An Analysis of Real-World Vulnerabilities and Root Causes in the LLM Supply Chain

Vitor Hugo Galhardo Moia, Rodrigo Duarte de Meneses, Igor Jochem Sanz

Departamento de Computação Embarcada
Instituto de Pesquisas Eldorado – Campinas, SP – Brazil

{vitor.moia, rodrigo.meneses, igor.sanz}@eldorado.org.br

Abstract. *Generative Artificial Intelligence (GenAI) libraries are increasingly foundational in modern applications, with special attention to Large Language Models (LLMs), yet they exhibit evolving security vulnerabilities that necessitate comprehensive analysis. In this study, we analyze 719 CVEs disclosed for 93 LLM-related libraries (from 2019 to 2024), revealing that more than 37% of libraries have at least one CVE, and that an alarming 62% of vulnerabilities are of high or critical severity, demanding immediate action. Our findings aim to inform actionable recommendations that strengthen security practices and ensure the safe deployment of GenAI technologies.*

1. Introduction

The rapid advancement and widespread adoption of Generative Artificial Intelligence (GenAI) technologies – especially Large Language Models (LLMs) – have revolutionized areas from natural language processing to complex decision-making systems [Cui et al. 2024a]. This fast development stems from numerous open-source projects shared by the community, fostering a culture of code reusability for rapid innovation. However, the code shared through libraries, composing the supply chain of a system, underpins critical AI-driven functionality and has become a prime target for adversaries [NIST 2024a, Cui et al. 2024b]. Moreover, their complexity and dynamic behavior often conceal flaws that evade conventional static and dynamic analyses [Huang et al. 2024]. As a result, thousands of vulnerabilities are discovered and published each year via the Common Vulnerabilities and Exposures (CVE) program[1].

Understanding the evolution and root causes of security vulnerabilities is paramount to protect users and companies using LLM systems. To this end, we perform an analysis of software supply chain vulnerabilities of LLM systems, evaluating 719 CVEs (created over the period 2019-2024) drawn from 93 libraries that are used across the LLM life cycle. We present the most vulnerable libraries, taking into consideration their popularity and complexity metrics, correlate each flaw with its underlying root cause, and define mitigation and detection strategies to address the most common vulnerabilities in the field. Our goal is to furnish evidence-based priorities for hardening next-generation GenAI software while the ecosystem matures.

2. Background

To understand software vulnerabilities in real-world scenarios, one can think about the CVE program, which provides a standardized identifier for publicly known cybersecurity

[1]The CVE program. Available at `https://cve.mitre.org/`

vulnerabilities, facilitating information sharing and risk assessment. When detecting a vulnerability in a software, a cybersecurity specialist can disclose their findings via the CVE program, pursuing a formal process to report problems with a particular software to the community. The adoption of the CVE standard fosters the secure software culture and a rich unified database for software vulnerability research.

By analyzing CVEs from the software supply chain of LLM systems, it is possible to assess which vulnerabilities prevail and detect common threat patterns that organizations should be aware of. Additionally, the Common Weakness Enumeration (CWE)[2] system offers a taxonomy of software and hardware weaknesses, enabling a deeper understanding of the root causes of vulnerabilities, i.e., underlying developer errors that provoke vulnerabilities exploitable by adversaries. Since each CVE maps to a CWE when created, we can understand the most common threats to LLM systems and their root causes.

2.1. LLM Life Cycle

The life cycle of LLMs encompasses several stages [Suresh and Guttag 2021], each presenting distinct security considerations:

Data Collection and Preprocessing: In this initial phase, vast amounts of data are gathered, cleaned, and normalized. Some risks in this phase include data poisoning, where malicious actors inject harmful data to influence model behavior.

Model Training: Process of creating a model from scratch or fine-tuning an existing one with data obtained in the previous phase. The process relies on the integrity of both data and model, with risks of data poisoning or backdoor introduction via supply-chain.

Deployment: Once the model is packaged, it enters production and becomes ready for users. Threats include tampering and obtaining unauthorized access, via infrastructure/app compromise (e.g., exploring weak API controls, side-channel attacks, etc).

Inference and Output Handling: At runtime, user queries are converted into prompts, passed through the model, and the outputs are post-processed. Prompt-injection attacks can override system instructions, inadequate filtering may let toxic or hallucinated content through, and naively using user data to retraining models can enable feedback poisoning.

The LLM supply chain plays a vital role, as vulnerabilities can be introduced in any phase, allowing adversaries to explore the development or deployment environments. For this reason, a careful analysis of all elements involved in a system is paramount.

2.2. Related Work

Prior research has extensively explored vulnerabilities and defenses specific to LLMs [Cui et al. 2024a]. Recent studies have proposed comprehensive risk taxonomies and benchmarks tailored for LLM systems [Cui et al. 2024b, NIST 2024a], associated known CVE records to MITRE's CWE classifications [Haddad et al. 2023, Shi et al. 2024], and examined broader security and privacy challenges – including supply chain risks – within GenAI pipelines [Yao et al. 2024, Huang et al. 2024].

In contrast to prior studies, this work specifically analyzes software supply chain vulnerabilities associated with LLM systems through the lens of CVE data. Our contribution lies in identifying the most prevalent root causes of vulnerabilities in the field and

[2]Common Weakness Enumeration. Available at `https://cwe.mitre.org/`

subsequently defining concrete security measures for their detection and mitigation. This analysis fills a critical gap by directly connecting vulnerabilities to actionable recommendations, thus contributing to ongoing efforts to secure LLM systems.

3. Methodology

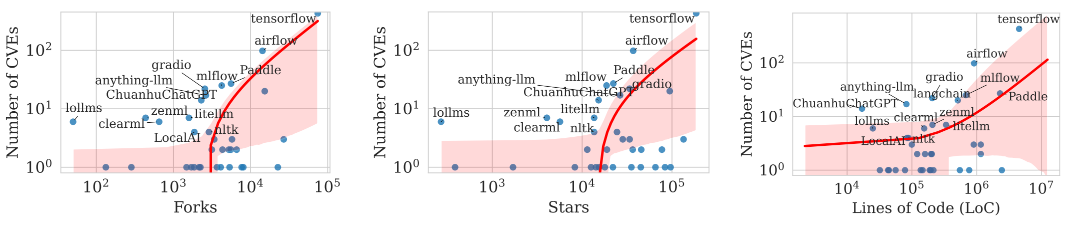
The first step in this study was to obtain a list of the most relevant software of LLMs supply chain to identify known vulnerabilities. The list was derived from the bug bounty platform *Huntr*[3], a place where security researchers can submit vulnerabilities for AI/ML apps and libraries. This platform was selected for concentrating the most well-known open source LLM libraries, with regular updates (inclusion of new libraries that are representative to the field). Although this platform also contains information about CVEs for some libraries, it presents a limited view of all existing vulnerabilities – focusing on those created via the platform itself – and it does not intend to gather and provide vulnerabilities from other sources. For this reason, we are using *Huntr* only to obtain library names, while obtaining all other library information elsewhere.

On October 31, 2024, we extracted all open-source projects available on *Huntr*. At that time, we identified 182 distinct libraries. Not all of these libraries are related to LLMs or are relevant to the field. For this reason, we established specific criteria to filter out the results. We retained only libraries with $\geq$ 100 GitHub stars, $\geq$ 50 forks, active status (not archived or disabled), and relevant to at least one phase of the LLM life cycle. The criteria based on stars and forks were inspired by prior studies [Borges et al. 2016, Hu et al. 2016], which identify these metrics as key indicators of software popularity and impact. For each selected library, we collected all CVEs published between 2019 and 2024 (on November 27, 2024), but limited our analysis to those having the *Analyzed* status to ensure only fully reviewed and vetted vulnerabilities were considered. Those under *Received* or *Awaiting / Undergoing Analysis* at the time we executed the experiments were discarded.

The time frame of 2019–2024 was deliberately chosen based on significant advances and widespread adoption of LLMs during this period. Specifically, the launch of GPT-2 in 2019 marked a shift in the scale and complexity of LLMs, triggering accelerated research, development, and deployment in both academia and industry. Consequently, this time frame ensures comprehensive coverage of vulnerabilities associated with contemporary LLMs and captures meaningful trends reflective of current industry practices.

To collect metadata from each library, we utilized the GitHub API[4]. Since all software in this study is open source and hosted on GitHub, we were able to extract all information necessary to our filtering process. Because subject matter relevance could not be inferred from metadata alone, we asked ChatGPT-4o-mini to rate each library's applicability to LLM life cycle tasks on a 0-5 Likert scale (retaining those scored $\geq$ 3). Finally, we manually reviewed the results to mitigate issues related to hallucinations or inaccuracies in this scoring process. After filtering and reviewing the data, we compiled a final list of 93 software relevant to the LLM ecosystem. We present in Table 1 (Appendix) the complete list of software chosen for our analysis, along with their GitHub status and

[3]Huntr: The worlds first bug bounty platform for AI/ML. Available at `https://huntr.com`
[4]GitHub API Documentation: `https://docs.github.com/en/rest`



(a) CVEs vs. Forks (b) CVEs vs. Stars (c) CVEs vs. Lines of Code

Figure 1. CVE occurrences versus GitHub repositories indicators: (a) Forks; (b) Stars; and (c) Lines of Code. Pearson coefficients: (a) $R = 0.85$ $(p = 0)$**; (b)** $R = 0.59$ $(p = 0)$**; (c)** $R = 0.29$ $(p = 0.005)$**.**

relevance score. Some edge cases (score < 3) were included due to our understanding that such software could potentially compromise an LLM application that utilizes it.

4. Results and Discussion

4.1. Software Libraries and Vulnerabilities

From the 93 libraries analyzed, we found 35 libraries (37.63%) having at least one CVE. By the time of our experiments (October 31, 2024), 98.75% of the analyzed CVEs received published fixes, indicating that most LLM-related vulnerabilities are remediated promptly once disclosed. Table 1 (Appendix) provides the CVEs count for each library of this study. The vulnerable software spans the LLM life cycle: 3 libraries used for data collection and preprocessing tasks, 13 for model development (training and testing), 21 for deployment, and 2 for inference and output handling.

Vulnerability Exposure. Analyzing vulnerability exposure requires considering not just the absolute CVE count, but also factors like complexity and popularity. We analyze the occurrence of CVEs to each library compared to its GitHub indicators: stars, forks and lines of code (LoC). The latter is obtained through the tool *cloc*[5]. While stars and forks are related to repository popularity and impact, the LoC is an indicator of complexity.

Figure 1 presents the number of CVEs versus each indicator. A regression line with 95% of confidence interval indicates the libraries with a disproportionately high number of CVEs relative to each indicator, while the Pearson coefficient R indicates the linear correlation of CVEs with each indicator. Among the three metrics, forks are the strongest indicator for CVE exposure, while stars express moderate linearity to security risks. Surprisingly, LoC exhibited a weaker correlation to CVEs, suggesting that active development and popularity better predict vulnerability exposure than codebase size alone. To assess an average level of CVE exposure for each library combining multiple indicators, we calculate the ratio of CVEs occurrences to the normalized indicator values. Next, we compute a geometric mean of the three ratios. Using this method, the top 10 libraries with most disproportionately high CVEs compared to their complexity and popularity combined are ChuanhuChatGPT, clearml-server, airflow, tensorflow, zenml, anything-llm, gradio, clearml, mlflow, and Paddle. Note that using this additional data, libraries that have only one CVE (clearml-server) appear in the top 10 list, while others with more vulnerability disclosures remained out (e.g., langchain).

[5]Count Lines of Code (cloc). Available at `https://github.com/AlDanial/cloc`.

Predominant Programming Language. The most frequently used programming language in vulnerable libraries (the main one in the repository) is Python (68.1%), followed by C++ (8%), TypeScript (7%), and Go (6%). Knowing the language used by a software gives a hint about the types of vulnerabilities to expect. Note that many projects may contain modules or specific functions written in a different language than the main one.

Temporal Trends. Figure 2 presents the distribution of CVEs across yearly quarters for all published CVEs and for the LLM software considered in our study. There is an increase, quarter by quarter, in the number of CVEs being published, but to determine whether the CVE disclosures is also increasing for the LLM software, we applied the Mann–Kendall trend test to the quarterly counts (Q1-19 through Q4-24). The test yields S = 91 (variance = 1625.33), Z = 2.232, Kendall's τ = 0.33, and two-sided p-value of 0.026. Since $p < 0.05$, we conclude there is an upward trend in reported vulnerabilities over this period. This trend suggests increased research and reporting of vulnerabilities during the initial adoption phases of LLM-related technologies, which subsequently stabilized as these technologies matured and security practices improved.

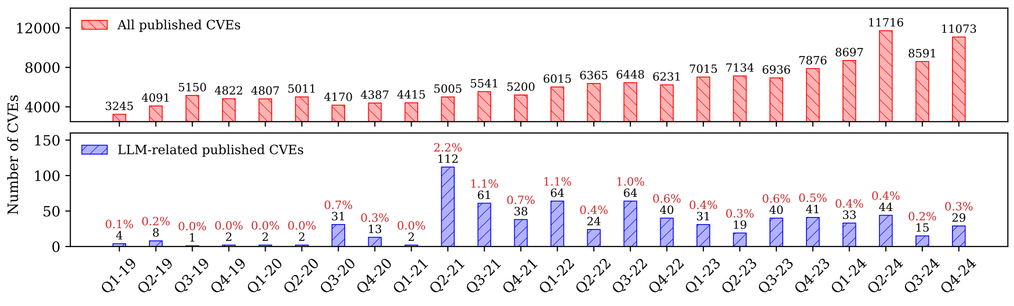


Figure 2. Quarterly CVE counts for LLM-related software compared to all published CVEs records (Q1 2019 – Q4 2024).

Severity Score. For determining the severity level of vulnerabilities, the CVE program adopts the CVSS (Common Vulnerability Scoring System) methodology. By analyzing the score of each vulnerability according to the attributed CVSS (version 3.1)[6], we found that 8 (1.11%) CVEs have a *Low* score and 260 (36.16%) a *Medium* score. An alarming number of 355 (49.37%) CVEs present a *High* score and others 96 (13.35%) a *Critical* one, indicating that more than 62% of vulnerabilities demand immediate attention due to their potential impact on the security and privacy of users and companies.

4.2. Root Causes of Vulnerabilities

To understand the root causes of vulnerabilities in LLM libraries, we analyzed the CWE attributed to each CVE. The CWEs follow a structured organization using Views, which is a subset of CWEs grouped based on some criteria. Our analysis is based on MITRE's CWE View 1000, containing 10 Pillars that group weaknesses based on how they can be detected [MITRE 2025]. Of the 97 identified CWEs, two were classified as NVD-CWE-Other (unsupported types), and 15 as NVD-CWE-noinfo (insufficient information). The resulting Pillar-level frequencies are summarized in Table 2 (Appendix). It is worth

[6]Common Vulnerability Scoring System v3.1: Specification Document. Available at `https://www.first.org/cvss/v3-1/specification-document`

noting that some CWEs share similar root causes and are therefore mapped by MITRE to multiple Pillars. For instance, CWE-476 is included in both Pillar 703 and Pillar 710; in this case, the frequency of CWE-476 is counted separately for each mapping.

The Pillar with most CVEs is CWE-664, encompassing 119 cases (CWE-119, 120, 122, 125, 415, 416, 787, 824) related to out-of-bound operations on memory or files, resulting in vulnerabilities like buffer overflow, use-after-free, double free, etc., most frequent in C/C++ software. This group also includes 40 cases (CWE-22, 23, 29, 36) related to language-agnostic path-traversal flaws occurring during name or reference resolution.

The most frequently occurring CWE is CWE-20 (Improper Input Validation) with 66 instances, belonging to the broader CWE-707 (Improper Neutralization) Pillar, underscoring the critical importance of correctly handling user input. The CWE-707 Pillar also covers flaws in which untrusted input – when used to build commands, data structures, or records – can alter program execution when the data is parsed or interpreted. This is the case of CWE-74, and within this class, 84 occurrences were found (CWE-75, 77, 78, 79, 88, 89, 94, 1236, 1336), allowing adversaries to perform different forms of injection attacks, via command, OS command, argument, template, code, SQL, special element, and Cross-site Scripting. With a few exceptions (CWE-94 and CWE-1336 for interpreted languages), all of these weaknesses are language-agnostic.

Other frequent CWEs were CWE-369 (Pillar 682) and CWE-476 (Pillars 703 and 710). The first one is a Divide by Zero weakness, with 59 occurrences of unexpected values provided to applications without a proper inspection and validation, e.g., providing a zero as denominator to a division function, crashing the application. This type of failure may occur in any programming language. On the other hand, CWE-476 is a NULL Pointer Dereference weakness (58 occurrences), with the application trying to access a value stored in a memory address referenced by a given pointer that is invalid (NULL), resulting in crashes or even allowing code execution. This problem is specific to some programming languages, such as C, C++, Java, C#, and Go, and is part of the CWE-754 class (11 occurrences), also related to the lack or improper check for unusual or exceptional conditions, but in this case, this is not specific to any programming language.

5. Recommendations and Best Practices

Drawing on the 10 most frequent CWEs in our analysis and the mitigation guidance provided by MITRE [MITRE 2025], we organize the recommendations into:

a) Preventive Controls: Enforce strict validation and sanitization of all inputs (e.g., JSON Schema, Pydantic), apply consistent encoding/decoding (e.g., HTML-escape, URL-encode), and separate user data from code (to mitigate CWE-20, 79, 22, 94, 125, 190, 476, 617); rely on vetted, purpose-built libraries for parsing and serialization to reduce unsafe deserialization and buffer flaws (CWE-22, 79, 190, 787); adopt memory-safe languages or frameworks (CWE-125, 190, 476, 787) and do compiler/build hardening – e.g., ASLR, buffer overflow detection, examining build warnings – (CWE-94, 190, 787); run risky operations in containers or sandboxes, enforce least-privilege execution, and harden environments (CWE-22, 79, 94); and ensure error messages only reveal minimal, non-sensitive information to avoid aiding adversaries (CWE-22).

b) Detective Controls: Integrate automated static analysis tools into CI/CD pipelines to catch improper validation, null dereferences, and injection patterns (CWE-20, 22, 79, 94, 125, 190, 369, 476, 617, 787); employ fuzzing and runtime monitoring to uncover logical errors, memory corruption, and unexpected behavior under malformed inputs (CWE-20, 22, 125, 190, 369, 476, 787); conduct source-code and architecture reviews, especially for high-risk components, validating boundary checks, data-flow separation, and error-handling (CWE-20, 22, 190); and leverage OWASP Cheat Sheets, automated test-generation, pentests, and threat modeling to identify exploit chains (CWE-79, 190).

These recommendations are aligned with those suggested by NIST [NIST 2022], OWASP [OWASP 2024], and other entities. Companies and software developers can adopt these strategies in addition to applying security principles [NIST 2024b, CISA 2025] and best practices, as well as monitoring their software supply chain for vulnerabilities. For end users, the main lesson is the importance of keeping software up-to-date, since a disclosed vulnerability is most likely to be fixed in newer software versions.

6. Limitations

This study may have bias due to some choices in the experiments, such as:

- **Use of Open Source Projects.** All LLM-related libraries analyzed are open-source projects; as a result, they tend to be more scrutinized and have more vulnerabilities reported compared to proprietary solutions. We should not consider the absolute number of vulnerabilities as a bad indicator, but take into consideration other aspects, e.g., number of fixed issues, time to handle issues, number of vulnerabilities without a fix, etc.
- **Data Source for Libraries List.** We are not aware of any study presenting reliable statistics about LLM-related libraries adoption at the time of conducting this study. We found *Huntr* to be a good alternative for selecting the libraries to be analyzed, since it gathers the most relevant ones in the field, which offers the greatest value to the community. Besides, our interest is not in analyzing all available libraries, but rather the most representative ones that can significantly impact the field.
- **LLM for Classification.** Using an LLM to define which libraries are related to LLMs life cycle phases may introduce inaccuracies due to hallucination. We performed a manual revision to mitigate any issues.

7. Conclusion

Over the past five years, the number of reported vulnerabilities in GenAI applications has surged, with disclosures skyrocketing since the beginning of 2021. As presented, most LLM-related vulnerabilities are situated in the high-to-critical severity range, highlighting the urgency for preemptive mitigation as popular and actively maintained software accounts for the majority of these reports.

Our findings underscore the critical need to integrate robust security measures throughout the GenAI software development life cycle. The recurrence of specific CWEs highlights areas where developers must concentrate their efforts to mitigate threats effectively, prioritizing the recommendations compiled in this work. Future research should cover how vulnerabilities are evolving in the field and how to promote a culture of security awareness among developers and stakeholders. Creating automated methods to assess risks of GenAI software supply chain, including data from reported vulnerabilities, code complexity and popularity, among other aspects, is also planned.

References

Borges, H., Hora, A., and Valente, M. T. (2016). Understanding the factors that impact the popularity of github repositories. In *2016 IEEE ICSME*, pages 334–344.

CISA (2025). AI Cybersecurity Collaboration Playbook. Technical report, Cybersecurity and Infrastructure Security Agency. Available at `https://www.cisa.gov/resources-tools/resources/ai-cybersecurity-collaboration-playbook`. Accessed 2025-07-21.

Cui, J., Xu, Y., Huang, Z., Zhou, S., Jiao, J., and Zhang, J. (2024a). Recent advances in attack and defense approaches of large language models. *arXiv preprint arXiv:2409.03274*.

Cui, T., Wang, Y., Fu, C., Xiao, Y., Li, S., Deng, X., Liu, Y., Zhang, Q., Qiu, Z., Li, P., et al. (2024b). Risk taxonomy, mitigation, and assessment benchmarks of large language model systems. *arXiv preprint arXiv:2401.05778*.

Haddad, A., Aaraj, N., Nakov, P., and Mare, S. F. (2023). Automated mapping of cve vulnerability records to mitre cwe weaknesses. *arXiv preprint arXiv:2304.11130*.

Hu, Y., Zhang, J., Bai, X., Yu, S., and Yang, Z. (2016). Influence analysis of github repositories. *SpringerPlus*, 5:1–19.

Huang, K., Chen, B., Lu, Y., Wu, S., Wang, D., Huang, Y., Jiang, H., Zhou, Z., Cao, J., and Peng, X. (2024). Lifting the veil on the large language model supply chain: Composition, risks, and mitigations. *arXiv preprint arXiv:2410.21218*.

MITRE (2025). CWE - Common Weakness Enumeration. Available at `https://cwe.mitre.org/index.html`. Accessed: 2025-07-21.

NIST (2022). Secure software development framework (ssdf) version 1.1: Recommendations for mitigating the risk of software vulnerabilities. NIST Special Publication 800-218, National Institute of Standards and Technology.

NIST (2024a). Adversarial Machine Learning: A Taxonomy and Terminology of Attacks and Mitigations. Technical Report 100-2 E2023, National Institute of Standards and Technology.

NIST (2024b). Artificial Intelligence Risk Management Framework: Generative Artificial Intelligence Profile. Technical Report 600-1, National Institute of Standards and Technology.

OWASP (2024). OWASP top 10 proactive controls: Top 10 2024. Available at `https://top10proactive.owasp.org/archive/2024/`. Accessed: 2025-07-21.

Shi, Z., Matyunin, N., Graffi, K., and Starobinski, D. (2024). Uncovering CWE-CVE-CPE relations with threat knowledge graphs. *ACM Trans. Priv. Secur.*, 27(1).

Suresh, H. and Guttag, J. V. (2021). A framework for understanding sources of harm throughout the machine learning life cycle. In *EAAMO '21*. ACM.

Yao, Y., Duan, J., Xu, K., Cai, Y., Sun, Z., and Zhang, Y. (2024). A survey on large language model (llm) security and privacy: The good, the bad, and the ugly. *High-Confidence Computing*, 4(2):100211.

Appendixes

Table 1. LLM Related Software – Repository Information and Stats.

Name	Stars	Forks	LoC (x1000)	Score	CVEs
tensorflow	186,223	74,298	4,514	3	430
airflow	37,003	14,270	910	1	98
Paddle	22,237	5,584	2,291	3	27
mlflow	18,699	4,229	679	3	25
gradio	33,731	2,556	206	4	22
langchain	94,455	15,268	512	5	20
anything-llm	26,439	2,634	82	5	17
ChuanhuChatGPT	15,241	2,297	17	5	14
litellm	13,600	1,591	337	5	7
zenml	4,046	436	207	3	7
clearml	5,665	653	155	3	6
lollms	269	50	25	4	6
LocalAI	24,424	1,868	84	5	4
nltk	13,596	2,887	88	4	4
transformers	134,501	26,897	1,158	5	3
ray	33,775	5,746	900	3	3
pytorch-lightning	28,327	3,381	100	3	3
fastapi	77,267	6,601	203	1	2
open-webui	45,503	5,559	121	5	2
llama_index	36,534	5,226	1,159	5	2
LibreChat	18,874	3,145	198	5	2
composio	11,400	4,328	159	2	2
ollama	96,473	7,664	189	5	1
pytorch	83,664	22,577	2,446	3	1
gpt_academic	65,445	8,051	44	5	1
MetaGPT	44,954	5,347	56	5	1
DeepSpeed	35,348	4,102	195	4	1
ragflow	21,995	2,156	767	4	1
onnx	17,888	3,669	215	3	1
SuperAGI	15,446	1,859	43	5	1
horovod	14,247	2,239	78	3	1
dask	12,568	1,707	146	3	1
server	8,297	1,479	137	4	1
modeldb	1,700	285	550	3	1
clearml-server	385	133	32	3	1
flask	68,016	16,214	19	1	0
keras	61,973	19,470	149	3	0
private-gpt	54,082	7,268	9	5	0
dify	50,817	7,306	573	5	0
FastChat	36,886	4,544	39	5	0
AgentGPT	31,720	9,232	49	5	0
faiss	31,300	3,631	134	3	0
fairseq	30,457	6,405	179	4	0
jax	30,405	2,789	333	3	0
spaCy	30,124	4,399	224	4	0
vllm	29,677	4,480	204	5	0
fastai	26,267	7,563	36	3	0
LLaVA	20,106	2,210	27	5	0
openui	19,185	1,769	380	3	0
gensim	15,652	4,374	136	4	0
argo-workflows	15,044	3,198	703	3	0
onnxruntime	14,618	2,921	3,833	3	0
DB-GPT	13,681	1,829	216	5	0
SWE-agent	13,644	1,383	48	5	0
langchainjs	12,636	2,166	281	5	0
tvm	11,761	3,469	1,065	3	0
dagster	11,638	1,465	1,802	3	0
TensorRT	10,762	2,130	2,032	3	0
danswer	10,576	1,319	186	4	0
OpenLLM	9,993	635	2	5	0
kedro	9,944	903	75	3	0
modin	9,871	651	99	3	0
sonnet	9,771	1,298	16	3	0
wandb	9,115	673	1,065	3	0
tokenizers	9,035	798	49	5	0
text-gen.-inference	9,005	1,059	176	5	0
pycaret	8,916	1,769	572	3	0
metaflow	8,223	772	93	3	0
catboost	8,077	1,188	12,418	3	0
cortex	8,020	607	103	3	0
autogluon	8,006	926	114	3	0
imaginAIry	7,948	441	68	4	0
GPTCache	7,211	502	21	5	0
BentoML	7,128	792	72	5	0
bitsandbytes	6,237	626	20	4	0
serving	6,180	2,189	73	3	0
flyte	5,752	655	578	2	0
serge	5,672	407	12	4	0
agentscope	5,177	316	58	3	0
llm-app	4,403	234	3	3	0
seldon-core	4,377	831	494	3	0
serve	4,221	862	103	2	0
kserve	3,586	1,060	1,470	4	0
polyaxon	3,567	314	32	3	0
hummingbird	3,352	279	16	3	0
keras-tuner	2,860	396	12	3	0
agents	2,801	722	96	3	0
promptbench	2,447	182	28	4	0
nvidia-container-toolkit	2,431	261	306	3	0
neural-compressor	2,218	256	535	3	0
sagemaker-python-sdk	2,104	1,137	305	3	0
gptq	1,922	153	7	5	0
spacy-transformers	1,349	165	3	5	0

Table 2. Vulnerabilities Root Causes found in the CVE analysis within CWE-1000.

CWE Pillar	#CWEs	#CVEs	Three Most Common CWEs for each Pillar
CWE-664 – Improper Control of a Resource Through its Lifetime	48	282	52× CWE-125 (Out-of-bounds read) 19× CWE-787 (Out-of-bounds write) 21× CWE-22 (Improper Limitation of a Pathname to a Restricted Directory)
CWE-707 – Improper Neutralization	11	150	66× CWE-20 (Improper Input Validation) 26× CWE-79 (Improper Neutralization of Input During Web Page Generation - XSS) 18× CWE-94 (Improper Control of Generation of Code - Code Injection)
CWE-682 – Incorrect Calculation	5	100	59× CWE-369 (Divide by Zero) 27× CWE-190 (Integer Overflow or Wraparound) 12× CWE-131 (Incorrect Calculation of Buffer Size)
CWE-703 – Improper Check or Handling of Exceptional Conditions	5	78	58× CWE-476 (NULL Pointer Dereference) 11× CWE-754 (Improper Check of Unusual or Exceptional Conditions) 5× CWE-755 (Improper Handling of Exceptional Conditions)
CWE-691 – Insufficient Control Flow Management	9	66	57× CWE-617 (Reachable Assertion) 2× CWE-670 (Always-Incorrect Control Flow Implementation) 2× CWE-835 (Loop with Unreachable Exit Condition – Infinite Loop)
CWE-710 – Improper Adherence to Coding Standards	7	66	58× CWE-476 (NULL Pointer Dereference) 2× CWE-475 (Undefined Behavior of Input to API) 2× CWE-798 (Use of Hard-coded Credentials)
CWE-284 – Improper Access Control	18	33	7× CWE-284 (Improper Access Control) 4× CWE-863 (Incorrect Authorization) 3× CWE-862 (Missing Authorization)
CWE-693 – Protection Mechanism Failure	6	13	6× CWE-352 (Cross-Site Request Forgery – CSRF) 2× CWE-345 (Insufficient Verification of Data Authenticity) 2× CWE-312 (Cleartext Storage of Sensitive Information)
CWE-697 – Incorrect Comparison	1	5	5× CWE-697 (Incorrect Comparison)
CWE-435 – Improper Interaction Between Multiple Correctly-Behaving Entities	0	0	–