBOM, source code, and documentation for verification. In addition to data sources, the LLM agent has access to tools that help it overcome some of the current limitations of LLMs. For example, a common weakness of LLMs is their difficulty with performing mathematical calculations. This can be overcome by giving LLMs access to calculator tools. For our workflow, we found that the model struggled to compare package semantic version numbers such as version 1.9.1 coming before 1.10. We built a version comparison tool that the agent uses to determine the relationship between package versions.

Finally, the check information is passed to the summarization and justification component. The summarization process retains information from the previous steps and concludes with additional reasoning regarding the exploitability of the CVE. The justification process assigns Vulnerability Exploitability eXchange (VEX) justification for the identified response. These VEX responses follow the format of [25, 26]. We describe the components in detail as follows.

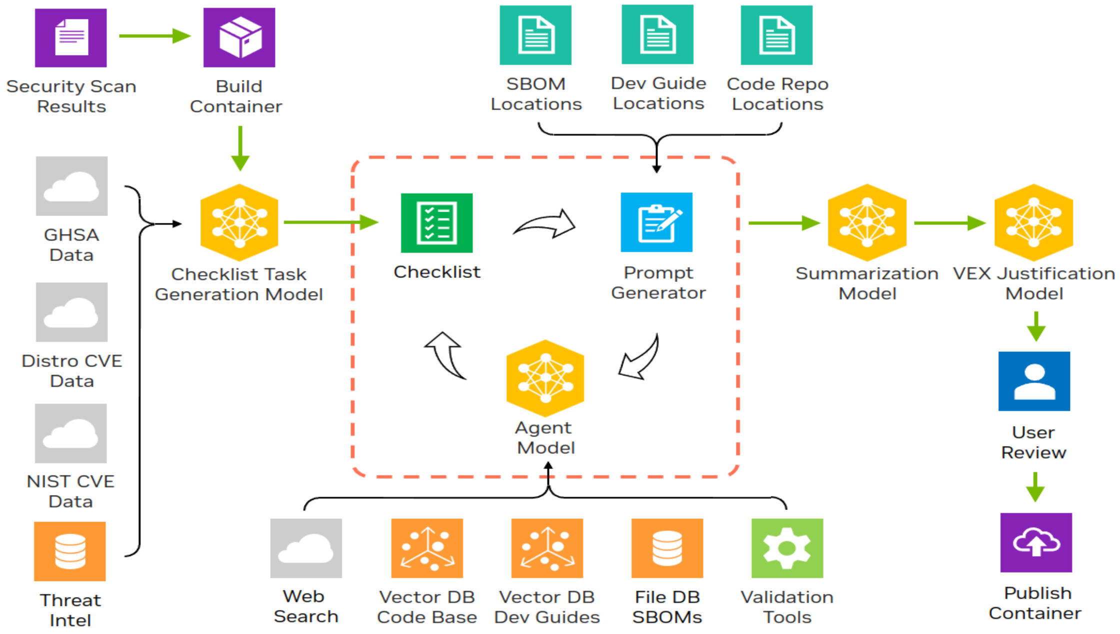


**Figure 1:** End-to-end workflow of the proposed system

### 3.2.1. Checklist Generator Model

The checklist model uses a generative LLM to serve as the surrogate for a cybersecurity analyst. Specifically, it constructs sequence of steps to check if a container is vulnerable to a given CVE. The model generates a customized checklist based on the CVE description and threat intelligence information related to the container and the vulnerability. Once created, the checklist items are passed to an LLM agent for further investigation. An example of generated checklist for *CVE-2021-41496: "Buffer overflow in the* $array\_from\_pyobj$ *function of* $fortranobject.c$ *in* $NumPy < 1.19$*, which allows attackers to conduct a Denial of Service attacks by carefully constructing an array with negative values".*

Then an example of generated checklist would be:

1. Check Numpy Version: The vulnerability affects Numpy versions before 1.19.0. What version of Numpy is installed in the Docker container? Is the container running a vulnerable version?
2. Identify Buffer Copy Operations: Review the application code within the Docker container to check for buffer copy operations without size checking. Are there any instances of buffer copying without proper input validation?
3. Assess Input Data Handling: Does the application process potentially untrusted input data that could trigger a buffer overflow? Evaluate how the application handles large or malformed input data.
4. Buffer Overflow Mitigations: Review the application's buffer overflow mitigation measures. Are there any protections in place to prevent buffer overflows, such as input validation, data sanitization, or address space layout randomization (ASLR)?

We designed five criteria for creating checklist items: actionability, simplicity, completeness, context relevance, and achievability using the provided agent tools. See Table 1 for the specific breakdowns for each of these categories. Additionally, model consistency criteria – that repeated calls to the model generate consistent outputs – is also considered along with these criteria.

### 3.2.2. LLM Agent Planner

The agent planner takes a created checklist as input to perform several checks for the target container. We adopt a ReAct agent model [22]. The ReAct agent model integrates reasoning capabilities of LLM

**Table 1**
checklist generation metric criteria

| | |
|---|---|
| Actionability | Does the checklist provide active and specific steps to be taken? |
| Simplicity | How simple is the operation to perform? |
| Completeness | Is the item well-formed and contains all needed information? |
| Context Relevance | Is the item relevant to the given context or scenario? |
| Achievable | Is the checklist item achievable using the provided source code, SBOM, or documentation of the container? |

with an ability to take actionable steps. The process involves "reasoning" and "acting". A typical ReAct agent consists of an LLM as its core, tools for interacting with external information, and chain-of-thought (CoT) prompting, allowing reasoning traces to create and maintain an adjustable action plan. The LLM agent is provided with the following tools to interact with the environment:

1. Web search results
2. SBOM package checker
3. Source code similarity checker tool
4. Documentation retrieval tool
5. Software version comparison tool

During agent process analysis, the LLM uses in-context learning to select tools that guide a specific task. For example, an LLM agent running for the above CVE-2021-41496, could be as follows.

```
> Entering new AgentExecutor chain...

> I need to check if NumPy is present in the Docker container and if so, determine its version
to see if it falls within the vulnerable range.

> Action: SBOM Package Checker

> Action Input: NumPy

> Observation: 1.22.2

> Thought: NumPy is present in the Docker container and the version is 1.22.2, which is not
vulnerable. I need to compare this version with the vulnerable range to confirm.

> Action: Docker Container software version comparator

> Action Input: 1.22.2,1.19,1.22

> Observation: False

> Thought: The NumPy version 1.22.2 in the Docker container is not vulnerable as it falls
outside the vulnerable range of versions prior to 1.19.

> Final Answer: The NumPy version in the Docker container is 1.22.2, which is not vulnerable
to the reported CVE. Finished chain
```

As we can see SBOM Package Checker tool is used for availability of NumPy package, and Software version comparator tool is used for comparing vulnerable Numpy versions before reaching final answer.

### 3.2.3. Summarization and Justification Process

Once the agent has compiled findings for each checklist item, the summarization model condenses these results into a concise, human-readable paragraph. This model uses an LLM to perform standard summarization tasks. The summarization process retains information from the previous steps and concludes with additional reasoning regarding the exploitability of the CVE.

Following the established standards of the Vulnerability Exploitability eXchange (VEX), the Justification Model assigns a VEX status to each identified vulnerability based on summarized findings [26]. Using an LLM for text-to-multiclass classification, this model categorizes CVEs by their exploitability in the given environment, detailing how they might be exploited or why they cannot be exploited. This classification step concludes the pipeline with a single label, which aids informed decision-making and allows for automation in downstream security systems.

## 4. Experimental Setup

To validate the proposed models, we conduct experiments at various stages of the workflow process. First, we measure the efficacy of the checklist model to guide the agent planner. Next, we evaluate the ability of the LLM agent to accurately identify false positives and, how different LLM models compare on their test dataset performance. Finally, we examine how often the LLM agent's final justification matches human evaluation.

### 4.1. Dataset

For the experiments, we employed the Anchore vulnerability scanner tool to generate SBOM files and scanner reports, which were used to collect datasets for our experiments. The tool was used to discover CVEs in a target Docker container and identify all available packages within it. We created two datasets for all test cases.

**Synthetic Dataset**: A synthetic dataset was generated by inserting CVEs trigger into the scanner report and manually modifying the versions of vulnerable packages in the container. In total, 45 CVEs collected using *Tensorflow:23.08* and *morpheus:23.07* release docker containers.

**Human generated Dataset**: Based on the output of Anchore scanner report, a checklist was generated for each triggered CVE. A team of container owners were asked to provide labeled responses for each checklist, which were informed by container information (SBOM, documentation) and base source code inspection of the target container. For this setup, total 35 CVEs with 96 checklists query pair collected from *morpheus:23.11-runtime* docker container.

### 4.2. Implementation and LLM models

We build the workflow pipeline using NVIDIA Morpheus framework [27] on top of generative LLMs. For the experiments we employed opensource LLM models served at [28]. We tested the experiments using Llama3-8B, Llama3-70B [24], and Mixtral-8x22B [29] models.

## 5. Evaluation and Results

### 5.1. Checklist Generation Model

To measure the efficacy of the checklist generation at a greater scale than hand-labeling would provide, we leverage LLM-as-a-judge[30]. The first steps were to hand score a selection of checklist items based on the criteria from Table 1. Once this sample was scored, we used LLM-as-a-judge to verify the applicability of scoring the items. Figure 2 shows that both GPT-4 and Llama 3 lined up well with the human labels, with GPT-4 performing better. We measure the squared error gap between the LLM-judge and Human judge, with average more than 75% agreement in our test dataset. This aligns with the reported in human and LLM-judge agreement and the expected human-to-human error gap [30]. Satisfied with the ability for an LLM to provide insight into these metrics, experiments comparing zero shot prompts vs. few shot prompts were run (See Figure 3 (a) and (b)). These experiments consisted of generating checklists for six vulnerabilities and averaging each individual metric over the vulnerabilities

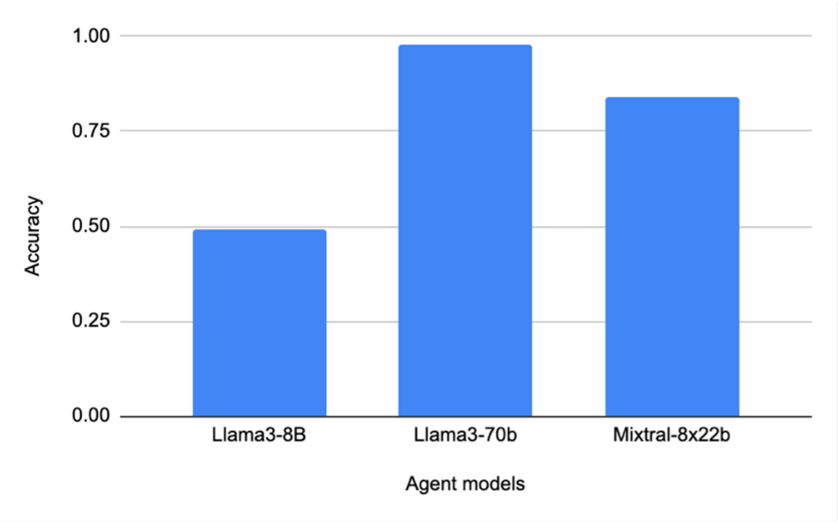
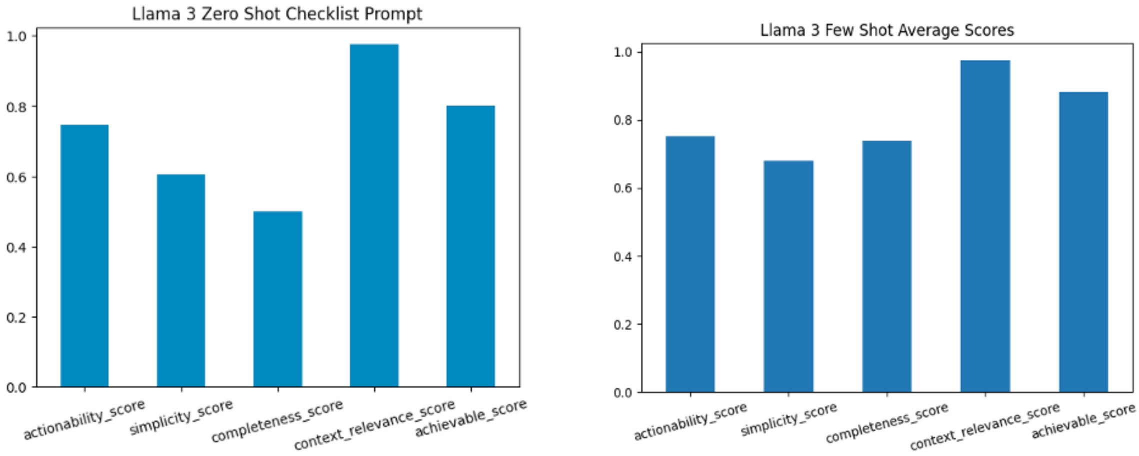


**Figure 2:** Error gap in agreement between human judge and LLM judge



(a) Zero shot prompting

(b) Multi-shot prompting

**Figure 3:** LLM checklist generation scores

to give a single score for the different prompt types. Using examples in the prompt led to overall better scores compared to the zero-shot approach.

In terms of consistency, the first measure was to see if repeated runs produce identical checklists. Although not identical, the different runs produced largely similar checklists. To explore this further, 18 sets of checklists were generated for 6 different vulnerabilities. Table 2 shows the number of unique checklists generated for each vulnerability. To give a metric for the overall similarity of the generated checklists, the checklist items were shingled into character 4-grams so that a weighted Jaccard distance could be taken between each set of checklists. Using this distance, the diameter (i.e., the largest distance observed between pairs of checklists) of the group was estimated (see Table 3). The observed diameters were within the accepted bound of error, except for *CVE-2023-24540* and *GHSA-5wvp-7f3h-6wmm*; due to an extra checklist being generated for some generations. Adjustments to the examples used in the few shot prompt have led to reducing the diameter for *CVE-2023-24540* to 0.171 and *GHSA-5wvp-7f3h-6wmm* to 0.384.

**Table 2**
Checklist generation count agreement

| Vulnerability | Checklist Counts | |
|---|---|---|
| | Created | Unique |
| CVE-2023-24538 | 18 | 7 |
| CVE-2023-24540 | 18 | 11 |
| CVE-2023-29402 | 18 | 8 |
| CVE-2023-29404 | 18 | 6 |
| CVE-2023-29405 | 18 | 12 |
| GHSA-5wvp-7f3h-6wmm | 18 | 14 |

**Table 3**
Jaccard distance of generated checklist

| Vulnerability | Group Diameter |
|---|---|
| CVE-2023-24538 | 0.195911 |
| CVE-2023-24540 | 0.460748 |
| CVE-2023-29402 | 0.140586 |
| CVE-2023-29404 | 0.248237 |
| CVE-2023-29405 | 0.233309 |
| GHSA-5wvp-7f3h-6wmm | 0.436236 |

## 5.2. LLM Agent

In this section, we describe experiments to validate the LLM agent's response to container scans' false positives and its overall response to checklist investigations.

A false positive trigger happens when a container scanner (e.g., Anchore scanner) performs scanning of CVEs, sometimes the scanner triggers CVEs that do not exist in the container. This mostly happens due to signature mismatch in the SBOM and reported package. Validating these findings can be a time-consuming task for analysts when the containers are not vulnerable.

To address this question, we create a dataset of synthetic SBOM files and scanner reports for a target container. The scanner report consists of both false positive and valid CVEs of the container. The vulnerability consists of version lower or equal to the vulnerable package, missing packages in the SBOM, and existing vulnerable in the SBOM. We measure the system's ability to identify both false positive CVEs and true positive vulnerable packages. We measure accuracy against ground truth label that indicates whether the CVE is "vulnerable" or "non vulnerable" for the target container. We tested three agent models based on Llama3-8b, Mixtral-22x7B and Llama3-70B against ground truth dataset. The result in Figure 4 shows the larger models are significantly better than the smaller models at identifying the false positive and vulnerable packages. The small model Llama3-8b tends to suffer at using appropriate tools, e.g., not using version comparison tool when comparing versions or unable to parse versions. This is not surprising, as smaller models tend to be worse at function call compared to large models [31].

Next, we evaluate the LLM models for agent response on the human labeled dataset. The goal is to evaluate LLM agent final answer to the input checklist query. We measure the performance with combination of LLM-as-Judge[30] and ground truth accuracy metrics. For LLM-as-Judge we use metrics such as *context relevance*, *answer relevance*, and *groundedness*. For ground truth agreement, we compare agent response token similarity against the token label provided.

The context relevance metric evaluates the alignment of the retrieved context with the query; answer relevance assesses the accuracy of the final answer and measure the extent to which the answer addresses the query; groundedness quantifies the degree to which the answer is supported by evidence from the retrieved documents. In this case, the retrieved documents are source codes, and documentation. At both answer and context relevance, all LLM agents achieve greater than 80% accuracy, with Llama3-70B

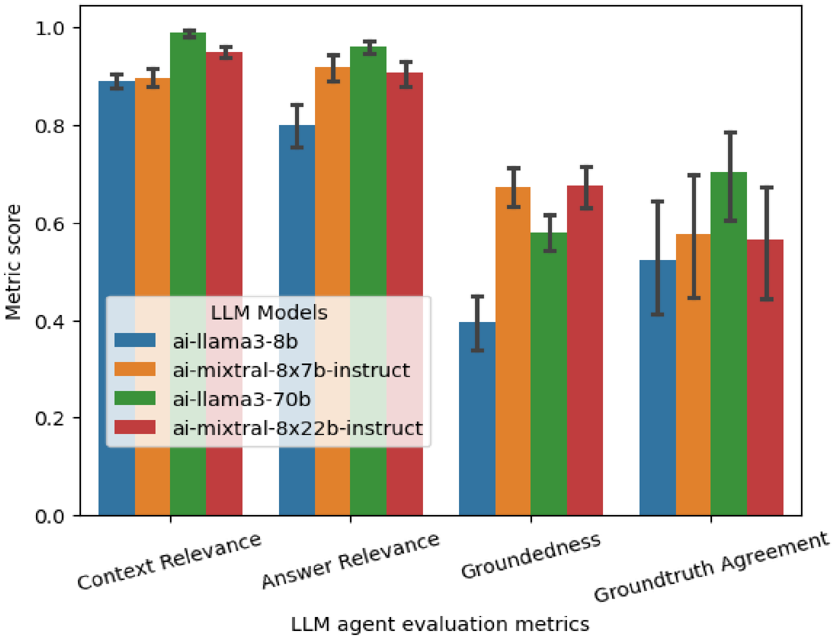


**Figure 4:** Evaluation of OSS LLM models for LLM agent on a human generated dataset

the highest. In comparison, the groundedness metric is lower compared to other metrics, with the best model averaging around 71%. This is expected because many of the retrieved documents do not necessarily end up being used for the final agent answer. For example, if the answer is not found in the retrieved source code or documentation, the agent might end up using internet search for answers.

Additionally, we measure agent response agreement to human provided feedback for the checklist. We compute similarity by measuring the agent token response against human provided feedback. Overall, the best model Llama3-70B achieves average agreement of 72% with the response feedback.

## 5.3. Justification and Summary Models

Finally, to measure the efficacy of the overall system, we compare the workflow justification results against a ground truth justification labeled by human annotators. The aim is to assess how often the workflow justification suggestion agrees with human justification. Using the best performing agent model, Llama3-70B, we summarize the response of the agent and format it as a VEX justification label to end users. We compare the VEX formatted output against human provided justifications in Figure 5 and Figure 6

The results in Figure 5 and Figure 6 show the LLM system's output compared to human annotators' labels. The labeled dataset consists of 35 CVEs that security analysts investigated against a given software container. Human analysts provided their final verdict on the exploitability (Boolean) of each CVE along with the reasoning categories. We provided a set of predefined categories (VEX justifications) for human analysts to choose from. If the CVE is deemed exploitable, the reasoning category is *"vulnerable"*. If it is not exploitable, there are 10 different reasoning categories to explain why the vulnerability is not exploitable in the given environment.

Overall, the accuracy of the pipeline's exploitability prediction is 75.7% (Figure 5). Considering the detailed reasoning for the non-exploitable classes, the pipeline's justification status accuracy is 54.0% (Figure 6). The pipeline has a high precision (92.9%) in predicting the vulnerable CVEs, which can help analysts prioritize and focus on the true positive that requires patching before having to investigate the entire batch of CVEs. The decent correlation between the pipeline output and the human labels shown in Figures 5 and 6 demonstrates that the pipeline is retrieving relevant information and performing meaningful evaluations of the CVEs' exploitability.

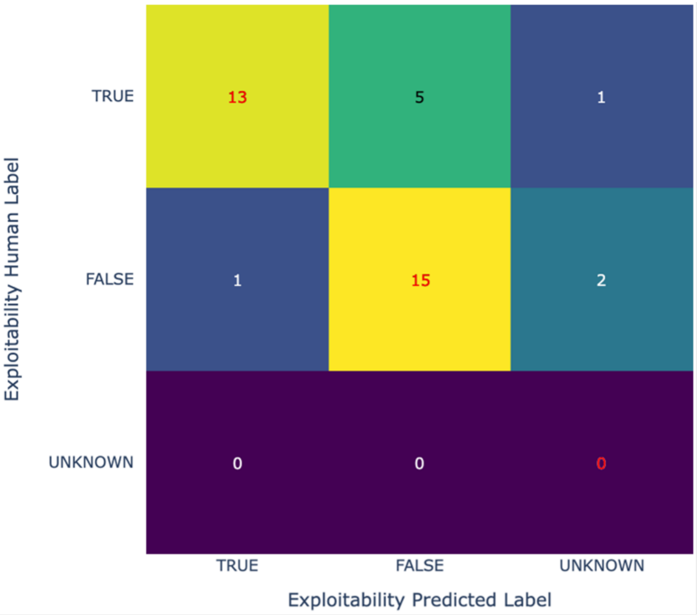


**Figure 5:** Confusion matrix comparing human exploitability labels with pipeline output.

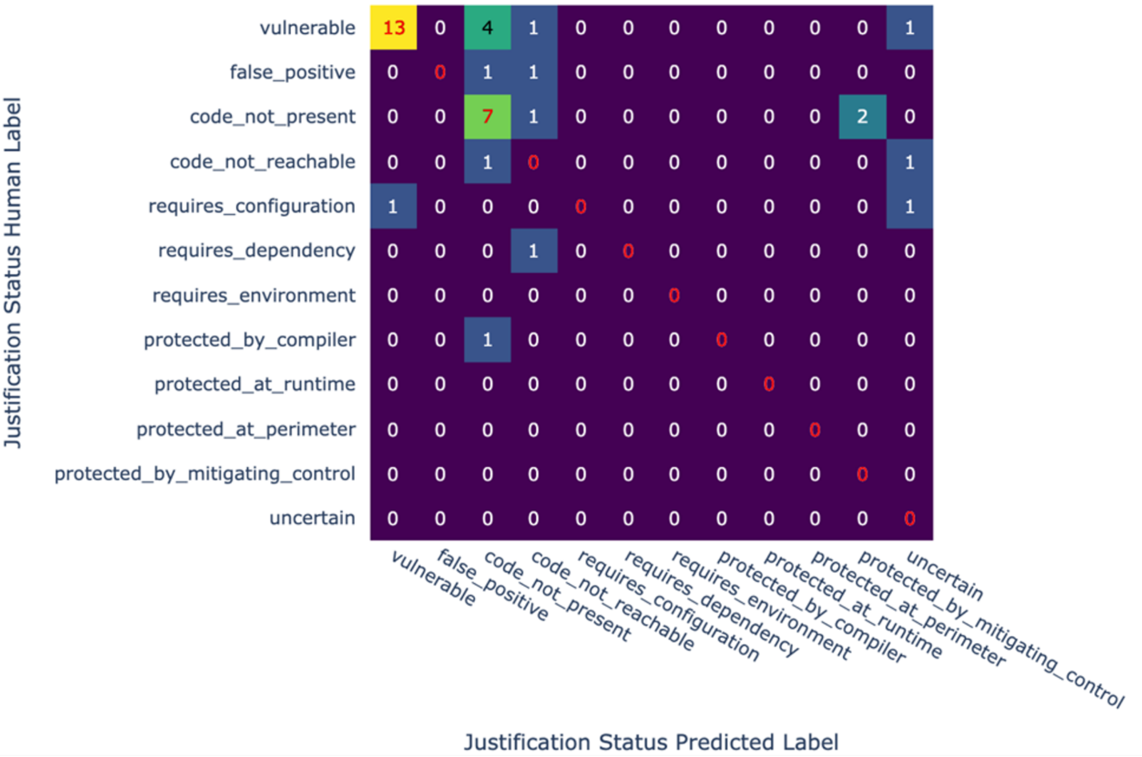


**Figure 6:** Confusion matrix comparing human justification status labels with pipeline output.

Evaluating the exploitability of a vulnerability involves many nuances. The verdict, including both the exploitable or not decision and the justification status, often depends on each organization's risk tolerance and each security analyst's perspective on risk evaluation. Even among human analysts, we observed frequent disagreements in the final exploitability and justification status assignments. The results show that using LLM to assist with CVE analysis and investigation is a promising application. While there is room for improvement in accuracy, this tool still significantly aids and expedites the decision-making process for security analysts by prioritizing alerts and providing context for the investigation.

## 6. Conclusion and Limitation

In this work, we demonstrate the potential of LLM agent in facilitating vulnerability and exploitability checks of Docker containers. To address vulnerability and exploitability checks, we propose a multi-component LLM agent system. Our results indicate that the proposed workflow can effectively perform Docker container vulnerability checks, leveraging only the container's input configuration and source codes.

In future works, we plan to explore extending agent capability for code path understanding, such as investigating execution path through LLM agent to further improve exploitability checks. Additionally, with enriching labeled dataset we foresee further improvement in the multilabel justification and summary models. With this we plan to explore fine-tuning LLM targeted further toward justification categories.

In summary, it is essential to note that this work does not encompass vulnerability verification at the host level or command execution level, such as scenarios requiring the execution of specific commands or access to the host operating system. This work's scope is limited to tasks involving verification at the resource level, specifically focusing on inputs like SBOM, source code, or documentation.

## References

[1] CVE Website, CVE common vulnerability exposure, 2024. URL: https://www.cve.org/.

[2] M. Rosen, Skybox security report reveals over 30,000 new vulnerabilities published in past year, https://www.skyboxsecurity.com/company/press-releases/skybox-security-report-reveals-over-30000-new-vulnerabilities-published-in-past-year/, 2023. Accessed: 2024-7-2.

[3] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, Ofir Press, SWE-agent: Agent-computer interfaces enable automated software engineering, arXiv [cs.SE] (2024).

[4] D. Huang, J. M. Zhang, M. Luck, Q. Bu, Y. Qing, H. Cui, AgentCoder: Multi-agent-based code generation with iterative testing and optimisation, arXiv [cs.CL] (2023).

[5] A. M Bran, S. Cox, O. Schilter, C. Baldassari, A. D. White, P. Schwaller, Augmenting large language models with chemistry tools, Nat. Mach. Intell. 6 (2024) 525–535.

[6] D. A. Boiko, R. MacKnight, G. Gomes, Emergent autonomous scientific research capabilities of large language models, arXiv [physics.chem-ph] (2023).

[7] J. Zhang, H. Bu, H. Wen, Y. Chen, L. Li, H. Zhu, When LLMs meet cybersecurity: A systematic literature review, arXiv [cs.CR] (2024).

[8] N. V. Database, NVD - vulnerabilities, https://nvd.nist.gov/vuln, 2023. Accessed: 2024-7-2.

[9] Anchore, Container vulnerability scanning & management •, https://anchore.com/container-vulnerability-scanning/, 2021. Accessed: 2024-6-12.

[10] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, M. McConley, Automated vulnerability detection in source code using deep representation learning, in: 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), IEEE, 2018.

[11] N. S. Harzevili, A. B. Belle, J. Wang, S. Wang, Z. Ming, Jiang, N. Nagappan, A survey on automated software vulnerability detection using machine learning and deep learning, arXiv [cs.SE] (2023).

[12] A. Shestov, R. Levichev, R. Mussabayev, E. Maslov, A. Cheshkov, P. Zadorozhny, Finetuning large language models for vulnerability detection, arXiv [cs.CR] (2024).

[13] Z. Gao, H. Wang, Y. Zhou, W. Zhu, C. Zhang, How far have we gone in vulnerability detection using large language models, arXiv [cs.AI] (2023).

[14] M. D. Purba, A. Ghosh, B. J. Radford, B. Chu, Software vulnerability detection using large language models, in: 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE, 2023.

[15] X. Zhou, T. Zhang, D. Lo, Large language model for vulnerability detection: Emerging results and future directions, arXiv [cs.SE] (2024).

[16] A. Cheshkov, P. Zadorozhny, R. Levichev, Evaluation of ChatGPT model for vulnerability detection, arXiv [cs.CR] (2023).

[17] V. Jakkal, Introducing microsoft security copilot: Empowering defenders at the speed of AI, https://blogs.microsoft.com/blog/2023/03/28/introducing-microsoft-security-copilot-empowering-defenders-at-the-speed-of-ai/, 2023. Accessed: 2024-6-23.

[18] A. Arora, A. Arora, J. McIntyre, Developing chatbots for cyber security: Assessing threats through sentiment analysis on social media, Sustainability 15 (2023) 13178.

[19] A. Happe, J. Cito, Getting pwn'd by AI: Penetration testing with large language models, arXiv [cs.CL] (2023).

[20] Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian, S. Zhao, L. Hong, R. Tian, R. Xie, J. Zhou, M. Gerstein, D. Li, Z. Liu, M. Sun, ToolLLM: Facilitating large language models to master 16000+ real-world APIs, arXiv [cs.AI] (2023).

[21] Z. Wang, Z. Cheng, H. Zhu, D. Fried, G. Neubig, What are tools anyway? a survey from the language model perspective (2024).

[22] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, Y. Cao, ReAct: Synergizing reasoning and acting in language models, arXiv [cs.CL] (2022).

[23] N. Shinn, F. Cassano, B. Labash, A. Gopinath, K. Narasimhan, S. Yao, Reflexion: language agents with verbal reinforcement learning, Adv. Neural Inf. Process. Syst. (2023).

[24] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan, et al., The llama 3 herd of models, arXiv preprint arXiv:2407.21783 (2024).

[25] Vulnerability exploitability eXchange (VEX) status justification document (june 2022), https://www.cisa.gov/resources-tools/resources/vulnerability-exploitability-exchange-vex-status-justification-document-june-2022, 2022. Accessed: 2024-6-11.

[26] CycloneDX - vulnerability exploitability eXchange (VEX), https://cyclonedx.org/capabilities/vex/, 2024. Accessed: 2024-6-23.

[27] NVIDIA, Morpheus: Morpheus SDK, https://github.com/nv-morpheus/Morpheus, 2024.

[28] Try NVIDIA NIM APIs, https://build.nvidia.com/explore/discover, 2024. Accessed: 2024-6-24.

[29] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. d. l. Casas, E. B. Hanna, F. Bressand, et al., Mixtral of experts, arXiv preprint arXiv:2401.04088 (2024).

[30] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, I. Stoica, Judging LLM-as-a-judge with MT-bench and chatbot arena, arXiv [cs.CL] (2023).

[31] Fanjia Yan and Huanzhi Mao and Charlie Cheng-Jie Ji and Tianjun Zhang and Shishir G. Patil and Ion Stoica and Joseph E. Gonzalez, Berkeley function calling leaderboard, https://gorilla.cs.berkeley.edu/blogs/8_berkeley_function_calling_leaderboard.html#citation, 2024. Accessed: 2024-6-11.