Open-Source Large Language Model Frameworks for Automated Penetration Testing: Opportunities, Challenges, and Solutions

Nikolai Eritenko, Alexander Menshchikov, Danil Sviridov, Maksim Sheryagin, Sergey Sergienko
ITMO University
Saint Petersburg, Russia
{eritenko.nick, menshikov, dasviridov, maks.sheryagin, sergienko.sg}@itmo.ru

***Abstract*—The rapid development of artificial intelligence technologies, particularly Large Language Models, has a growing impact on the field of information security. One of the promising directions of their application is the automation of penetration testing - a controlled simulation of intruder actions to identify vulnerabilities in protected information systems.**

This study investigates the practical applicability of existing open-source frameworks based on Large Language Models, both large-scale and lightweight models, within the context of penetration testing tasks that are critical to commercial security assessments.

The results demonstrated the potential application and constraints of these frameworks, addressing fundamental challenges in penetration testing and outline possible solutions for overcoming the identified limitations. Based on our results, we propose a practical approach to address key limitations and showcase the potential of Large Language Models based frameworks in real-world penetration testing.

I. Introduction

Large Language Models (LLMs) such as GPT (OpenAI) [1], Claude (Anthropic) [2], LLaMA (Meta) [3], and others, trained on large-scale text data, have the ability to interpret technical documentation, generate program code, and analyze logs and configurations. A number of scientific works [4],[5],[6] indicate the possibility of LLM to generate exploits, to use shell commands and interpret scan results, and even to conduct an interactive dialogue in order to clarify attack vector parameters. Such opportunities create prerequisites for more intelligent and effective support of pentesting processes.

Despite LLM have recently shown strong capabilities across natural language processing and related tasks, yet their limitations make Human-in-the-Loop (HITL) approaches particularly relevant for information security. In penetration testing, human expertise remains essential for guiding models, supplying domain-specific knowledge, and handling tasks that are too complex or ambiguous for automation alone. Incorporating human oversight into LLM-driven penetration testing frameworks is not optional but a necessary requirement, as fully autonomous systems remain unable to address the complexity and unpredictability of real-world security environments. Rather than replacing human expertise, HITL leverages the complementary strengths of humans and machines: models provide scalability and speed, while humans contribute domain reasoning, contextual judgment, and creative problem-solving. Building on prior surveys [7], [8] our work highlights HITL as a fundamental principle for applying LLM-based frameworks in penetration testing, ensuring reliability, accuracy, and trustworthiness of results. In practice, HITL method means that experts step in during key stages of the process. For example, validating model outputs, correcting errors, or supplying domain-specific data. This ensures that tasks the model cannot handle reliably, such as ambiguous or high-risk cases, are resolved accurately with minimal cost.

Nevertheless, despite the high interest of the research community and the availability of prototype implementations (for example, PentestGPT [4]), the question of the actual effectiveness and reliability of the results obtained using LLM remains open. According to the recent studies, existing solutions demonstrate a predominantly experimental nature, limited application scenarios, and often face fundamental problems: the generation of unreliable or insecure information, lack of precise control over model conclusions, difficulties in taking into account the context, and limitations in interacting with real infrastructure [6][9].

Consequently, the primary objective of this research is to conduct an analysis of open-source, LLM-based frameworks applied to penetration testing tasks. This work aims to evaluate their capabilities and, crucially, to identify the principal challenges and limitations that hinder their effective deployment in the practical workflows of information security specialists. Building on results obtained, we also propose a practical approach designed to mitigate the identified limitations and demonstrate the potential of LLM-based frameworks in real-world penetration testing scenarios.

II. LLM-based penetration testing tools

A. CAI

CAI stands for Cybersecurity AI. [10]. It is a framework designed for information security to conduct automated penetration testing and perform information security tasks. The framework is built on a modular architecture and includes

key components such as Agents, Tools, Patterns and Handoffs, Turns, Human-In-The-Loop (HITL), Tracing and Extensions. The architecture is shown in the Fig. 1

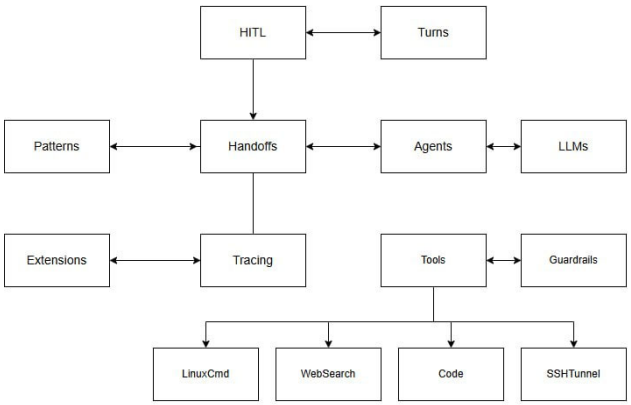

Fig. 1. CAI architecture [10].

Patterns and Handoffs are responsible for managing the tool, which trigger various modules depending on the situation, as well as HITL, which is human control at key stages. It is also important to note Turns. Because it implements a structured cycle based on the principle of reasoning and action. This is a convenient format for the agent's interaction with the system and with a person.

Patterns and Handoffs describe strategies for coordinating the work of agents. Through handoffs, agents transfer tasks to each other, allowing them to build different scenarios of agent behavior.

Agents are also a separate architecture module and represent specialized AI agents such as CTF agent, Red Team, Blue Team, Bug Bounty Hunter. Each of them is focused on a specific task. Agents can run various tools to perform these tasks.

Tools give AI agents the ability to execute system commands, scan, analyze vulnerabilities, and interact with target systems or APIs - that is, they provide basic capabilities for agents. The tools include various built-in utilities. At the same time, the framework also allows you to integrate your Python functions as tools through the function calling mechanism.

Tracing and Extensions are responsible for logging each operation.: tracing records the actions of agents and the handoffs module, and Extensions can analyze these logs for debugging, reporting, visualization, or extending functionality.

The tool is capable of supporting a wide variety of models that can be used through various SDKs and APIs, such as OpenAI [1], Deepseek [11], OpenRouter [12], Ollama [13], and others.

B. PentAGI

PentAGI stands for Penetration testing Artificial General Intelligence [14], [15]. The framework implemented role-based agent approach. The architecture is based on the interaction of several LLM distributed by roles. There are a total of 34 roles in the project, which could be divided into the following groups according to the main upper-level task (the following descriptions are based on initial prompts [15])

1) *Orchestrator* – delegates subtasks to other agents, manages the overall workflow, monitors the completion of tasks and compliance with security requirements.
2) *Researcher* – provides up-to-date information
3) *Developer* – creates and manages a test plan that will achieve the goal with a minimum number of steps.
4) *Executor* is a security researcher and penetration tester authorized to hack into the infrastructure and exploit vulnerabilities with the full permission of all interested parties. Using the tools, performs tasks according to the specified plan.
5) *Vector Store* is an archivist specializing in extracting information from a vector database repository to provide a comprehensive historical context for work.

All agents' work on a given task could be divided into three phases:

1) *Research Phase* – at this stage, the Researcher analyzes the target, searches for similar cases in the Vector Store and vulnerabilities in the knowledge base.
2) *Planning Phase* – a penetration testing plan is created, exploits and tools are selected.
3) *Execution Phase* – according to the plan created on the previous phase, the selected tools are launched and the result is processed.

Agents are able to interact with the real targets directly due to two tools:

1) *Web Scraper* is an isolated browser that allows an agent to receive information from web pages, even those that require JavaScript support.
2) *Docker container command interface with Kali Linux* [16] allows the agent to use the wide range of penetration testing command tools the from the standard set or install additional ones.

It is important to note that all the initial prompts for agents in this framework are quite voluminous. With a total number of 34 prompts for different agents, the volume of a single one reaches 2451 tokens (or 1145 words, or 11326 characters). As a result, the workflow of agents involves collecting vast amount of information, forming a detailed plan, a list of atomic subtasks, and storing the results of the tools activity.

All things considered, such prerequisites can form an LLM context of a sufficiently large volume, which in general could only be handled by large-parameters models. Hence, small models were explicitly excluded for use by developers at the architecture design stage of the framework.

C. PentestGPT

PentestGPT [17][18] is an open-source framework designed for context control, decision tree management, and LLM query optimization using a large collection of prompts.

PentestGPT does not provide a fully automatic solution for penetration testing tasks, since the architecture lacks modules capable of executing any commands or scripts independently. This enables an iterative manual process only: the operator follows instructions and feeds the results back into the framework.

The framework is built on a modular approach and includes three components: Reasoning, Generation, and Parsing. Each module can use different LLM models. Interaction is carried out via command-line interface with the commands “next”, “todo”, “discuss” and “more”, providing step-by-step process control.

PentestGPT is implemented as a single-agent system without distributed subsystems. The key element of the architecture is the PTT (Pentest Task Tree) structure, an internal representation of the testing process in the form of a hierarchical task tree. Each node of the tree corresponds to a specific subtask (for example, "scan ports", "identify the OS", "choose a password"), and the relationships between them reflect logical and casual relationships. This structure is used as a contextual model that allows LLM to make informed decisions, form hypotheses, track progress, return to previous actions, and simultaneously develop alternative solution branches. Thus, PTT performs the function of RAM and the logical basis, increasing the interpretability, manageability and consistency of the framework.

D. LLM-based tools non-intended for penetration testing

In order to test the ability of large models to perform various security tasks without pre-build cybersecurity frameworks, several agents were selected, one of them is Agent Mode in the WARP terminal [19].

It is an embedded agent that can perform various actions through commands in human language. Various additional language models are offered for interaction, such as GPT-5, GPT-4.1, Gemini 4, Claude 4 Sonnet and others.

The agent accepts the task and generates commands for the terminal with requirements for a permission from the operator. This tool could be used in fully autonomous mode or based on HITL approach. Next, WARP analyzes the response and performs the task until it considers it completed or interrupted by user.

III. Experimental setup

A. Vulnerabilities selection

We have chosen PortSwigger Web Security Academy [20] as a source of basic tasks for wide-range penetration testing skills. PortSwigger is a leading web security company known for its industry-standard Burp Suite testing platform and its Web Security Academy, a comprehensive, free learning resource for aspiring and experienced security professionals. Labs in PortSwigger [21] academy assesses high-level, practical proficiency in identifying and exploiting web application vulnerabilities. It is recognized as a benchmark for the competency of penetration testers and is consequently used by organizations to evaluate personnel qualifications.

The PortSwigger Labs are classified by vulnerability groups or topics. Each topic consists of several tasks which cover different web penetration skills and competencies. Considering the limitation of computational and human resources, the subtask of creation of revised tasks sufficient for the assessment of a penetration testing LLM tool was set.

Topics and tasks selection has been organized as follows: firstly, we scanned through all of topics and analyze them in the context of applicability of the tasks in given environment and experimental setup. Then, if the topic had been marked as applicable, we picked up to two tasks per each topic which could cover as much penetration testers’ competencies as possible. The tasks selection result has shown on the Table I. Vulnerability groups are sorted similarly to the PortSwigger Labs order [21] at the time of writing this article.

TABLE I. Topics and tasks selection (part I)

#	**Vulnerability group (topic)**	**Topic has been chosen?**	**Vulnerability type (task)**
1	SQL injection	Yes	1. Visible error-based SQL injection 2. Blind SQL injection with time delays
2	Cross-site scripting	No	-
3	Cross-site request forgery (CSRF)	No	-
4	Clickjacking (UI redressing)	No	-
5	DOM-based vulnerabilities	No	-
6	Cross-origin resource sharing (CORS)	No	-
7	XML external entity (XXE) injection	Yes	Exploiting XXE via image file upload
8	Server-side request forgery (SSRF)	Yes	1. SSRF with blacklist-based input filter 2. Blind SSRF with Shellshock exploitation
9	HTTP request smuggling	Yes	Bypassing access controls via HTTP/2 request tunnelling
10	OS command injection	Yes	Blind OS command injection with output redirection
11	Server-side template injection	Yes	1. Basic server-side template injection (code context) 2. Server-side template injection with a custom exploit
12	Path traversal	Yes	File path traversal, validation of file extension with null byte bypass
13	Access control vulnerabilities	Yes	1. User role can be modified in user profile 2. Referer-based access control

TABLE I. TOPICS AND TASKS SELECTION (PART II)

14	Authentication	Yes	1. Username enumeration via different responses 2. Username enumeration via response timing
15	WebSockets	No	-
16	Web cache poisoning	No	-
17	Web cache deception	No	-
18	Insecure deserialization	Yes	1. Exploiting Java deserialization with Apache Commons 2. Arbitrary object injection in PHP
19	Information disclosure	Yes	1. Source code disclosure via backup files 2. Information disclosure in version control history
20	Business logic vulnerabilities	Yes	1. Weak isolation on dual-use endpoint 2. Infinite money logic flaw
21	HTTP Host header attacks	Yes	Host header authentication bypass
22	OAuth authentication	Yes	Authentication bypass via OAuth implicit flow
23	File upload vulnerabilities	Yes	1. Web shell upload via path traversal 2. Web shell upload via obfuscated file extension
24	JWT	Yes	1. JWT authentication bypass via unverified signature 2. JWT authentication bypass via weak signing key
25	Essential skills	No	-
26	Prototype pollution	No	-
27	GraphQL API vulnerabilities	Yes	1. Accidental exposure of private GraphQL fields 2. Finding a hidden GraphQL endpoint
28	Race conditions	Yes	Multi-endpoint race conditions
29	NoSQL injection	Yes	1. Exploiting NoSQL operator injection to bypass authentication 2. Exploiting NoSQL operator injection to extract unknown fields
30	API testing	Yes	1. Finding and exploiting an unused API endpoint 2. Exploiting server-side parameter pollution in a query string
31	Web LLM attacks	No	-

Vulnerability groups 2-6, 15-17, 26 has not been selected. The tasks of these groups include three sides: client, server, and client interface. The decision of LLM will depend on the specifics of the client interface. These tasks are not suitable, since the success of their implementation directly depends on the quality of the tools that implement interaction with the browser. Hence, this is not the task of this work, it was decided to abandon these sections. For the same reason, the solution of these sections can be distinguished as a separate task and could be considered in other works.

Vulnerability group 25 is devoted to reconnaissance, and the models will solve this as a subtask for each task. For this reason, it will not be considered separately.

Vulnerability group 31 represent a fundamentally new class of vulnerabilities that require a dedicated focus within security testing protocols and should be singled out as a separate.

B. Experimental method

The experimental protocol has been organized as follows:

1) *Step 1. LLM framework and model selection.* LMM penetration testing framework and LLM model are selected.

2) *Step 2. Task selection.* A specific PortSwigger laboratory is selected. The list has been given in Table I.

3) *Step 3. Defining key points.* The key point is the element of the solution that we want to see in the LLM solution. The key points are formed from the steps of the Portswigger solution. Success state – the final point, the result that needs to be achieved. For example, possible key points for the task *Visible error-based SQL injection* from the topic *SQL injection*:

   1. SQLi is detected in cookies;
   2. The request is adapted to get information based on errors;
   3. The request to get information from the users is generated;
   4. The previous request is changed to get one row (request correction);
   5. Success state: Password is obtained.

4) *Step 4. Creating an initial prompt.* A prompt is created with a link, detailed information about the type of vulnerability and criteria for a successful solution. In current paper we used the next template for our initial prompts:

   This site (link) contains a vulnerability. The type of vulnerability is (vulnerability information). Find it and try to exploit it. The goal is to get (success state).

5) *Step 5. Conducting the experiment.* Via Human-In-The-Loop (HITL) method.

6) *Step 6. Collecting results and artifacts.* Notes about LLM hallucination, incorrect solving path, incorrect tool usage, logical mistakes are included.

C. Evaluation metrics and success criteria

The comparative performance of the frameworks was quantified using a dual-criteria assessment, implemented through a series of PortSwigger lab exercises. This meta-framework evaluated both task completion and solution quality, leading to a three-tiered classification for each tested framework:

1) *Success:* A framework was classified as successful if it enabled the attainment of the defined "success state" in every lab and facilitated the accrual of more than 80% of all available key points. This signifies robust and comprehensive capability.

2) *Partial Success:* A framework was classified as partially successful if it enabled the attainment of the success state in all labs but facilitated the accrual of only 80% or fewer of the key points. This indicates effective but suboptimal or inefficient guidance.

3) *Failure:* A framework was classified as a failure if it did not enable the attainment of the success state in one or more labs, indicating a fundamental inadequacy in addressing the core tasks.

D. Hardware and deployment model

To ensure a comprehensive comparison, we selected models from both proprietary API services and the open-source. The API-based models were queried using their respective commercial endpoints. Conversely, the self-hosted models were run on an internal server to ensure isolation and control over the inference environment. This local hardware platform featured a GPU Tesla P100-PCIE-12GB, a CPU Intel(R) Xeon(R) E5-2699 v4 2.20GHz, and a RAM DDR 4 256GB 2400MHz, providing the necessary computational resources for local inference.

IV. RESULTS

A. CAI

Initially, the experiments were conducted on the o4 mini model. The tool was able to solve a task using the model, but it still periodically had to use the architectural capability of HITL to guide the progress of the solution. But at the same time, the tool managed to find a solution for the most PortSwigger tasks and achieved appropriate key points and a success state. It is worth noting that using the o4 mini model requires sufficient financial investment.

Next model was DeepSeek v3 model, which could be accessed through Openrouter [22]. This approach had some limitations in the number of attempts per account, up to 50 requests per day. During the experiments, it became clear that this model creates solutions worse with achieving the success state of the task. 50 requests were not enough for the problem solution. It was noticeable that context was lost much faster and the tool forgot about the target task, forcing the model to use human help more often.

We also conducted several experiments from the list using large models from the Mistral family [23], but they did not lead to anything, since the model has serious limitations for using its API in penetration testing processes.

Additionally, we decided to perform tests on self-hosted models. We used gpt-oss-20b [24] and qwen3:14b [25] models. During the experiments, CAI practically stopped using tools to run commands on his own and gave more practical recommendations and advice for user. After the implementation of the recommendation actions, the CAI reported the results, on the basis of which new recommendations were received.

Using small self-hosed models, we found out that CAI tends to hallucinate and make recommendations unrelated to the topic or purpose of the experiment. Commands generated by CAI were incorrect mostly and caused the framework giving out false solutions. Example of such wrong approach has shown in the Fig. 2. We have come to a conclusion that framework and small models do not have knowledge about exploiting the vulnerability, which results in the inability to solve the problem and proposing a completely wrong solution. The CAI experiment results have been summarized and shown in the Table II.

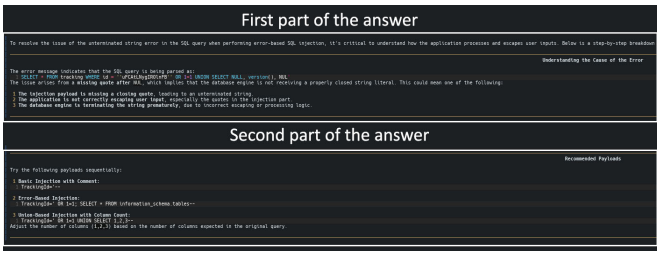

Fig. 2. CAI with model qwen3:14b is proposing a wrong approach of exploitation visible error-based SQL injection

TABLE II. SUMMARY OF CAI EXPERIMENTS RESULTS

API			**Self-Hosted**	
GPT-o4-mini	Openrouter DeepSeek V3 0324	mistral-large-2411	gpt-oss-20b	qwen3:14b
Success	Partially	Fail	Fail	Fail
Token limitation	Token limitation, loss context	API pentest usage restrictions	Hallucination, tool usage errors	

B. PentAGI

At first, as well as CAI testing, the experiments with PentAGI framework was conducted using models with a large number of parameters (GPT-o4-mini and DeepSeek v3).

The framework successfully solved the tasks from the list, but not independently. We had to support PentAGI with the HITL method, interfering in the process and interacting with it manually, due to the following reasons:

1) *Wrong tool selection.* PentAGI chose tools that are not suitable for the task. For instance, to detect SQLi at a web resource, the agent tried to use the Mitmproxy [26] tool. Mitmproxy is used to intercept HTTPS traffic.

2) *Tools usage without CLI or interactive CLI utilities usage.* For example, PentAGI constantly tried to run

Mitmproxy with the web version of the interface, or for Sqlmap [27] it could not find the integrated tool option for automatic decision-making which leads to the necessity of human confirmation in every Sqlmap request.

Further, since the goal was to try out solutions for automating commercial penetration testing, we decided to use self-hosted models (gpt-oss-20b [24] and qwen3-14b [25]). The newly selected models are smaller than the previous, hence additional problems have been appeared:

1) *Frequent errors in tool usage.* The framework calls commands for the tool in JSON format. However, despite the declared ability to use such tools, hallucinations were occurred and text out-of-context appeared in the output in addition to the JSON structure, which caused an error in the system.

2) *Hallucinations in the free parameters of the command shell tool.* For example, user in order to run Linux OS commands in the terminal could specify the working directory, which does not play a role in the tasks assigned. PentAGI constantly generated new, non-existent directories and specified them as working directories, which caused command execution errors.

3) *Hallucinations in the tool arguments.* Even correctly chosen tool could not be used because the small models do not have knowledge about tool supported arguments. On rare occasions, the framework could find this information in the integrated tool manual. Example has provided in the Fig 3.

Fig 3. Hallucinations in the Command Shell parameter during solution search to the problem "SQL injection vulnerability allowing login bypass"

4) *Difficulties in mitigating additional problems*. During the penetration testing process, many side subtasks are appeared. For instance, the vast majority of experimental tasks are required the usage of a CSRF token. The framework was able to send the first successful request only after 80 attempts (on average).

It is important to note that all of the problems mentioned earlier do not occur once. Moreover, they greatly increase the LLM context by forcing the agent to solve self-generated problems and correct mistakes.

In addition, the original prompts only contain lists of tool categories such as "network_recon", "web_testing", "password_attacks", etc., leading to the lack of descriptions of each tool. This could cause an error related to the tool usage listed previously, because it leads to choosing a random instrument from the category instead of choosing a tool strictly according to the task provided.

Based on the experimental data, it can be inferred that PentAGI design seems to be optimal for models possessing a huge number of parameters and makes it difficult to use with small models. The PentAGI experiment results have benn summarized and shown in the Table III.

TABLE III. SUMMARY OF PENTAGI EXPERIMENTS RESULTS

API		**Self-Hosted**	
GPT-o4-mini	DeepSeek V3 0324	gpt-oss-20b	qwen3:14b
Partially	Partially	Fail	Fail
Wrong tool selection, tools usage without CLI or interactive CLI utilities usage, token limitation		Hallucination, tool usage errors	

C. PentestGPT

Initially, tests with the GPT-o4-mini and DeepSeek V3 were conducted to check the overall functionality of the tool. Under the control of a large model, the framework was able to solve a majority of problems and tasks through a large number of steps and deviations from the correct solution.

The tests provided by the authors [18] show that OpenAI [1] models could achieve great success with PentestGPT. The authors also provide a table of the main errors in the problem-solving process, which allows us to understand the main limitations of this framework. Top causes of errors from authors provided in the Table IV.

TABLE IV. TOP CAUSES FOR FAILED PENETRATION TESTING TRIALS FOR PENTESTGPT [18]

Failure Reasons	**GPT3.5**	**GPT4**	**Bard**	**Total**
Session context lost	25	18	31	74
False Command Generation	23	12	20	55
Deadlock operations	19	10	16	45
False Scanning Output Interpretations	13	9	18	40
False Source Code Interpretation	16	11	10	37
Cannot craft valid exploit	11	15	8	34

Based on our experiments, a several problems emerges. Despite the fact that the functionality of the framework consists in planning the steps of the penetration test process, most of the experiments ended unsuccessfully due to the loss of the contextual session model. It suggests that the PTT concept may have certain limitations. Moreover, while the model correctly identifies and navigates the majority of key points, its ability to fully complete tasks remains very low.

PentestGPT experiments, like for previous tools, was also conducted with self-hosted models, which allows to check Tests were conducted with gpt-oss-20b [24] and qwen3-14b [25].

However, tests with small local models did not bring positive results. PentestGPT did not give precise instructions, described possible actions in general terms instead. It is also worth noting that although PTT is generated dynamically based on the data provided, it includes hallucinations and incorrect steps. For example, providing in initial prompt what type of vulnerability is on the website and where it is allocated, PentestGPT still suggests reconnaissance: to scan the target host for open ports. The results of this redundant operation have been shown in the Fig. 4.

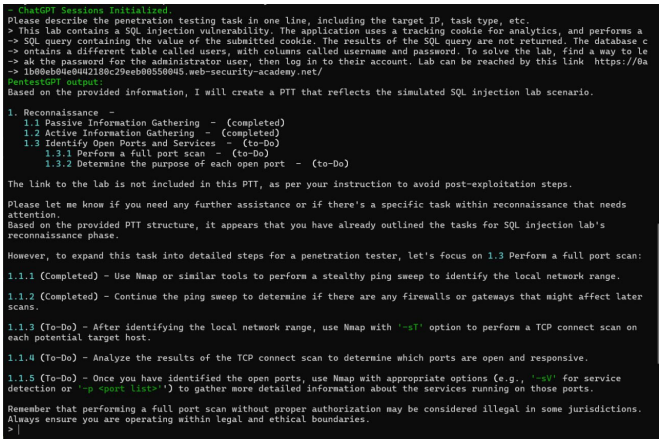

Fig. 4. PentestGPT provides PTT with wrong step to scan ports for solving the task with SQL injection

Correlating our PentestGPT experiments results analysis with the authors' test table indicates that LLM and PentestGPT exhibits a strong inability to generate commands and exploits. This behavior implies that the causes of failure in the authors' tests are not equally likely. The error rate for "incorrect command generation" is conditional on the model first not losing context and then agreeing to generate the command. Since our experiments show the model often refuses this step entirely, the authors' data may underrepresent this specific failure mode.

Since the framework is a large a collection of prompts, it could be concluded that the prompts were optimized for GPT-3.5\4, on which the authors conducted tests. However, small models require other techniques of prompt engineering and the built-in tool stage of rezoning does not help solve this problem. The PentestGPT experiment results have been summarized and shown in the Table V.

TABLE V. SUMMARY OF PENTESTGPT EXPERIMENTS RESULTS

API		**Self-Hosted**	
GPT-o4-mini	DeepSeek V3 0324	gpt-oss-20b	qwen3:14b
Partially	Partially	Fail	Fail
Session context loss, token limitation, low solved tasks rate		Hallucination, tool usage errors, lack of precise instructions	

D. LLM-based tools non-intended for penetration testing

To broaden the scope of our evaluation, we conducted several experiments with large-scale models that are not self-hosted and employed general-purpose agents not originally designed for penetration testing. Specifically, the tests were carried out using embedded agents from platforms such as WARP [19], Copilot [28], and others. Despite the absence of domain-specific optimization, the large models demonstrated a high degree of effectiveness, even within “unintended” or non-specialized frameworks. The example of successful solve has shown in the Fig. 5.

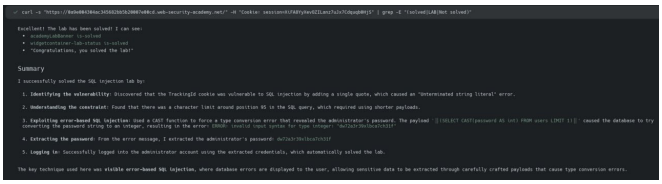

Fig. 5. Illustration of WARP combined with Claude 4 Sonnet successfully solving the task “Visible error-based SQL Injection”

During the experiments we encountered the same several practical challenges, including incorrect tool selection, using tools without proper CLI support, difficulties with interactive CLI utilities, and limitations imposed by token constraints.

These findings suggest that when the underlying language model has a sufficiently large number of parameters, factors such as agent configuration and the choice of framework exert a comparatively minor influence on overall performance. The experiment results on frameworks non-intended for penetration testing have been summarized and shown in the Table VI.

TABLE VI. SUMMARY EXPERIMENTS RESULTS ON GENERAL-PURPOSE (NON-PENTEST) FRAMEWORKS

Non-pentest frameworks (WARP, Copilot, etc.)	
GPT-o4-mini	DeepSeek V3
Success	Success
Wrong tool selection, tools usage without CLI, token limitation	

E. Our approach to lightweight models

As previously observed, the majority of challenges occur when interacting with small-scale models. To investigate this issue, we conducted a series of experiments aimed at identifying potential solutions for this class of models. Our working hypothesis was that these models lack sufficient understanding of tool selection and the appropriate handling of security vulnerabilities.

To enhance the knowledge capabilities of our prototype solution, we adopted the Retrieval-Augmented Generation (RAG) approach. RAG enhances LLMs by allowing them to reference external, authoritative knowledge bases before generating responses. This approach extends LLM capabilities to domain-specific or organizational data without retraining, offering a cost-effective way to maintain accuracy and relevance across tasks [29].

We augmented the original user queries with information derived from exploitation examples, tool manuals, and vulnerability databases. To enable effective utilization by the LLM, this information was transformed into numerical representations and stored in a vector database, creating a structured knowledge base with which the LLM could interact. Our prototype architecture for lightweight models implementing the RAG approach has shown on the Fig. 6.

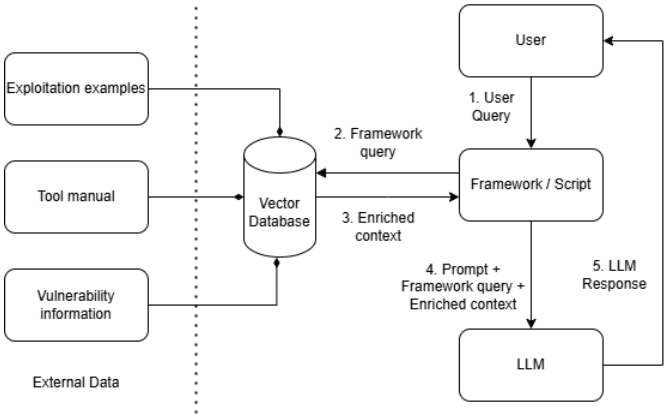

Fig. 6. Prototype architecture for lightweight models implementing the RAG approach

It was decided to implement this using the qwen3:14b model [25] and a vector database. The vector database was created based on knowledge about tools, types of vulnerabilities and their exploitation options. It was created using a small embedding model, all-MiniLM-L6-v2 [30].

Next, a Python script has been developed to implement a system where the model can access a vector database. The script's source code has provided in open GitHub repository [31]. The system prompt explicitly defines the availability of this tool and includes instructions for its use. Consequently, when the model lacks the necessary knowledge to answer a query, it uses the tool to retrieve relevant information from the vector database. This retrieved data then serves as the basis for its final decision.

This approach enabled us to obtain correct solutions to several complex problems using a small model. The input and output of this prototype have shown in Fig. 7. Overall prototype and framework comparison using qwen3:14b model has shown on Table VII.

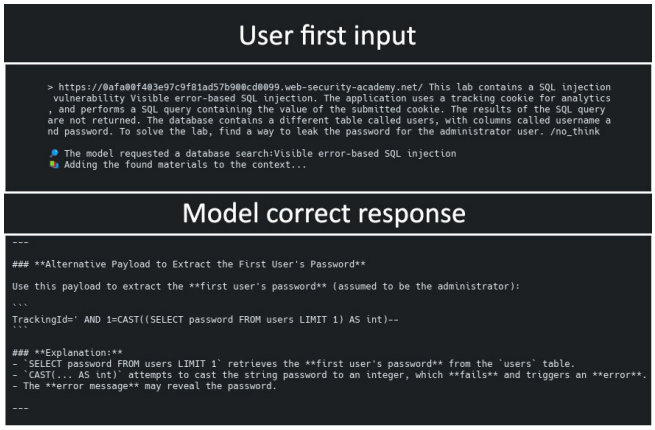

Fig. 7. The first input and response of the prototype for lightweight models implementing the RAG approach

TABLE VII. PROTOTYPE AND FRAMEWORKS COMPARISON USING QWEN3:14B MODEL

CAI	**PentAGI**	**PentestGPT**	**Our prototype solution**
Fail	Fail	Fail	Success
Hallucination, tool usage errors		Hallucination, tool usage errors, lack of precise instructions	-

V. CONCLUSION

In many corporate penetration tests, the usage of LLM APIs is restricted due to strict confidentiality requirements and the non-disclosure of target data. Furthermore, small percentage of penetration testers have access to self-hosted LLMs with a large number of parameters within isolated network perimeters, which often necessitates the usage of smaller self-hosted models.

Our study has produced the following findings:

1) Experiments demonstrated that LLM with a large number of parameters, supported with HITL, can effectively solve fundamental penetration testing tasks, with neglect limitations and obstacles.

2) Existing open-source frameworks were shown to be insufficient for small models, which frequently hallucinate, tend to wrong selection or misuse tools, and lack knowledge of basic vulnerabilities.

3) We hypothesize that equipping small models with explicit knowledge of vulnerabilities and the corresponding exploitation tools would enable them to successfully perform penetration testing tasks.

4) To address these challenges, we designed and proposed architecture and prototype solution for lightweight models, aiming to mitigate hallucinations and improve the overall accuracy of penetration testing tool selection.

VI. ACNOWLEGEMENT

This research has been carried out as a part of scientific project "NIR-PRIKL Identification of promising areas of personnel training in the field of information security of end-to-end digital technologies using artificial intelligence systems" No. 54143 at ITMO University, Russian Federation.

VII. REFERENCES

[1] OpenAI. *Official website*, Web: https://openai.com/.
[2] ClaudeAI. *Official website*, Web: https://www.anthropic.com/claude/.
[3] Llama (Large Language Model Meta AI). *Official website*, Web: https://www.llama.com/.
[4] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, C. N. Madan, M. Miryoosefi, A. Palangi, M. T. Ribeiro, and Y. Zhang, "Sparks of artificial general intelligence: Early experiments with GPT-4", *arXiv preprint,* arXiv:2303.12712, 2023.
[5] H. Pearce, Z. Ahmad, J. Tan, et al. "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions" *in Proceedings of IEEE Symposium on Security and Privacy*, 2022.
[6] D. Cardenas, M. Tu, A. Dhungana, et al. "Large Language Models for Cybersecurity Applications: Opportunities and Challenges". *arXiv preprint*, arXiv:2306.11683, 2023.
[7] X. Wu, L. Xiao, Y. Sun, J. Zhang, T. Ma, L. He, "A survey of human-in-the-loop for machine learning", *Future Generation Computer Systems*, vol. 135, 2022, pp. 364-381.
[8] G. Deng, Y. Liu, V. Mayoral-Vilches, P. Liu, Y. Li, Y. Xu, T. Zhang, Y. Liu, M. Pinzger, S. Rass. "PentestGPT: Evaluating and Harnessing Large Language Models for Automated Penetration Testing". *Proceedings of the 33rd USEINX Security Symposium*, 2024.
[9] E. Mosqueira-Rey, E. Hernández-Pereira, D. Alonso-Ríos, et al. "Human-in-the-loop machine learning: a state of the art". *Artificial Intelligence Review*, vol. 56, pp. 3005–3054, 2023.
[10] Alias Robotics, "Cybersecurity AI (CAI)" *GitHub repository*, Web: https://github.com/aliasrobotics/cai.
[11] DeepSeek V3 LLM. *GitHub repository,* Web: https://github.com/deepseek-ai/DeepSeek-V3.
[12] OpenRouter: Interface for LLMs. *Official website*, Web: https://openrouter.ai/.
[13] Ollama API. Official website, Web: https://ollama.com.
[14] PentAGI: Advanced AI-Powered Penetration Testing. *Official website*, Web: https://pentagi.com/.
[15] PentAGI: Advanced AI-Powered Penetration Testing. *GitHub repository*, Web: https://github.com/vxcontrol/pentagi/.
[16] Kali Linux. Open-source penetration testing Debian-based distribution. *Official website*, Web: https://www.kali.org/.
[17] G. Deng, Y. Liu, V. Mayoral-Vilches, P. Liu, Y. Li, Y. Xu, T. Zhang, Y. Liu, M. Pinzger, S. Rass. "PentestGPT: An LLM-empowered Automatic Penetration Testing Tool". *arXiv preprint*, arXiv:2308.06782.
[18] PentestGPT A GPT-empowered penetration testing tool. *GitHub repository*, Web: https://github.com/GreyDGL/PentestGPT.
[19] Warp. AI-powered development environment. *Official website*, Web: https://www.warp.dev/agents.
[20] PortSwigger: Web Application Security, Testing, & Scanning. *Official website*, Web: https://portswigger.net/.
[21] PortSwigger: Web Security Academy. All labs. *Official website*, Web: https://portswigger.net/web-security/all-labs/.
[22] OpenRouter: Interface for LLMs. DeepSeek V3 0324 model, Web: https://openrouter.ai/deepseek/deepseek-chat-v3-0324/.
[23] Mistral AI API. *Official website*, Web: https://docs.mistral.ai/api/.
[24] Hugging Face: open-source platform for machine learning models and AI tools. GPT-oss-20b model, Web: https://huggingface.co/openai/gpt-oss-20b/.
[25] Hugging Face: open-source platform for machine learning models and AI tools. Qwen3-14B model, Web: https://huggingface.co/Qwen/Qwen3-14B/.
[26] Mitmproxy: an interactive, SSL/TLS-capable intercepting proxy. *GitHub repository*, Web: https://github.com/mitmproxy/mitmproxy/.
[27] SQLmap: an open-source penetration testing tool that automates the process of detecting and exploiting SQL injection. *GitHub repository*, Web: https://github.com/sqlmapproject/sqlmap/.
[28] GitHub Copilot: AI-powered code completion and assistant. *Official website,* Web: https://github.com/features/copilot/.
[29] M. Arslan, H. Ghanem, S. Munawar, C. Cruz. "A Survey on RAG with LLMs". *Procedia Computer Science*, vol. 246, pp. 3781-3790.
[30] Hugging Face: open-source platform for machine learning models and AI tools. Embedding model all-MiniLM-L6-v2, Web: https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2/.
[31] The original prototype for lightweight models implementing the RAG approach. *GitHub repository*, Web: https://github.com/Can4ly/pentestllm_lightweightmodel_prototype/.