GLITCH_AI: Autonomous Pentesting with LLM-Powered Strategy and Reporting

Maria Gonzalez Herrero

2401205

A thesis submitted for the degree of Master of Science in Computer Networks and Security

Supervisor: Dr. Amit Singh

School of Computer Science and Electronic Engineering

University of Essex

August 2025

Abstract

Cybersecurity faces increasing challenges due to the growing complexity of attack platforms and the limitations of traditional penetration testing workflows, which are often manual, slow, and dependent on specialised knowledge. Existing automated tools offer partial solutions, but tend to be rigid, isolated, and rarely incorporate advanced reasoning or contextual adaptation.

This thesis introduces **Glitch_ai**, an automated penetration testing framework that combines established security tools with large language models (LLMs) to enhance adaptability, interpretability, and usability. The framework uses a hybrid architecture consisting of a **FastAPI backend**, a dynamic **web interface**, and **role-based** execution logic. Analyses are conducted either locally or via a secure remote backend to maintain compatibility in restricted environments. The framework coordinates nine widely used penetration testing tools, such as Nmap, Nikto, SQLmap, Hydra, Sublist3r, Whois, Dmitry, WPScan, and WhatWeb. Specialised variants of the **Mistral** large language model, including 7B, 3B, Nemo, and CodeStral, support task-specific reasoning, error handling, and natural language reporting.

The evaluation was conducted through unit, integration, and validation testing on intentionally vulnerable targets. The results show that Glitch_ai automates multi-tool workflows, reduces execution errors through adaptive retry logic, and generates reports tailored to both technical and non-technical audiences. The hybrid local-remote design also allows for comprehensive testing without elevated privileges.

List of Illustrations

List of Tables

Contents

1. Introduction

This chapter explains why advanced cybersecurity automation tools are needed. It gives an overview of the problem the project addresses, describes its main objectives, and clearly defines the scope of the work.

1.1. Context and Motivation

The field of cybersecurity continues to evolve rapidly in response to increasingly sophisticated threats and an ever-expanding attack surface. As digital transformation accelerates across all industries, penetration testing remains a crucial method for evaluating a security posture. However, traditional workflows are often lengthy, highly manual, and require significant technical expertise.

Small teams and organisations with limited resources often struggle with evolving vulnerabilities and complex environments. At the same time, automation and artificial intelligence are advancing quickly, strongly impacting security operations.

Recent progress in AI, particularly in **large language models (LLMs)**, has made it easier to manage information, gain valuable insights, and support informed decision-making in technical fields. Still, most current tools only improve certain tasks. They are often isolated, inflexible, or not truly adaptable.

This project was started to explore how LLMs can be used in penetration testing. The use is not limited to summaries, but also extends to guiding actions, interpreting results, and suggesting ways to resolve problems. The main aim is to combine manual testing skills with the benefits of smart automation.

1.2. Problem Statement

Although more **automated security tools** exist, penetration testing workflows still face major limits. These limits hinder effectiveness and scalability. Most current solutions run in fixed channels, offer limited integration with command-line tools, and cannot adapt to each target or the results of earlier steps.

Furthermore, the integration of artificial intelligence into cybersecurity tools has largely focused on classification tasks or simple alert generation, rather than deeper reasoning, decision-making, or coordination. Large language models (LLMs), while increasingly adopted in other domains, remain underutilised in modular, real-time penetration testing environments.

Due to this gap, teams still rely on manual work, encounter many false positives, and struggle to interpret tool results or generate useful advice. This makes the job harder for security professionals and limits the ease with which good testing methods can be repeated or scaled up.

1.3. Objectives

The main goal of this project is to develop an automated penetration testing tool. It combines standard security methods with LLM features. This should increase flexibility in testing, improve reasoning, and make reporting clearer.

To achieve this, the project introduces **Glitch_ai**, a **modular system**. It combines local and remote command execution with natural language analysis, automates tasks such as vulnerability scanning and enumeration, adapts execution paths based on real-time results and provides natural language mitigation reports that both technical and non-technical users can access.

The system also serves users with different privilege levels. Normal users can perform unprivileged scans with minimal configuration. Superusers have access to advanced features such as tool installation or the use of commands that need elevated permissions.

The tool combines key functions into a single system and manages command-line tools through a unified interface. It uses LLMs to interpret and summarize results. The tool can run security tools locally or remotely, producing results that are useful for both technical and strategic needs.

1.4. Scope and Limitations

This project presents the design and implementation of Glitch_ai, a modular and automated penetration testing system. It integrates traditional security tools with large language models (LLMs) to enable dynamic orchestration, contextual reasoning, and natural language reporting. The tool tries to bridge the gap between static rule-based scanners and more adaptive, AI-enhanced cybersecurity workflows.

The main scope of this work includes:

- **Integration of nine core security tools**: Nikto, SQLmap, Hydra, Nmap, Whois, Dmitry, Sublist3r, WPScan, and WhatWeb, covering multiple attack surfaces (e.g., reconnaissance, scanning, brute force, and CMS analysis).
- **Modular backend** using FastAPI that supports command construction, validation, execution (both local and remote), and adaptive error handling.
- **Frontend web interface** that enables users to authenticate, select tools, monitor progress, and generate reports in PDF and JSON formats.
- **Role-based access control**, with support for both standard users (U) and superusers (S). Tool availability and UI behavior adapt accordingly.
- **LLM integration with Mistral 3B, Mistral 7B, Mistral Nemo, and CodeStral**, selected based on task requirements (e.g., summarization, code correction, adaptive retry, or mitigation explanation).
- **Remote execution support via virtual machine**: when tools are missing or sudo access is not available locally, commands are offloaded to a secure backend environment to ensure full functionality.
- **Automatic target validation and a conditional tool enabling ensuring** that only suitable tools and options are available. This depends on the input format and protocol.

Despite the system's wide functional coverage, several limitations remain:

- The **scope is currently limited to web-based targets**. While the underlying architecture is extensible, the current implementation primarily focuses on reconnaissance and vulnerability scanning for websites and IP addresses.
- **LLM reasoning is not yet real-time**, and some decisions (e.g., adaptive retries or fallback tool selection) require multiple failed attempts before suggestions are triggered.
- **No support for authenticated web app testing** (e.g., login brute force with session handling, token management, etc.) has been implemented at this stage.
- **Tool updates and vulnerability databases** must be managed manually in the current version; live syncing or CVE feed parsing is left for future work.

Overall, these limitations help keep the system practical, secure, and testable for a master's thesis. However, the platform's flexible design allows for future improvements to be added.

2. Literature Review

This chapter provides the conceptual and technical foundations that underpin the development of Glitch_ai. It reviews traditional penetration testing tools, examines the limitations of current automated solutions, and explores the role of artificial intelligence, particularly LLMs, in cybersecurity. The chapter concludes with a comparative analysis of existing AI-based tools.

2.1. Traditional Pentesting Tools

Penetration testing, or **pentesting**, is a well-established practice in cybersecurity that aims to assess the security posture of an information system by simulating real attack scenarios. The process involves methodically probing systems, networks, and applications to uncover exploitable vulnerabilities before malicious attackers do. Traditional penetration testing typically follows a structured methodology that includes phases such as reconnaissance, scanning, enumeration, exploitation, and post-exploitation [1] [2].

Over the years, a wide range of tools has been developed to help professionals execute these phases. For example, Nmap and Dmitry are widely used for network discovery and service identification, while tools such as Nikto, SQLmap, and Hydra are used for web vulnerability analysis, SQL injection testing, and brute force authentication attacks, respectively [3] [4].

However, the sheer volume and diversity of tools available can pose challenges. As Phong points out, **selecting optimal combination** of utilities for a specific test can be overwhelming, particularly given the overlap in functionalities and inconsistent performance across different environments [4]. Additionally, many tools demand advanced technical expertise for configuration and interpretation, which limits their accessibility to non-specialized teams [5].

Despite these limitations, traditional penetration testing remains a critical component of offensive security assessments. It provides detailed insights into system vulnerabilities and supports the **validation of security controls**. The dependence on manual expertise and static methodologies, however, highlights the necessity for more adaptable and intelligent systems. AI-enhanced tools, including Glitch_ai, are developed to address these requirements.

2.2. Limitations of Existing Automated Tools

Automated security tools have improved the speed and scalability of vulnerability assessments. Nevertheless, these tools demonstrate several critical limitations that diminish their effectiveness in practical penetration testing workflows. The primary limitations are categorized as follows:

- Rigid execution channels

Most tools follow predefined analysis workflows that users cannot modify or restructure. This rigid structure prevents adaptation to specific environments or scenarios that require tool chaining or conditional execution. For example, Detectify and Bright Security offer valuable analyses, but do not support dynamic tool orchestration or logical branching during runtime [6] [7].

- Limited local integration and CLI support

A significant number of commercial solutions operate as cloud-based platforms, providing little or no integration with local command-line tools. This limits their ability to run custom analyses or interact with existing scripts or tools on the analyst's machine. Tools such as ImmuniWeb and Acunetix are powerful, but they lack support for integrating custom CLI workflows or parsing outputs from local tools [8] [9].

- Lack of contextual understanding and adaptive logic

Most tools lack the ability to conditionally adjust execution based on previous scan outcomes or errors. If an analysis fails or produces ambiguous data, there is no mechanism to retry with alternative logic. Invicti and *Nuclei*, for example, work well in structured tests but do not adapt to context or unexpected results [10] [11].

- Weak Natural Language Reporting

Although some tools generate summaries or reports in PDF format, few offer legible, practical explanations in natural language that are tailored to both technical and non-technical users. Most displays raw results or CVE references without providing guidance on how to mitigate them. OWASP ZAP and Nuclei, despite being widely used, rely on this static approach [12] [11].

- AI Usage is Limited or Narrow

Even in tools that claim to use AI or machine learning, it is often limited to specific tasks (e.g., risk scoring or reducing false positives). Large language models are rarely integrated for deeper reasoning, adaptive decision-making, or natural language interaction. For example, Bright Security uses machine learning for CI/CD feedback but lacks broader AI-driven adaptability [7].

Table 2.2.1 provides a comparative overview of selected tools, evaluating key features such as execution control, AI and large language model (LLM) integration, integration capabilities, and report quality.

Tool	AI/LLM Integration	Execution Flexibility	Custom Tool Integration	Natural Language Reports	Adaptive Retry Logic	Deployment Mode
Detectify	Limited (CVE AI)	No	No	Yes	No	Cloud
Bright Sec.	ML for CI/CD	No	No	Yes	No	Cloud
ImmuniWeb	ML only	No	No	Yes	No	Cloud
Invicti	Risk Prioritisation	No	No	Yes	No	On-prem / Cloud
Acunetix	Target Profiling	No	No	Yes	No	On-prem / Cloud
OWASP ZAP	No	Yes	Yes	No	No	Local
Nuclei	No	Yes	Yes	No	No	Local
Glitch_ai	Full LLM + CodeStral	Conditional + Orchestrated	CLI + Remote	Exec Summaries + Mitigations	Adaptive + Contextual	Hybrid (Local + Remote)

Table 2.2.1 - Comparison of Existing Automated Tools vs. Glitch_ai

This comparative analysis confirms that, while many tools offer partial automation or limited AI assistance, none provide the full spectrum of capabilities required to perform flexible, intelligent penetration testing that is adaptable to local conditions, which Glitch_ai is designed to address.

2.3. AI and LLMs in Cybersecurity

AI is increasingly used in cybersecurity for threat detection, behavior profiling, anomaly classification, and response automation. LLMs have expanded AI's capabilities to contextual reasoning, natural language interaction, and dynamic decision-making.

In addition, AI is increasingly integrated into cybersecurity tools and platforms to optimize detection, reduce false positives, and enhance incident response. The most common applications include:

- **Intrusion Detection and Response**: Machine learning algorithms are widely used to identify deviations from normal behaviour in network traffic and user activities [13]. Systems such as SIEM (Security Information and Event Management) increasingly incorporate AI modules to accelerate detection and classification.
- **Malware and Phishing Detection:** Supervised models are trained with large data sets to classify suspicious code or URLs, thereby flagging potential threats before they can cause damage.
- **Vulnerability assessment and threat intelligence**: AI helps correlate known vulnerabilities (CVEs) with exposed services and assets, providing prioritised information on potential risks. Some platforms also use NLP to extract information from unstructured sources, such as blogs, forums, or threat reports [14].
- **Automation in CI/CD processes:** Tools such as Bright Security integrate AI into DevSecOps workflows, automating the detection of security issues during development cycles.
- **Behavioural biometrics and anomaly scoring:** AI models enable adaptive authentication by learning individual user patterns and triggering alerts when anomalies occur.

These applications span various sectors, including healthcare, retail, and government infrastructure, demonstrating the scalability and adaptability of AI in the face of constantly evolving threats [13]. Nonetheless, the literature also highlights significant challenges, including hallucinations, adversarial manipulation, and explainability. These limitations are analysed in detail in *Section 3.2.*

3. Technical background

This chapter provides the technical foundations necessary to **understand** the **design and implementation** of Glitch_ai. It covers three key areas: the basics and evolution of LLMs, the specific challenges posed by applying these models to cybersecurity tasks, and an overview of the tools and technologies used in the project.

3.1. Overview of Large Language Models (LLMs)

LLMs are a core technology in natural language processing, enabling the synthesis, classification, and generation of human-like text. Built on the Transformer architecture [15], they learn linguistic patterns through large-scale pre-training and can be adapted to specialised domains by fine-tuning on smaller datasets. This approach combines **general linguistic knowledge with task-specific expertise**, including domains such as medicine, law, or cybersecurity.

One of the main limitations of LLMs is the context window, which is the maximum number of tokens processed at once. Most open-source models, like Mistral, can handle only a few thousand tokens; exceeding this limit causes truncation and loss of information. For cybersecurity tasks, where tool outputs can span multiple pages of logs, prompt efficiency and token optimisation are essential.

Several surveys trace the evolution of LLMs, describing both their capabilities and limitations. Zhang [16] reviews fundamental models such as GPT-3, LLaMA, and PaLM, while Xue [17] examines their applications in cybersecurity, highlighting their uses in threat detection, vulnerability explanation, and technical translation. These studies confirm the growing feasibility of applying LLMs to high-risk environments.

Recent advances have made it increasingly feasible to integrate LLMs in real-world situations, especially where understanding language, reasoning, or automation is important. Consequently, cybersecurity represents a promising area for new uses of LLMs. The specific models and their functions within Glitch_ai are described in *Section 4.3.1.*

3.2. Challenges of LLMs in Cybersecurity

Although large language models offer significant capabilities, their application in cybersecurity introduces distinct challenges. Recognizing these limitations is essential for effective and secure system implementation.

- **Hallucinations (Misleading Outputs)**

One of the **most critical risks** associated with LLMs is hallucination, i.e., when the model generates information that seems plausible but is incorrect or fabricated. Recent surveys classify hallucinations into intrinsic and extrinsic types. Intrinsic hallucinations conflict with the input context, while extrinsic hallucinations produce unverifiable information [20]. For example, Xu [20] demonstrates that hallucinations are theoretically inevitable, which is particularly problematic in cybersecurity, where misleading diagnoses can compromise penetration testing or response measures.

- **Prompt Injection**

Prompt injection is **ranked** as the most severe LLM vulnerability in the **OWASP Top 10** (LLM01), as it allows attackers to manipulate instructions and override safeguards [19]. Attacks can be direct (plain-text commands) or indirect (malicious payloads hidden in documents, logs, or tool outputs). More advanced variants, such as adversarial prompting or "jailbreaks," can bypass defences. Wang [20] shows that failed prompts can even be iteratively refined into successful attacks. These risks stress the need for isolation, sanitisation, and validation when integrating LLMs into semi-automated systems like Glitch_ai.

- **Data poisoning and malicious fine-tuning**

LLMs fine-tuned on domain data (e.g., vulnerability logs, threat reports) may be compromised by poisoned samples, leading to skewed or backdoored outputs [21]. Even a small number of manipulated prompts can reliably trigger malicious behaviour without degrading overall performance [22]. This is especially dangerous in security contexts, where poisoned models could generate vulnerable code or misleading recommendations.

- **Limited context window and truncated outputs**

Transformer-based LLMs usually operate within context windows of a few thousand tokens (1000 - 4000 tokens). In cybersecurity, tool outputs (e.g., Nmap or Snort logs) often exceed this size, causing truncation and loss of critical information. Solutions like Sliding Window Attention Training (SWAT) have been proposed to extend context handling efficiently [23]. In systems such as Glitch_ai, these methods are essential for maintaining accuracy when interpreting long tool outputs.

- **False positives and precision issues**

LLMs applied to cybersecurity tasks, such as vulnerability detection, often have accuracy and consistency issues. These models can misclassify benign behaviors or code patterns as threats, resulting in high false-positive rates. This not only undermines analyst confidence but also increases classification workload and operational costs.

Recent literature reviews, such as the study by Zhang [24], confirm that accuracy issues remain a significant barrier to the practical deployment of LLMs in security environments. The review highlights that while LLMs show promise across a wide range of cybersecurity tasks, improving accuracy and reducing false positives are key challenges for future research.

- **Scalability, performance, and data limitations**

Effective **fine-tuning requires large-scale**, representative datasets, but cybersecurity corpora are often scarce and domain-limited. As Hanxiang [25] notes, this reduces generalisation and robustness in real-world deployments. Without any domain-specific tuning, augmentation, or efficient prompting, models risk overfitting or underperforming in dynamic threat environments.

In summary, large language models make new types of security tools possible; however, there are still open questions about how much we can trust them, their reliability, and how well they scale. Solving these problems is necessary to safely use AI-based penetration testing systems such as Glitch_ai.

3.3. Tools and Technology Overview

Glitch_ai integrates a range of technologies, including traditional penetration testing tools, backend and frontend frameworks, and large language models (LLMs). This section provides an overview of the core technologies used, their roles within the system, and their relevance to cybersecurity.

Penetration Testing Tools

Glitch_ai **incorporates** several widely used **open-source tools**, each selected for its effectiveness in a specific phase of the penetration testing lifecycle. These include:

- **Dmitry:** A multipurpose recognition tool that collects information such as Whois records, subdomains, open ports, etc [26].
- **Hydra:** A fast and flexible brute force tool that supports numerous protocols, including SSH, HTTP, FTP, and RDP. It is designed to perform credential testing in network environments [27].
- **Nikto:** A web server scanner maintained by CISA that detects outdated software, insecure configurations, and known vulnerabilities in HTTP services [28].
- **Nmap:** A powerful network scanner for host detection, port scanning, and operating system detection. It is widely used for network mapping and vulnerability assessment [29].
- **Sublist3r:** A fast subdomain enumeration tool that uses open-source intelligence (OSINT) sources, ideal for early reconnaissance in mapping the external attack surface [30].

- **SQLmap:** An automated tool for detecting and exploiting SQL injection vulnerabilities. It supports a wide range of databases and offers extensive identification and takeover functions [31].
- **WhatWeb:** A website fingerprinting tool that identifies the technologies used by web applications (CMS, frameworks, JS libraries, etc.), helping to create target profiles [32].
- **Whois:** A standard utility for querying domain registration information, including ownership, registrar details, and DNS records [33].
- **Wpscan:** A specialised scanner to identify vulnerabilities in WordPress installations, including issues with plugins and themes, weak credentials, and configuration flaws [34].

Together, **these tools support a modular and extensible architecture** within Glitch_ai, enabling automated and user-driven assessments across network, web, and application layers.

Backend and Execution Logic

The backend is built with FastAPI, a modern, high-performance **Python web framework** designed to create APIs with automatic OpenAPI documentation and support for asynchronous operations. It is particularly well-suited to microservices and rapid development thanks to its speed, type hints, and ease of integration with modern Python tools [35].

The execution of tools within Glitch_ai is managed through a modular orchestration layer that adapts to the user's role (standard or superuser), the tools available, and system permissions. If a command cannot be executed locally due to missing dependencies, lack of privileges, or operating system restrictions, the system automatically redirects execution to a secure remote backend hosted on a Linux virtual machine with all necessary tools pre-installed.

LLM Integration

Glitch_ai integrates large language models developed by Mistral, a company focused on high-performance, open-source transformer models designed for reasoning, code generation, and complex natural language tasks [36]. Mistral models are known for their strong performance in benchmark tests and their efficient architecture, which offers high performance and ease of implementation through open APIs.

Instead of relying on a single general-purpose model, Glitch_ai dynamically selects from four specialised models within the Mistral ecosystem, assigning each one the task for which it is best suited. This model routing strategy improves both performance and task relevance, ensuring that language understanding, code generation, and executive synthesis are handled by models that are tailored or optimised for those domains. The selected models [37] are:

- **Mistral 7B:** A dense decoding-only transformer with 7 billion parameters, trained with a context window of 32,000 tokens. It is used in Glitch_ai to interpret detailed technical results from tools such as Nmap or SQLmap and transform them into structured information. Its strength lies in general-purpose reasoning and understanding long contexts.
- **Mistral 3B:** A smaller, lighter model that offers lower latency without sacrificing robust overall capabilities. At Glitch_ai, it is responsible for generating executive summaries, transforming raw results into clear, non-technical explanations suitable for a more general audience.
- **Mistral NEMO:** A refined and optimised variant for structured tasks such as classification, mitigation assignment, and threat categorisation. Glitch_ai uses it to translate technical findings into defensive recommendations, taking advantage of its structured response format capabilities.
- **CodeStral:** A model trained for code comprehension and generation, based on the StarCoder2 architecture. It is applied in backup and recovery modules, where Glitch_ai uses it to suggest corrected commands or scripts after failed executions.

This architecture ensures that Glitch_ai adapts its AI reasoning to the demands of each task, minimising ambiguity and maximising effectiveness in both technical and non-technical results.

Frontend and User Interface

The interface is built with HTML, JavaScript, and CSS, and is structured as a dynamic panel that allows users to:

- Authenticate via Firebase.
- Select tools and configure targets.
- Visualise execution progress and results.
- Generate and download reports (in JSON or PDF format).

Firebase is a development platform maintained by Google that offers secure and scalable backend services for web and mobile applications. In Glitch_ai, **Firebase Authentication manages user roles and access**: by default, the application runs in standard user mode, allowing unprivileged operations. Users can optionally log in to enter superuser mode, unlocking additional features such as installing local tools and executing privileged commands.

The interface dynamically adapts to both the user's role and operating system. For instance, installation options and commands requiring elevated privileges are displayed only when the user is authenticated as a superuser on Linux. This makes the system easier to use and more secure, and it helps users identify available tools immediately.

4. Methodology

This chapter describes the methodology followed during the development of Glitch_ai. It covers the initial design principles of the system, the data collection process, the tuning and prompt engineering of large language models, and the step-by-step implementation plan.

Each section details the technical decisions made during development and shows how the system moved from concept to execution in clear steps.

4.1. System Design

Glitch_ai is an automated penetration testing framework that integrates classic scanning tools with AI-driven analysis in a modular, adaptive architecture. **The system operates autonomously from tool selection to reporting**, while adapting to the user's role and execution environment.

Users interact through a web frontend. By default, they act as standard users, but Firebase authentication gets superuser rights, allowing them to run privileged commands and install tools. When users log in, the system checks the operating system and lists the available tools, indicating which can run locally and which require manual installation or a remote backend.

After a target is specified, Glitch_ai validates its format and enables or disables tools as necessary to prevent errors. Tools execute either locally or on a Linux-based remote backend, depending on user permissions and resource availability. If failures occur, the system initiates a retry and fallback mechanism. Persistent errors prompt LLM reasoning to suggest and validate alternative commands.

Finally, LLM modules analyse tool outputs, detect vulnerabilities, and propose mitigations. Results are compiled into two types of reports: a structured JSON file and a formatted PDF that combines technical findings with executive-level summaries.

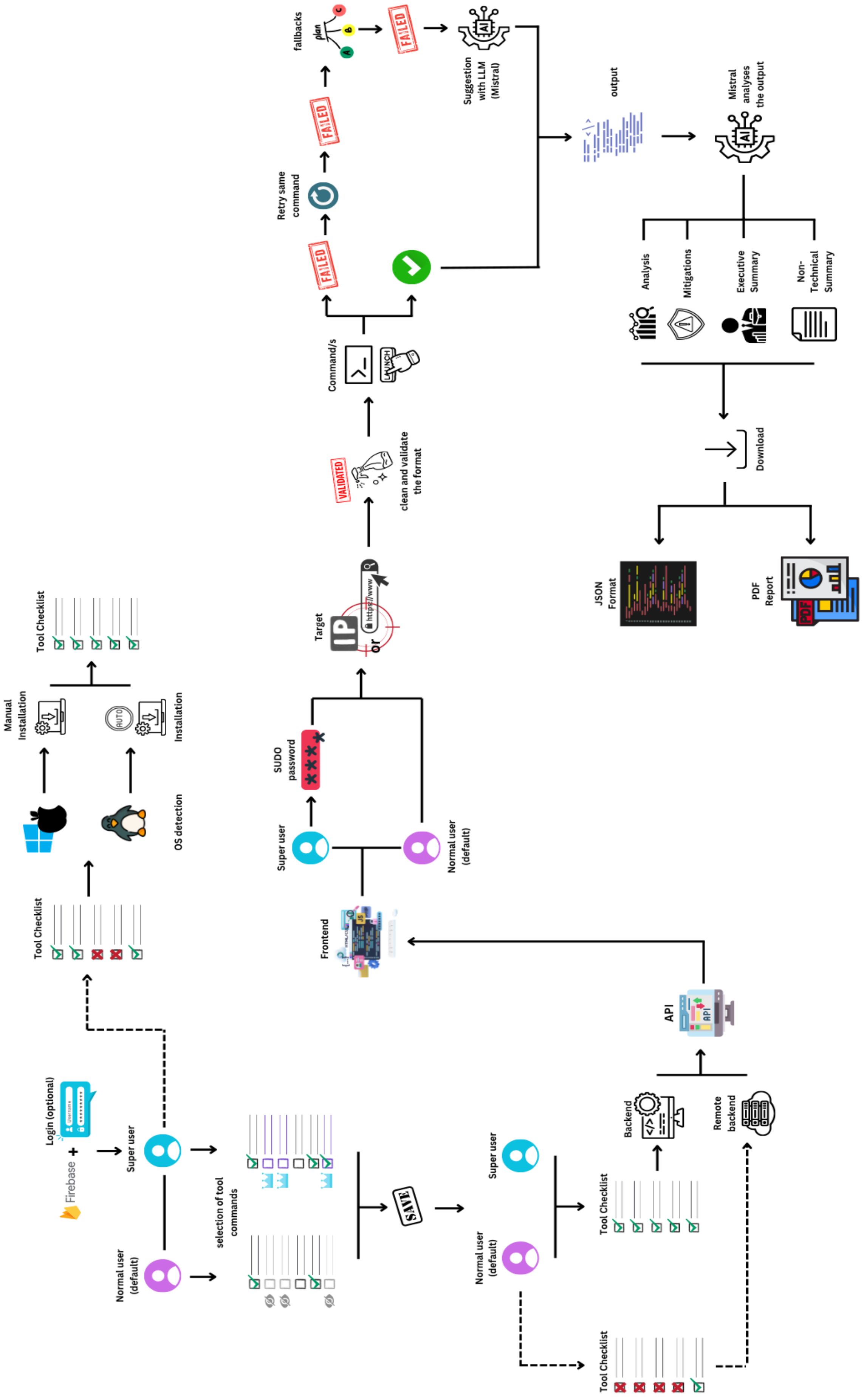


Illustration 4.1.1 - Glitch_ai workflow from tool selection to LLM-based report generation

4.2. Data Collection and Dataset Preparation

To support the training and contextual reasoning of the large language models integrated into Glitch_ai, a total of **14 custom datasets** were created or adapted. These datasets were carefully selected, generated, or cleaned depending on their intended purpose: either for **fine-tuning** or for **prompt engineering.**

All datasets used for fine-tuning were converted to .jsonl **(JSON Lines)** format, as required by the Mistral fine-tuning interface. Unlike standard .json files (which typically store a single object or an array of objects) .jsonl files contain **one independent JSON object per line**, allowing for more efficient line-by-line parsing and streaming. This format is particularly suited for large-scale training pipelines and AI model ingestion [38].

Most original datasets were provided in formats such as .csv, .xlsx, or .json, and were converted to .jsonl using custom Jupyter notebooks. Cleaning tasks included:

- Removing null or malformed entries.
- Normalizing fields (e.g., stripping whitespace, correcting labels).
- Creating instruction-response pairs for LLM fine-tuning.

These datasets provided the foundation for model fine-tuning and context management strategies, which are detailed in the subsequent section.

4.3. Fine-Tuning Strategy

Fine-tuning in machine learning is the process of adapting a pre-trained model for specific tasks or use cases. It has become a fundamental deep learning technique, particularly in the training process of foundation models used for generative AI [39].

In the context of Glitch_ai, fine-tuning was used to customize large language models (LLMs) for tasks such as:

- Interpreting the output of penetration testing tools,
- Generating mitigation strategies,
- Producing executive summaries,
- Suggesting adaptive fixes when tools failed.

Glitch_ai employs multiple fine-tuned models, each dedicated to a specific task or output. This section describes how the fine-tuning process was designed, executed, and aligned with the system's functional objectives.

4.3.1. Objectives of Fine-Tuning

Glitch_ai utilizes multiple models from the Mistral AI ecosystem, which provides a family of large, open-source language models optimized for flexibility, local deployment, and fine-tuning. These models were selected for their high performance, transparent licensing, and the ability to customise them without additional costs or API restrictions.

Each model was adjusted to specialise in a specific function within the system:

- **Mistral 7B**: focused on the technical interpretation of analysis results and detection logic.
- **Mistral 3B**: trained to generate non-technical executive summaries and results that are easy for people to understand.
- **Mistral NEMO**: used for interpreting structured data, advising on mitigation, and classifying the severity of threats.
- **CodeStral**: specialised in adaptive reasoning, providing alternative commands and analysing errors or incorrect tool configurations.

This setup helps each model give consistent, relevant, and high-quality results within its domain, reducing the risk of generic or irrelevant responses.

Additionally, the datasets described in *Section 4.2* were mapped to the models based on task relevance. For example:

- Technical outputs from Nmap and SQLmap were routed to Mistral 7B.
- Executive summary data was directed to Mistral 3B.
- Log-based and structured threat data (e.g., from SDN or threat detection logs) were used with Mistral NEMO.
- Command failure scenarios were used to train CodeStral.

Keeping tasks separate made it possible to improve each model without mixing up its behaviors.

4.3.2. Training Parameters

All fine-tuning was conducted using the official Mistral web interface. For most training sessions, the following hyperparameters were applied:

- **Epochs**: 5
- **Learning rate**: 0.00001

These values were selected based on recommendations for fine-tuning large language models to **prevent overfitting**. According to IBM [40] and Hugging Face [41], to ensure sufficient adaptation without overfitting. This setup allows models like Mistral 7B and 3B to learn the task structure while still generalizing well.

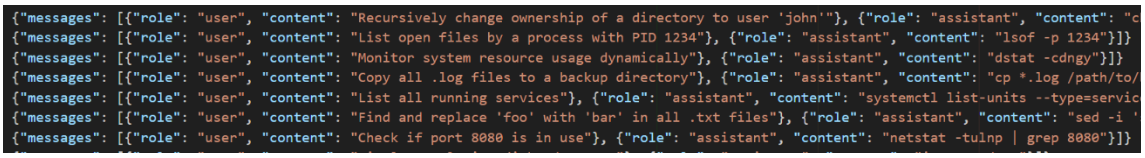
For smaller or repetitive datasets, such as the SDN set used with Mistral NEMO, the number of **epochs** was **reduced to 3**, in line with guidance from Milvus [42], to prevent memorisation. The learning rate remained fixed at 1e-5, as higher values risk catastrophic forgetting [41], [43].

Due to API restrictions on Mistral's free plan, training tasks were distributed across two accounts, allowing for parallel runs and compliance with quota limits. This strategy ensured that multiple LLM variants could be fine-tuned efficiently without exceeding usage constraints.

This approach enabled Glitch_ai to fine-tune multiple LLM versions at the same time by using different Mistral accounts for separate tasks, staying within API limits.

4.3.3. Instruction Format and Token Control

All training samples were structured in .jsonl format with a chat-style question and answer setup, as expected by conversational LLMs. Each entry has a user prompt and an assistant response, simulating a real conversation between the user and the system. As illustrated in *Illustration 4.3.1*, each message pair was designed to represent a **Prompt engineering approach.**



```
{"messages": [{"role": "user", "content": "Recursively change ownership of a directory to user 'john'"}, {"role": "assistant", "content": "c
{"messages": [{"role": "user", "content": "List open files by a process with PID 1234"}, {"role": "assistant", "content": "lsof -p 1234"}]}
{"messages": [{"role": "user", "content": "Monitor system resource usage dynamically"}, {"role": "assistant", "content": "dstat -cdngy"}]}
{"messages": [{"role": "user", "content": "Copy all .log files to a backup directory"}, {"role": "assistant", "content": "cp *.log /path/to/
{"messages": [{"role": "user", "content": "List all running services"}, {"role": "assistant", "content": "systemctl list-units --type=servic
{"messages": [{"role": "user", "content": "Find and replace 'foo' with 'bar' in all .txt files"}, {"role": "assistant", "content": "sed -i '
{"messages": [{"role": "user", "content": "Check if port 8080 is in use"}, {"role": "assistant", "content": "netstat -tulnp | grep 8080"}]}
```

Illustration 4.3.1 - Sample entries from complex_linux_commands_mistral.jsonl following a chat-style fine-tuning format

To ensure compatibility with the model's input constraints, all samples were reviewed to ensure they **did not exceed 1024 tokens**, including both input and output text.

Most open-weight models, including those hosted by Mistral through its fine-tuning interface, operate with a context window limit, i.e., the total number of tokens that the model can 'see' at once. For certain Mistral variants, especially in web-based fine-tuning, exceeding 1024 tokens can cause:

- Truncated input (the end of the prompt or output is cut off),
- Training failures or silent errors.
- An alignment issue between prompts and labels.

This limit includes not only the visible prompt and response, but also formatting tokens and role metadata (e.g., 'role': 'user' and system-specific markers). For this reason, even a moderately sized text can approach this limit.

4.4. Prompt engineering Design

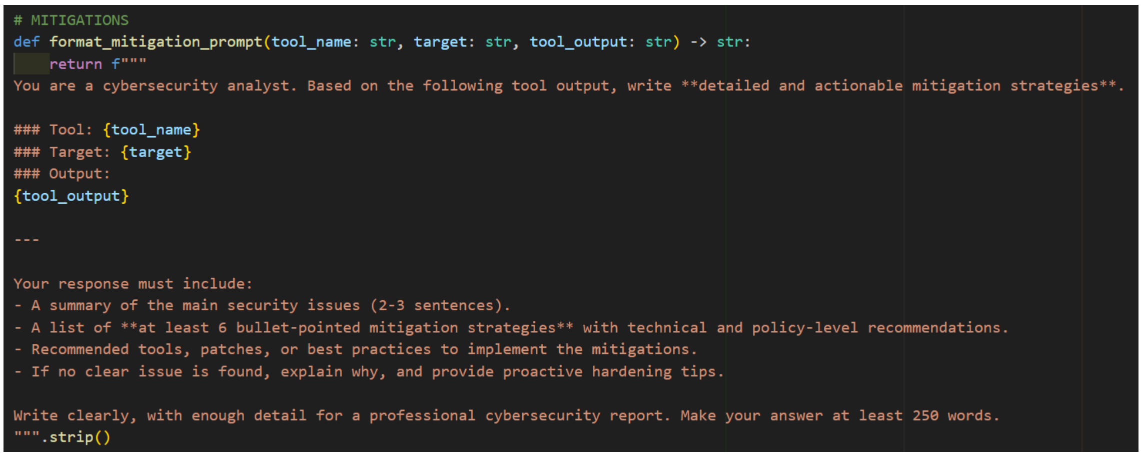
Although Glitch_ai relies primarily on **fine-tuned models** to deliver task-specific results, a minimal layer of **prompt engineering** is still necessary to structure the interaction between the system and the language models. This prompt layer is neither advanced nor dynamic, but it plays a key role in ensuring that instructions are clear, consistent, and safe.

To achieve this, all prompts sent to the models are constructed using a dedicated Python module called *prompt_templates.py*. This module defines **static prompt templates** for various use cases, including:

- Technical summaries of tool outputs,
- Non-technical summaries for executive reporting,
- Mitigation advice based on detected vulnerabilities,
- Risk level classification,
- Fallback suggestions when tools fail (used with CodeStral).

Each function in the module returns a formatted string sent to the model through the API. These templates do not use few-shot examples or logic chains, but they set a clear instruction-response format to guide the model's behavior during use.

Illustration 4.4.1 shows one of the prompt templates used specifically to generate mitigation recommendations based on the results of a security tool. The prompt clearly defines the model's role, inserts structured input fields (*tool_name*, *target*, *tool_output*), and includes detailed instructions to guide both the style and depth of the generated response.



```
# MITIGATIONS
def format_mitigation_prompt(tool_name: str, target: str, tool_output: str) -> str:
    return f"""
You are a cybersecurity analyst. Based on the following tool output, write **detailed and actionable mitigation strategies**.

### Tool: {tool_name}
### Target: {target}
### Output:
{tool_output}

---

Your response must include:
- A summary of the main security issues (2-3 sentences).
- A list of **at least 6 bullet-pointed mitigation strategies** with technical and policy-level recommendations.
- Recommended tools, patches, or best practices to implement the mitigations.
- If no clear issue is found, explain why, and provide proactive hardening tips.

Write clearly, with enough detail for a professional cybersecurity report. Make your answer at least 250 words.
""".strip()
```

Illustration 4.4.1 - Structured mitigation prompt template used for generating LLM responses in Glitch_ai

This structured approach ensures that, even after fine-tuning, the model always gets a clear context. This helps maintain consistent, clear, and professional output.

Although model weights are already adapted to each task (see *Section 4.3*), the use of prompt templates serves as an additional control layer. Specifically, prompt design helps to:

- Minimise ambiguity in open-ended instructions.
- Adjust the tone and language depending on the type of user (technical or non-technical).
- Standardise inputs across all tools and situations.
- Maintain modularity by directing requests to different LLM endpoints.

By encapsulating all prompt logic in one file (*prompt_templates.py*), the system remains both maintainable and scalable. If the tone, format, or audience needs to change, only the templates have to be changed, without retraining any model.

Finally, all prompts are designed to minimize token usage and remain concise. Tool outputs are pre-processed and condensed before integration into templates to ensure compatibility with each model's context window. This practice reduces the risk of hallucinations or performance degradation caused by excessive or irrelevant input.

4.5. Implementation Plan

Glitch_ai was built using a modular, step-by-step development plan that aligned with the system architecture outlined in *Section 5.1*. The goal was to build and test each major component: frontend, backend, tool integration, LLM logic, and reporting, while maintaining flexibility for future updates.

- **Phase 1: Local Backend and Command Execution:** The initial stage focused on creating the FastAPI backend with basic endpoints and local tool execution (e.g., Nmap, Dmitry). This provided the foundation for secure and structured command orchestration.
- **Phase 2: Frontend Interface and User Roles:** Next, the web interface was introduced with tool selection, target validation, and Firebase-based role management. Standard and superuser modes were differentiated, with superusers granted privileges such as local tool installation and sudo execution.
- **Phase 3: LLM Integration and Adaptive Logic**: Once the core workflow was operational, large language models (Mistral 7B, 3B, NEMO, and CodeStral) were incorporated to provide adaptive retry logic, interpretation of tool outputs, and generation of human-readable summaries.
- **Phase 4: Remote Backend for Cross-Platform Support**: To support users without local privileges or Linux environments, a remote backend was deployed on a preconfigured virtual machine. This ensured full functionality even in restricted platforms, with robust client–server communication for remote scans.
- **Phase 5: Report Generation and Output Formatting**: Finally, reporting modules were added, producing both JSON (structured and automation-ready) and PDF (client-friendly) formats. Reports combine technical results with LLM-generated insights, summaries, and mitigation advice.

This modular, phased approach enabled independent development, testing, and refinement of each system component while maintaining overall integration and readiness for future updates.

5. System Architecture and Implementation

This section presents the architecture and technical implementation of Glitch_ai, an automated penetration testing platform based on artificial intelligence developed as part of this project. Glitch_ai integrates traditional security tools with a large language model (LLM) trained to automate vulnerability analysis, adapt to real-time attack strategies, and generate human-readable reports.

The system employs **a modular architecture that supports scalability**. Users may operate the platform locally or remotely, contingent on assigned roles and permissions. Key features include an intuitive user interface, a robust backend for tool management, a remote service for restricted environments, and a language model that interprets results and guides penetration testing procedures.

5.1. Overall Architecture

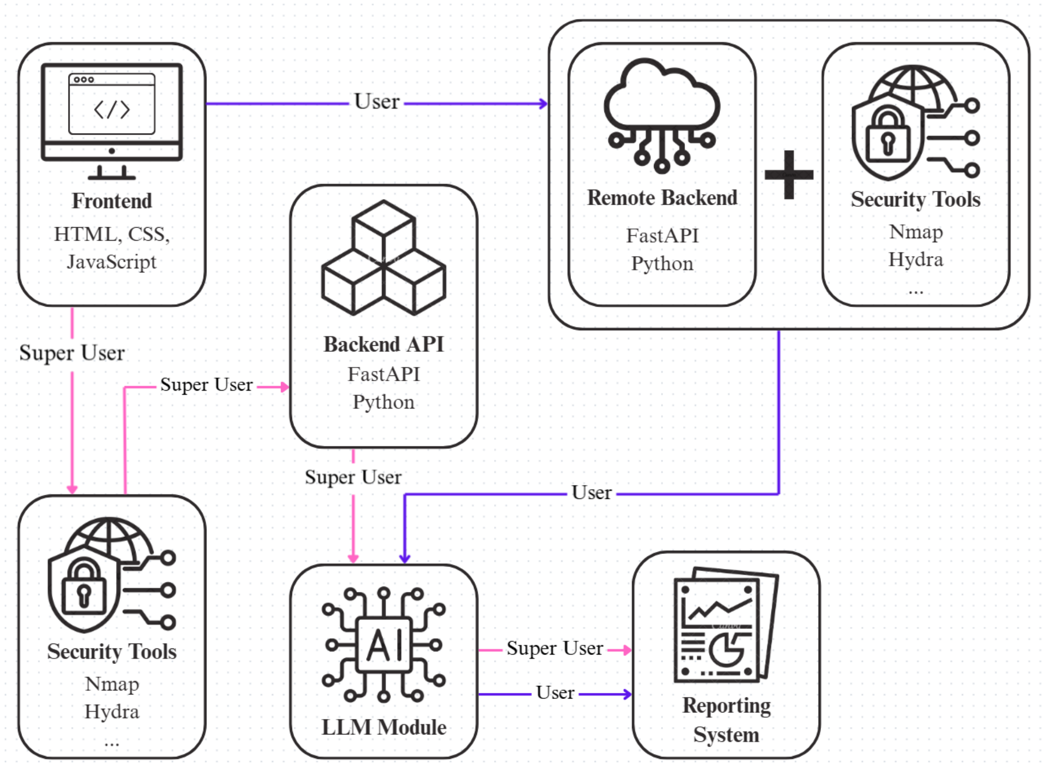


Illustration 4.5.1 - Architecture of Glitch_ai

Glitch_ai follows a modular architecture designed for flexibility and scalability, integrating classic security tools and AI modules in an automated workflow. The main parts are the frontend, backend API, security tools layer, LLM module, reporting system, and an optional remote backend.

The **frontend**, built with HTML, CSS, and JavaScript, provides a chatbot-style interface where users select tools, set targets, and view results. It manages authentication through Firebase and dynamically adapts to user roles: standard users (“U”) or superusers (“S”), the latter of which have sudo privileges.

The **backend**, implemented with FastAPI and Python, orchestrates execution: validating targets, managing permissions, and triggering tools either locally or via the remote backend when local execution is not possible.

The **security tools layer** integrates widely used penetration testing utilities, such as Nmap, Hydra, Nikto, Sublist3r, SQLmap, WhatWeb, Whois, Dmitry, and Wpscan. These utilities can be combined and executed sequentially, depending on the target and user role.

The **LLM module** uses fine-tuned Mistral models to explain tool results, suggest next steps, and create advice and summaries for both technical and non-technical users. Reports are available as structured JSON files and easy-to-read PDFs, combining found issues found and AI recommendations.

Finally, the **remote backend**, hosted on a dedicated Linux VM, extends functionality to users lacking installation privileges or required tools, ensuring full compatibility across environments.

5.2. Frontend Design

The **Glitch_ai** interface was developed using standard web technologies: **HTML**, **CSS**, and **JavaScript**. Its main goal is to provide an intuitive and user-friendly environment that enables users to configure and launch penetration tests without the need to interact with the command line.

The interface follows a modular design and is structured around several key components:

- **Tool Selection Panel**: Users can browse and select tools from a categorized list. The Reconnaissance section includes tools such as Dmitry, Nmap, Sublist3r, and Whois, aimed at gathering publicly available yet non-obvious information. The Vulnerability Discovery category focuses on identifying potential weaknesses and includes tools like Nikto, SQLmap, WhatWeb, and Wpscan. Finally, the Exploitation category assesses user-related vulnerabilities—such as weak passwords—using tools like Hydra to evaluate whether password security meets minimum standards.

  Each tool offers predefined commands with hover-based tooltips that explain their purpose. These can be activated individually or combined depending on the user's needs.

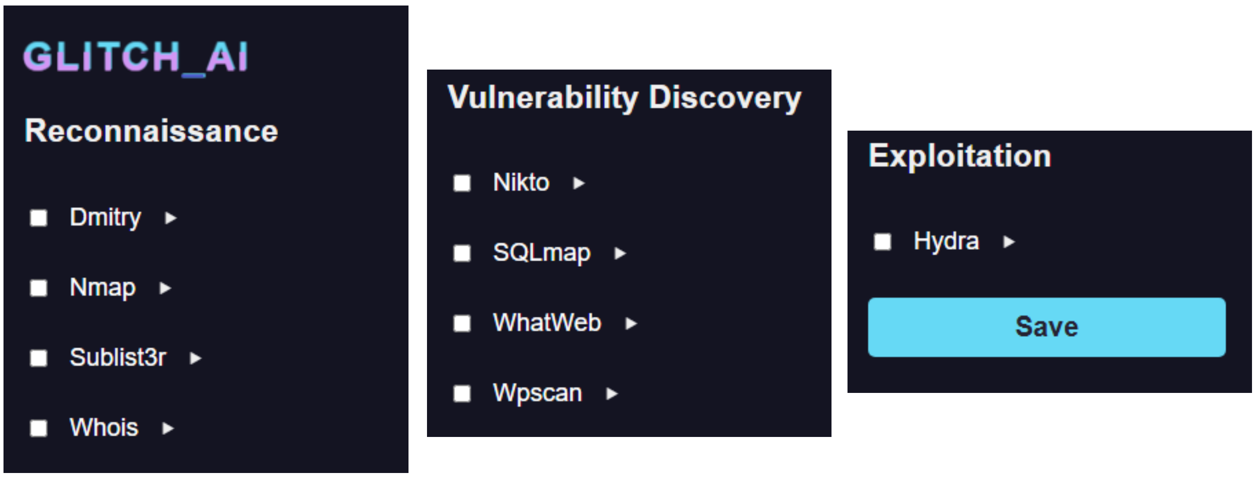
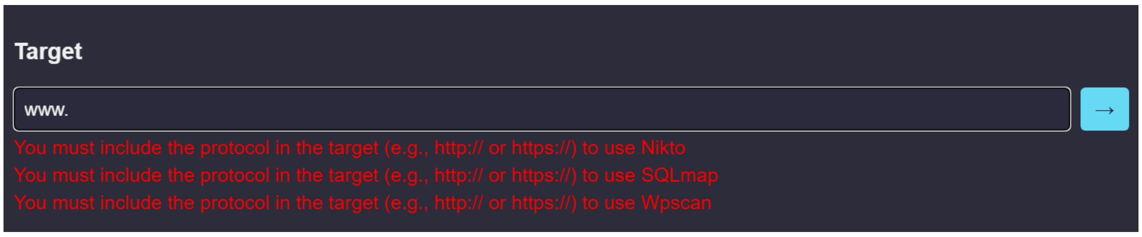


Illustration 5.2.1 - Categorized tool selection panel

- **Target Input Field**: The system allows users to enter either a domain name or an IP address as the scan target. Input validation ensures proper formatting and displays real-time warnings when specific tools require protocol information (e.g., Nikto, SQLmap or Wpscan).



You must include the protocol in the target (e.g., http:// or https://) to use Nikto
You must include the protocol in the target (e.g., http:// or https://) to use SQLmap
You must include the protocol in the target (e.g., http:// or https://) to use Wpscan

Illustration 5.2.2 - Protocol validation warning for tools requiring HTTP/HTTPS

- **Role-Based Interface Behavior**: Depending on the user role, standard user ("U") or superuser ("S"), the interface dynamically adjusts. For instance, only superusers can install tools or run commands that require sudo privileges. This behavior ensures security and prevents misuse of elevated access.
- **Tool Availability Checker**: By clicking on the Glitch_ai logo, users can access a pop-up window displaying the status of installed tools. On Linux systems with sudo privileges, it includes the option to automatically install missing tools.

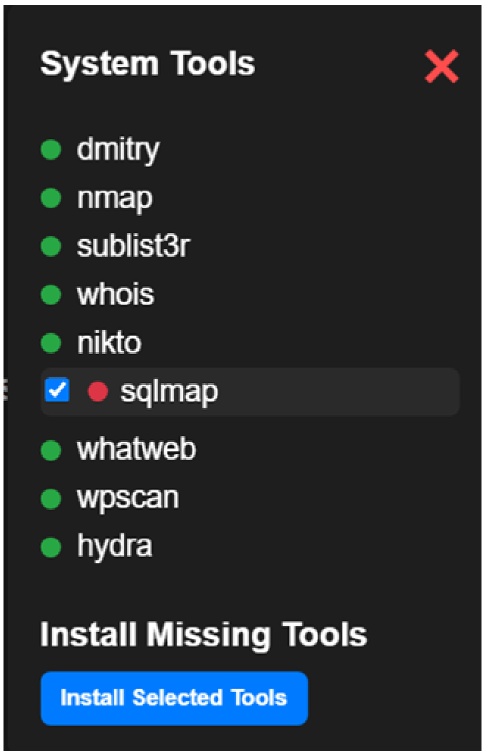
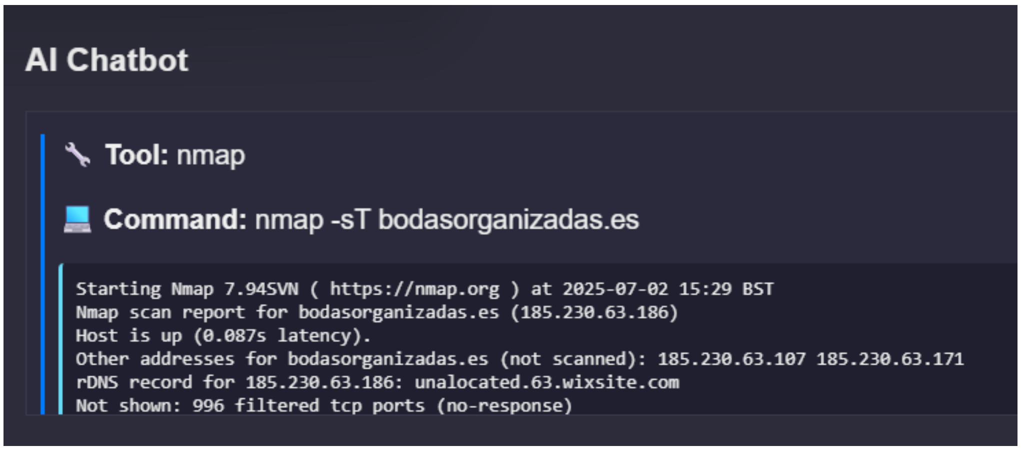


Illustration 5.2.3 - Tool availability popup with installation options

- **Command Summary and Launch Button**: After selecting the desired tools and entering a valid target, the interface generates a list of ready-to-run commands. These are stored locally and only sent to the backend upon execution.
- **Chatbot-Style Output Console**: Results are displayed in a scrollable, terminal-inspired chatbot. Each tool's output is presented in a structured format with visual indicators for success, failure, or warnings, simplifying the interpretation of technical output.


AI Chatbot

Tool: nmap

Command: nmap -sT bodasorganizadas.es

```
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-07-02 15:29 BST
Nmap scan report for bodasorganizadas.es (185.230.63.186)
Host is up (0.087s latency).
Other addresses for bodasorganizadas.es (not scanned): 185.230.63.107 185.230.63.171
rDNS record for 185.230.63.186: unalocated.63.wixsite.com
Not shown: 996 filtered tcp ports (no-response)
```

Illustration 5.2.4 - Chatbot showing tool execution and results

- **Reports**: Once the scan is completed, users can download the results in **JSON** or **PDF** format. The reports include both raw output and AI-generated summaries.

- **Authentication System**: Glitch_ai integrates Firebase Authentication to manage user sessions and roles. Based on the login credentials, the system assigns the correct role and adjusts available functionalities accordingly.

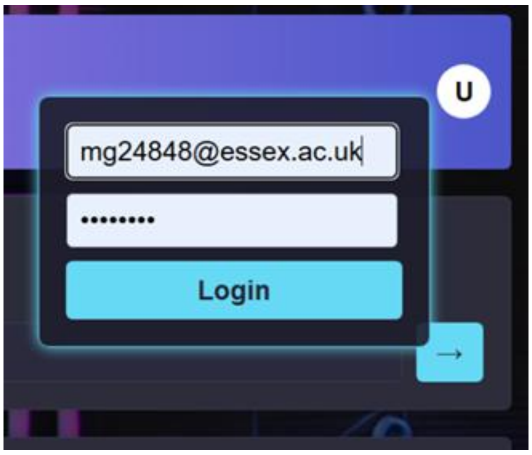


Illustration 5.2.5 - Firebase login screen

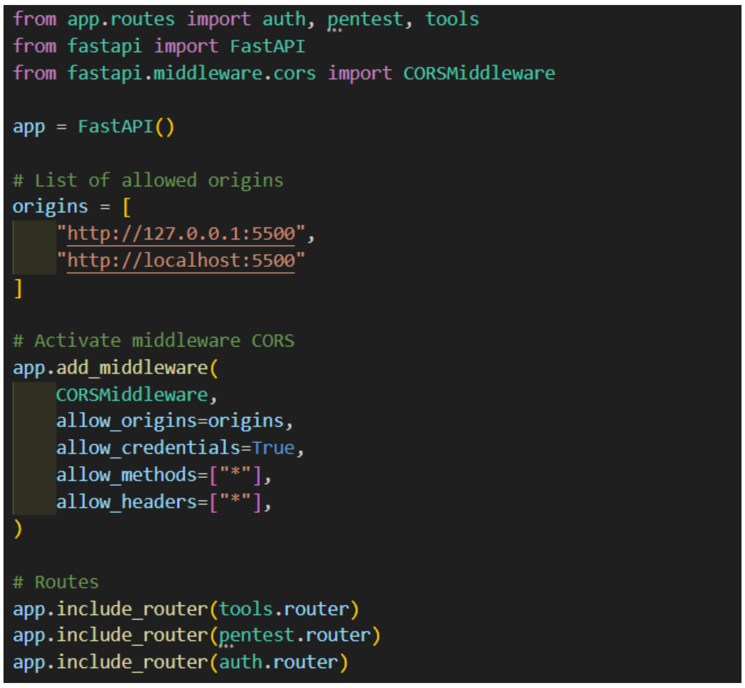
Glitch_ai makes security tools easier to use with a guided interface that adapts to each user's role. This approach helps beginners get started while still giving advanced users the flexibility they need.

5.3. Backend API & Execution Logic

The backend of **Glitch_ai** is implemented using **FastAPI,** a modern, fast (high-performance), web framework for building APIs with Python based on standard Python type hints [35]. The codebase is modular and organized into logical components, which improves readability, maintainability, and scalability.

The entry point of the backend is defined in the *main.py* file, where the FastAPI instance is created and the various API routers are included. Each functional group of routes, such as pentest execution, tool management, or installation logic, is separated into dedicated Python modules under the app/ directory.

A separate backend instance runs remotely on a Linux-based virtual machine and handles restricted execution cases.



```python
from app.routes import auth, pentest, tools
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI()

# List of allowed origins
origins = [
    "http://127.0.0.1:5500",
    "http://localhost:5500"
]

# Activate middleware CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Routes
app.include_router(tools.router)
app.include_router(pentest.router)
app.include_router(auth.router)
```

Illustration 5.3.1 - main.py structure showing FastAPI app and router mounting

The full list of API endpoints is grouped below by functional area:

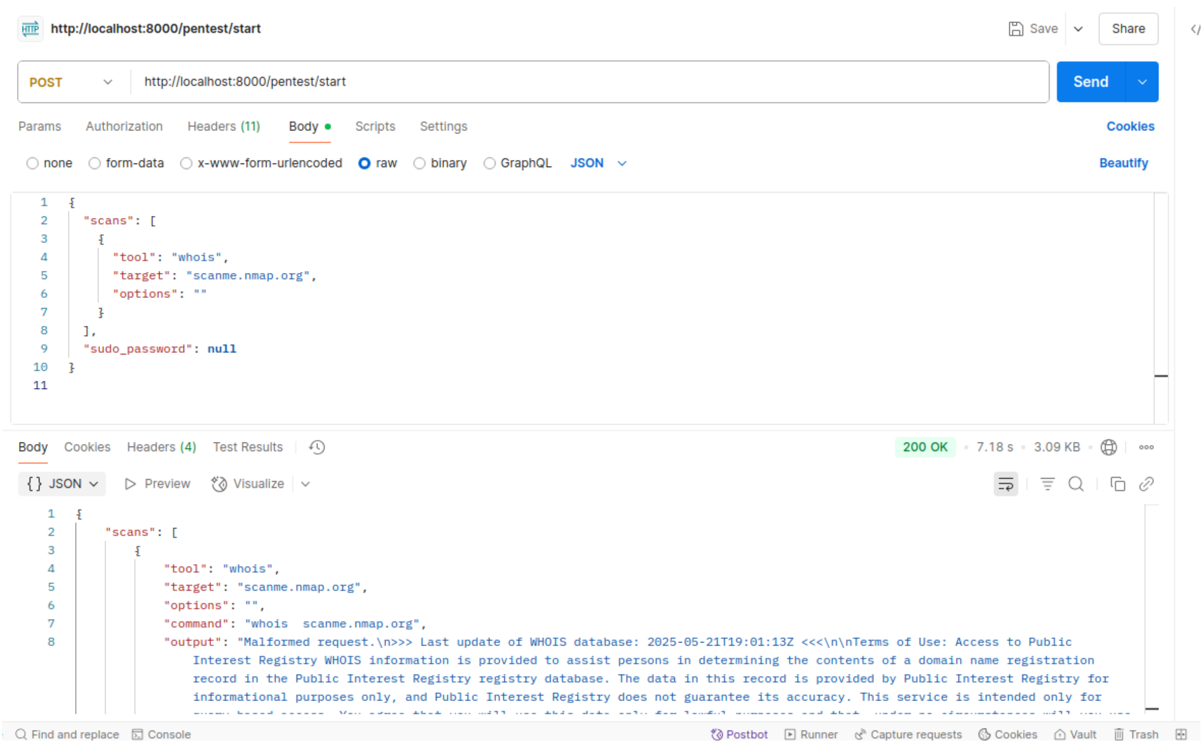
1. <u>Pentest Execution and Result Management</u>

- *POST /pentest/start*

  This is the main endpoint of the system. It receives:

  - a list of selected tools and their corresponding command options,
  - a target (domain or IP)
  - and the system password (if provided and required for sudo-based commands).

  The backend validates the input and dynamically builds the command queue. Based on system conditions and user privileges, it determines whether to run the scan locally or delegate it to the remote backend. If any command fails, adaptive retry logic is triggered, and after several failed attempts, backup mechanisms or LLM suggestions are applied.



http://localhost:8000/pentest/start

```
{
  "scans": [
    {
      "tool": "whois",
      "target": "scanme.nmap.org",
      "options": ""
    }
  ],
  "sudo_password": null
}
```

```
{
    "scans": [
        {
            "tool": "whois",
            "target": "scanme.nmap.org",
            "options": "",
            "command": "whois  scanme.nmap.org",
            "output": "Malformed request.\n>>> Last update of WHOIS database: 2025-05-21T19:01:13Z <<<\n\nTerms of Use: Access to Public
                Interest Registry WHOIS information is provided to assist persons in determining the contents of a domain name registration
                record in the Public Interest Registry registry database. The data in this record is provided by Public Interest Registry for
                informational purposes only, and Public Interest Registry does not guarantee its accuracy. This service is intended only for
```

Illustration 5.3.2 - Postman test of /pentest/start, showing a valid request and backend JSON response

This endpoint is central to the overall execution logic, and its behavior depends on both the tools selected and the role of the current user (standard or superuser)

All routes, the backend uses Pydantic models for request validation and response shaping. Is the most widely used data validation library for Python and a consistent structure across the system [44].

Error handling is managed through FastAPI’s native exception system, with appropriate status codes and descriptive messages. For example:

- **401 Unauthorized** if the token is invalid or missing,
- **422 Unprocessable** Entity if the request format is incorrect or incomplete,
- and **200 OK** if the request succeeds, as shown in *Illustration 5.3.2*.

- *POST /pentest/generate-response*

  This endpoint is used after the analysis is complete. It formats the tool's results into a structured prompt for the LLM model, which then generates human-readable summaries in two formats. Mitigation advice and strategic knowledge. This endpoint acts as a bridge between raw technical results and natural language interpretation, which is described in more detail in *Section 5.4.*

- *GET /pentest/download*

  Returns the latest analysis result as a structured JSON file. This allows users to access raw data for offline analysis or to generate their own reports. It also serves as input for the /pentest/download-pdf endpoint. This endpoint, as well as the next one, will be explained in more detail in *Section 5.8.*

- *POST /pentest/download-pdf*

  Generates a clear visual report in PDF format based on the JSON result created in the previous endpoint. This process includes the generation of the layout, AI-generated summaries, graphical indicators (e.g., icons of criticality differentiated by colour at different levels) and timestamps. The complete report generation flow is described in *Section 5.8.*

2. <u>Tool Management and System Information</u>

- *GET /tools/list*

  This endpoint returns a list of all supported tools along with their installation status. For each tool, the backend checks whether the binary is present in the system's PATH using '*shutil.which()*'. The response contains two arrays: one for installed tools and one for missing tools. This information is used by the frontend to:

  - show visual indicators next to each tool (available/unavailable),
  - and conditionally enable or disable the corresponding commands.

- *GET /tools/metadata*

  This endpoint returns descriptive metadata for each tool, including:

  - a brief explanation of what the tool does,
  - its category (e.g., reconnaissance, exploitation),
  - whether it requires sudo permissions,
  - and whether the target needs to include a protocol.

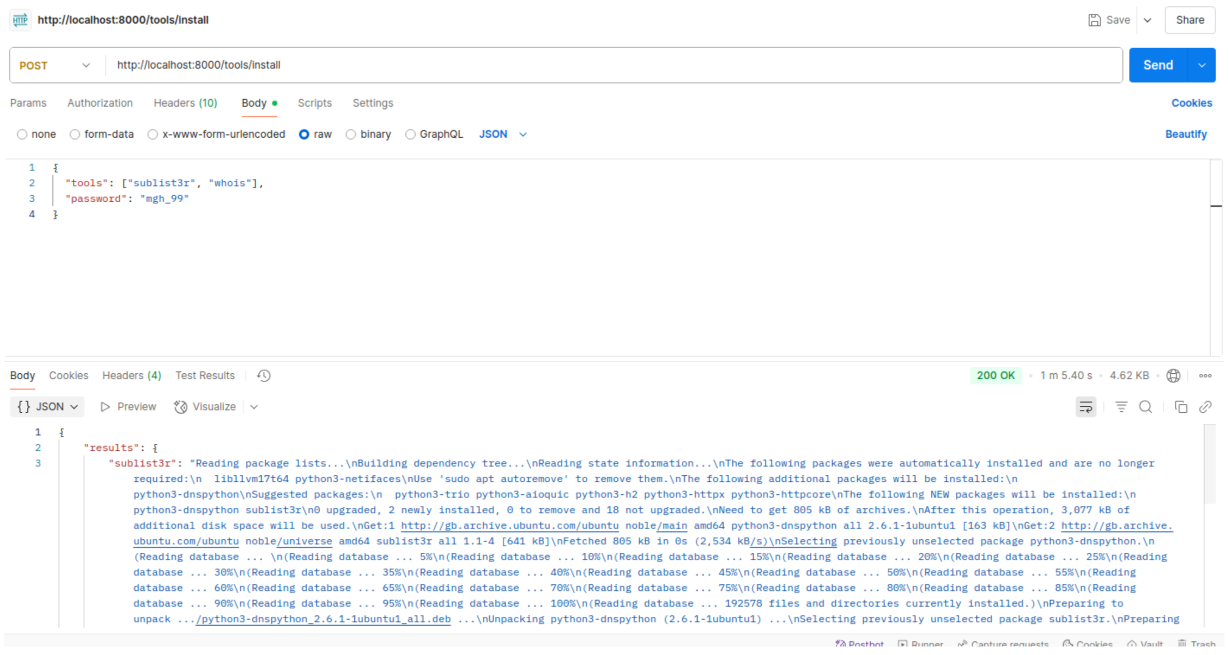
  This metadata is used in the frontend to display tooltips when hovering over each command and to enforce validation rules (e.g., Nikto requiring http:// or https:// in the target).

- *POST /tools/install*

  This endpoint allows users with superuser ("S") privileges to install missing tools directly from the interface, provided that the system is running Linux. The backend first validates:

  - that the operating system is supported,
  - that the user has sudo rights (checked via '*os.geteuid()*' or system info),
  - and that the selected tools are available in the package manager (apt).

It then attempts to install each missing tool using subprocess commands like '*sudo apt install [tool] -y*'. The result for each tool (success or error) is captured and returned in a structured response. Any failures are handled gracefully and sent back to the frontend for display.



```
POST http://localhost:8000/tools/install
```

```
{
  "tools": ["sublist3r", "whois"],
  "password": "mgh_99"
}
```

```
{
    "results": {
        "sublist3r": "Reading package lists...\nBuilding dependency tree...\nReading state information...\nThe following packages were automatically installed and are no longer
            required:\n  libllvm17t64 python3-netifaces\nUse 'sudo apt autoremove' to remove them.\nThe following additional packages will be installed:\n
            python3-dnspython\nSuggested packages:\n  python3-trio python3-aioquic python3-h2 python3-httpx python3-httpcore\nThe following NEW packages will be installed:\n
            python3-dnspython sublist3r\n0 upgraded, 2 newly installed, 0 to remove and 18 not upgraded.\nNeed to get 805 kB of archives.\nAfter this operation, 3,077 kB of
            additional disk space will be used.\nGet:1 http://gb.archive.ubuntu.com/ubuntu noble/main amd64 python3-dnspython all 2.6.1-1ubuntu1 [163 kB]\nGet:2 http://gb.archive.
            ubuntu.com/ubuntu noble/universe amd64 sublist3r all 1.1-4 [641 kB]\nFetched 805 kB in 0s (2,534 kB/s)\nSelecting previously unselected package python3-dnspython.\n
            (Reading database ... \n(Reading database ... 5%\n(Reading database ... 10%\n(Reading database ... 15%\n(Reading database ... 20%\n(Reading database ... 25%\n(Reading
            database ... 30%\n(Reading database ... 35%\n(Reading database ... 40%\n(Reading database ... 45%\n(Reading database ... 50%\n(Reading database ... 55%\n(Reading
            database ... 60%\n(Reading database ... 65%\n(Reading database ... 70%\n(Reading database ... 75%\n(Reading database ... 80%\n(Reading database ... 85%\n(Reading
            database ... 90%\n(Reading database ... 95%\n(Reading database ... 100%\n(Reading database ... 192578 files and directories currently installed.)\nPreparing to
            unpack .../python3-dnspython_2.6.1-1ubuntu1_all.deb ...\nUnpacking python3-dnspython (2.6.1-1ubuntu1) ...\nSelecting previously unselected package sublist3r.\nPreparing
```

Illustration 5.3.3 - Postman request to /tools/list showing tool installation for sublist3r and whois

- *GET /system/check*

  This route is used to return information about the current system. It includes:

  - the operating system type (Linux or Windows),
  - whether sudo is available,
  - and whether the system allows local execution or needs to use the remote backend.

  The frontend uses this information to:

  - decide whether the "Install Tools" feature should be enabled,
  - display warnings if the system is not supported,
  - and choose between local or remote execution pathways.

3. <u>User Role Verification</u>

- GET /auth/me

  This endpoint verifies the Firebase token sent in the *Authorization* header of the request. It decodes the token using the '*firebase_admin*' SDK and returns user information including:

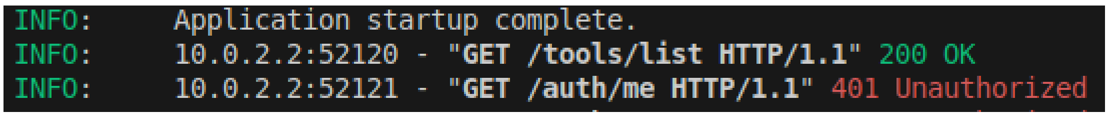
  - email address
  - unique Firebase ID,
  - and a custom flag '*is_superuser*' based on a predefined list of authorized users.

  This endpoint is called immediately after the user logs in from the frontend. The response determines what the user can see and do in the interface:

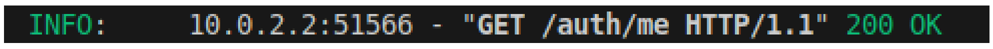
- If '*is_superuser: true*', the user can install tools, execute sudo-level commands, and use the full local system.
- If false, the user is limited to non-sudo tools and remote execution.

The token verification is performed using the method '*auth.verify_id_token(token)*', which raises an exception if the token is missing, expired, or invalid.

If the token is invalid or missing, the backend returns an HTTP 401 Unauthorized response. This mechanism ensures that only authenticated users can access execution-related endpoints, and that dangerous actions are limited to trusted accounts.



```
INFO:     Application startup complete.
INFO:     10.0.2.2:52120 - "GET /tools/list HTTP/1.1" 200 OK
INFO:     10.0.2.2:52121 - "GET /auth/me HTTP/1.1" 401 Unauthorized
```



```
INFO:     10.0.2.2:51566 - "GET /auth/me HTTP/1.1" 200 OK
```

Illustration 5.3.4 - Postman request to /auth/me with a valid and invalid token and superuser flag in the response

Further technical details on the Firebase authentication system and backend integration can be found in *Section 5.7.*

Together, these endpoints form the operational core of Glitch_ai. Their modular structure, strict validation, and role-aware design allow the system to adapt securely to different user environments, supporting flexible and robust execution workflows.

Additional code samples, including complete route definitions and supporting functions, can be found in: https://cseegit.essex.ac.uk/24-25-ce901-su-ce902-sp/24-25_CE901-SU_CE902-SP_gonzalez_herrero

5.4. LLM Interpretation

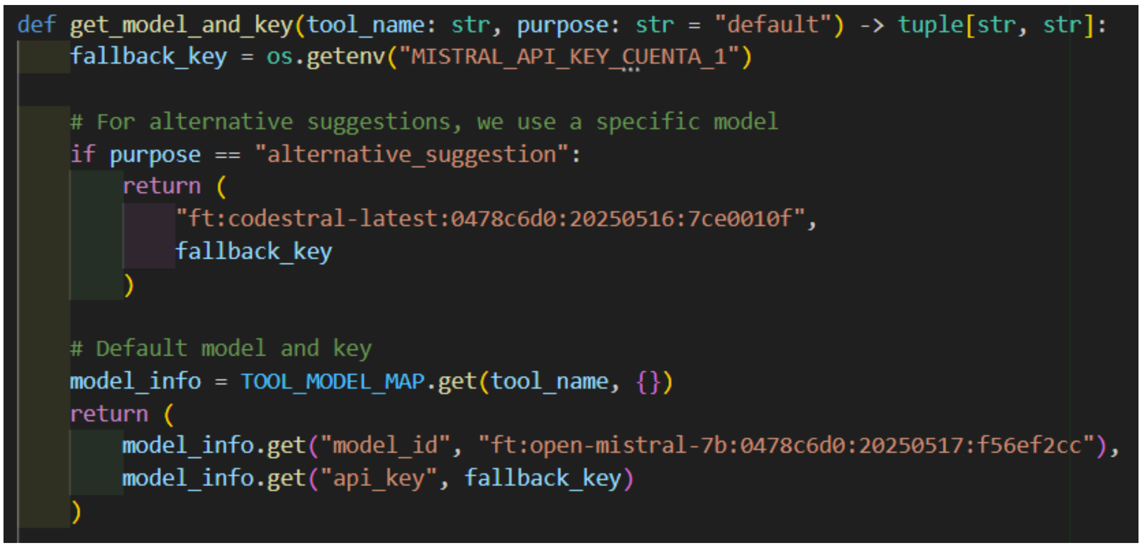
A key innovation in **Glitch_ai** is the integration of multiple **Large Language Models (LLMs)** to interpret tool output, suggest adaptive actions in case of failure, and generate human-readable summaries. Instead of relying on a single general-purpose model, Glitch_ai dynamically routes each task to the most appropriate model, depending on the tool, task type, and context.

This is achieved through a **custom model selection mechanism** that assigns fine-tuned Mistral models, including **Mistral 7B**, **Mistral 3B**, **Mistral Nemo,** and **CodeStral**, to different functions in the system.

- **Model Selection Strategy**

Each tool is mapped to a specific model via the TOOL_MODEL_MAP configuration. For example:

- Nmap uses a fine-tuned version of **CodeStral** for alternative command suggestions.
- Nikto, Hydra and SQLmap use **Mistral Nemo** for vulnerability detection.
- Sublist3r, Whois and Dmitry use **Mistral 7B** for passive reconnaissance summaries.
- Wpscan and WhatWeb uses **Mistral 3B** to balance performance and output clarity.



```python
def get_model_and_key(tool_name: str, purpose: str = "default") -> tuple[str, str]:
    fallback_key = os.getenv("MISTRAL_API_KEY_CUENTA_1")

    # For alternative suggestions, we use a specific model
    if purpose == "alternative_suggestion":
        return (
            "ft:codestral-latest:0478c6d0:20250516:7ce0010f",
            fallback_key
        )

    # Default model and key
    model_info = TOOL_MODEL_MAP.get(tool_name, {})
    return (
        model_info.get("model_id", "ft:open-mistral-7b:0478c6d0:20250517:f56ef2cc"),
        model_info.get("api_key", fallback_key)
    )
```

Illustration 5.4.1 - Code fragment of get_model_and_key() in mistral_api.py, showing tool-to-model mapping and fallback selection

Additionally, specific task types such as **executive summaries**, **technical analysis**, or **adaptive fallbacks** are routed through the model_router.py, which selects the optimal model for the given objective. This strategy allows Glitch_ai to:

- Reduce API latency and cost by using smaller models where appropriate.
- Improve quality and focus in LLM responses.
- Enable model fine-tuning and specialization for specific tool outputs.

- **<u>Adaptive Execution with LLMs</u>**

Glitch_ai includes an adaptive execution flow that intelligently responds to tool failures. Rather than immediately discarding a command that fails, the system attempts a sequence of progressively smarter fixes. The process is as follows:

1. **First attempt:** The original command is executed as-is. If it succeeds (‘*returncode == 0*’), the result is stored and no fallback is triggered.
2. **Second attempt (automatic retry):** If the first attempt fails (e.g., due to a timeout, syntax error, or inaccessible destination), the same command is re-executed once more without modification. This is done to detect transient problems or deadlock conditions.
3. **Third attempt (adaptive fix):** If the second attempt also fails, the system checks whether there are known issues associated with the tool and output. For example:

   - If Hydra fails with a timeout, the system appends ‘*-t 4*’ to reduce thread concurrency.
   - If Nmap returns “0 hosts up”, it adds ‘*-Pn*’ to skip host discovery. These adaptive fixes are hardcoded in the file *adaptive_fixes.py*.

4. **Fourth attempt (final fallback fix):** If the adaptive fix also fails, a second layer of tool-specific fallback is attempted. For example, changing WHOIS servers if the default one times out.
5. **Fifth attempt (LLM Suggestion):** If all previous attempts fail, Glitch_ai invokes the LLM to suggest an alternative command. The model receives:

   - The original command.
   - The target.
   - The full error output.

   Using this context, the model generates a corrected version of the command, which is then validated and executed.

The model used at this stage is typically a fine-tuned version of **CodeStral**, selected **specifically for command generation** and recovery tasks.

Each failed command and its output are stored in a structured log (*fallback_suggestions.jsonl*) for debugging and future improvements. This step-by-step approach increases the resilience of the system and demonstrates the integration between automation and intelligent reasoning.

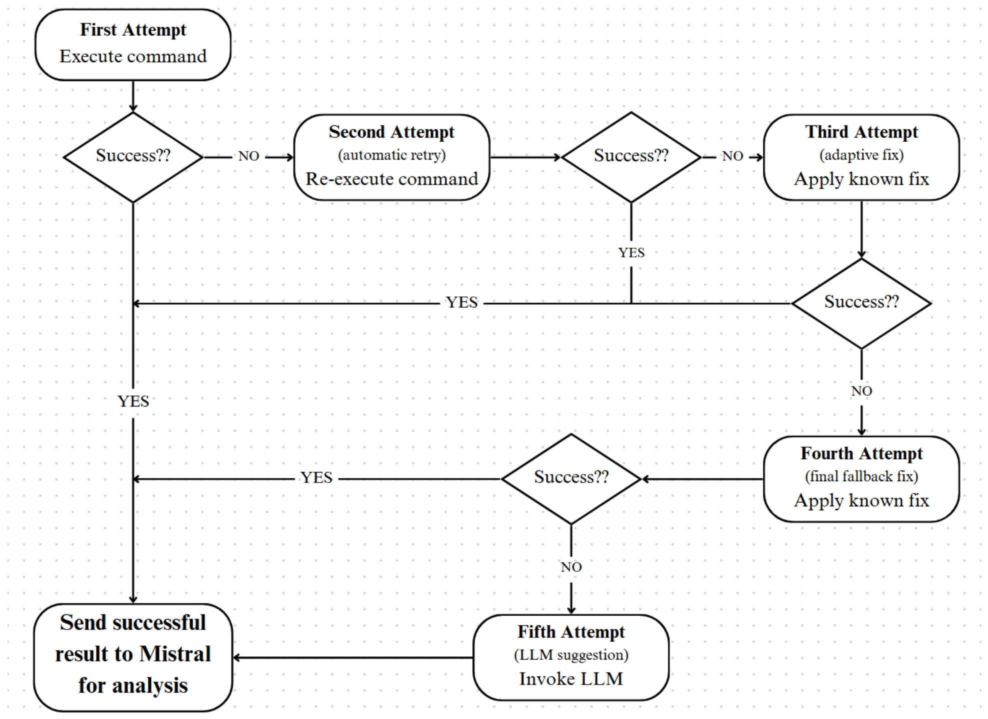


Illustration 5.4.2 - Adaptive execution flow in Glitch_ai

- <u>**Multi-level Interpretation of Scan Results**</u>

Once the analysis is complete, whether successful or partially successful, Glitch_ai uses its LLM module to generate multiple layers of interpretation, each adapted to a specific audience or use case. This multi-level strategy ensures that the result is useful for security analysts and understandable for stakeholders with less technical knowledge. The LLM module produces four types of results:

1. <u>Technical Analysis (per tool)</u>

For each tool executed, the **raw output is sent to the LLM** along with contextual instructions that guide the model to:

- Identify what was scanned or tested.
- Extract any potential findings or abnormal results.
- Explain the output in technically accurate terms.

These analyses are written using cybersecurity terminology and are intended for technical users, such as red team analysts or system administrators.

The model used for this task is usually **Mistral 7B or Mistral Nemo**, depending on the tool and the complexity of the result. For example, Nemo is preferable for web-centric tools such as Nikto or SQLmap due to its adjustment to unstructured records and vulnerability patterns.

2. Mitigation Advice

If a **vulnerability** or **risky configuration is detected** (either directly or by inference), the LLM is asked to provide advice on how to mitigate it. These requests ask the model to:

- Propose specific mitigations.
- Prioritise actions (e.g., patching versus decommissioning).
- Link technical findings to industry best practices.

This block is designed to be concise, practical, and immediately applicable by security teams.

Mitigation recommendations are managed by **Mistral 7B or Nemo**, depending on the source of the tool's output. The system ensures that ambiguous outputs (e.g., 'the host appears vulnerable') are not overstated by applying additional validation rules before allowing the LLM to interpret the risk.

3. Executive Summary

To make the **results accessible** to stakeholders without technical knowledge (e.g., managers, legal or compliance teams), a simplified executive summary is generated. This summary includes:

- A high-level overview of what has been tested,
- The most relevant results,
- An overall risk assessment (e.g., 'low risk,' 'moderate risk,' etc.).

The guidelines for this task are intentionally non-technical and are managed by **Mistral 3B**, which offers a good balance between speed, readability, and summary performance.

The summaries are toned appropriately, and if no vulnerabilities have been found, the summary reflects this positively. If there are critical findings, they are described in simple terms, with attention focused on clarity and priority.

4. Non-technical Explanation

Finally, an additional explanation is generated to help bridge the gap between the tool's raw results and human understanding. These explanations are designed for users with minimal knowledge of cybersecurity and often include analogies, comparisons, or simplified terminology.

This block is useful in training environments or when presenting reports to clients. It typically reuses elements from the executive summary but slightly expands on the significance of the results.

- **Model Routing**

Each of the outputs above is generated using its own custom prompt, defined in the '*prompt_templates.py'* module. The system uses the *model_router.py* to route each task to the most appropriate model:

Task	Model Used
Technical Analysis	Mistral 7B / Nemo
Mitigation Advice	Mistral 7B / Nemo
Executive Summary	Mistral 3B
Alternative Suggestions	CodeStral
Target Explanation	Mistral 3B

Table 5.4.1 - Model used per task

- **API Integration**

All interactions with the LLMs in Glitch_ai are performed through the '*query_mistral_model()*' function, which acts as a unified interface to Mistral's API. This function is responsible for constructing the HTTP POST request, embedding the prompt, and managing temperature, token limits, and authentication via the appropriate API key.

The system dynamically sets the temperature depending on the nature of the task:

- For creative or adaptive suggestions (e.g., command alternatives), a **temperature of '*0.7*'** is used to allow more flexibility and diverse responses.
- For deterministic outputs such as executive summaries or technical explanations, the temperature is lowered to '*0.3*' to reduce variation and ensure consistency across reports.

The maximum number of tokens is capped at 1024 to avoid excessive generation or truncated outputs. Prompts are also trimmed or compacted when necessary to ensure the full input fits within the token limit.

5.5. Security Tools Integration

Glitch_ai integrates a selected set of nine penetration testing tools that cover different phases of the security testing workflow, including reconnaissance, vulnerability detection, and exploitation. Each tool has been integrated into the backend through controlled thread execution and is dynamically managed based on user role, system conditions, and tool availability.

The backend distinguishes between tools that require local resources, those that require elevated privileges (sudo), and those that must be run remotely due to platform limitations (e.g., Windows). Tools are run sequentially or selectively, depending on user selection, and their results are captured, interpreted, and included in the final report.

Each tool integration includes:

- Input validation and parameter management.
- Execution control (local or remote).
- Error handling and adaptive fallback logic.
- Post-processing of results.

5.5.1. Tool by tool breakdown

Dmitry

Dmitry (Deepmagic Information Gathering Tool) is an open-source reconnaissance tool capable of collecting various types of information about a host, such as open ports, Whois lookups, Netcraft data and subdomain discovery [26].

In Glitch_ai, Dmitry is used as a **general-purpose recon tool**, especially when the target is a domain and the user wants to gather passive information quickly. It serves as a backup or complement to Sublist3r and Whois, offering redundancy in case other tools fail or return limited data.

The standard command template used is: '*dmitry -winsepfb [target]*'

The backend inserts the target (domain or IP) after sanitizing the input. Dmitry does not require protocol (*http/https*) and works directly with clean targets like *example.com* or *192.168.1.1*.

Dmitry may occasionally fail if:

- Netcraft access is blocked,
- The domain does not resolve,
- Port scanning is restricted by a firewall.

In such cases, the system logs the error, skips the failed section (e.g., Netcraft), and continues parsing the rest of the output. Since Dmitry does not rely on external APIs, it is generally stable in local environments.

When execution fails completely or partially, Glitch_ai queries the LLM (typically Mistral 7B or Nemo) to generate a useful summary from partial results or to explain why the tool might not have returned any data.

Nmap

Nmap is used in Glitch_ai to **network exploration and security auditing** [29]. It is available to all users regardless of role, as it does not require elevated privileges for basic scans.

When Nmap is selected, the system builds a predefined set of scan commands:

- *nmap -sS [target]*
- *nmap -O [target]*
- *nmap -sn [target]*
- *nmap -p 80,443 [target]*
- *nmap -sT [target]*

These commands are dynamically populated using the sanitized version of the target (IP or domain). If the user includes a URL with http:// or trailing slashes, the system removes them before execution to ensure compatibility with Nmap's syntax.

If Nmap returns an error such as '0 hosts up', Glitch_ai automatically applies the *-Pn* reserve flag to skip host detection and retry the scan. This adaptive correction is implemented directly in the backend and does not require user intervention.

In case of repeated failures, the LLM is invoked with the command and error message. The model (usually CodeStral) can suggest alternative scan parameters or highlight network-level issues (e.g., blocked ICMP or filtered ports).

The output of Nmap is parsed and formatted to:

- Extract open ports and services.
- Identify the operating system (when available).
- Summarize the results in natural language using Mistral 7B.

The integration also logs metadata such as duration, status, and return code, which are included in the final report structure.

Sublist3r

Sublist3r is a tool used to **enumerate subdomains of a given domain** using OSINT techniques (Open Source Intelligence). It queries multiple public sources to discover additional assets linked to the primary domain [30].

In Glitch_ai, Sublist3r is used during the reconnaissance phase to expand the surface of analysis by identifying hidden or forgotten subdomains. Its execution is triggered only if the target is a valid domain name, not an IP address.

If the user enters an IP address or a malformed domain, the frontend disables Sublist3r automatically. The backend also performs a second validation check to ensure execution is safe.

The default command used is: '*sublist3r -d [domain]*'

This command runs without elevated privileges and collects subdomains from a range of public sources. In Glitch_ai, the output is cleaned and stored for potential use in further scans (e.g., passing subdomains to Nmap or WhatWeb).

Sublist3r occasionally fails if public APIs are rate-limited, or if certain sources are unavailable (e.g., VirusTotal API errors). To improve resilience, Glitch_ai implements a multi-step fallback mechanism:

1. Retry Sublist3r with reduced sources (disabling DNSdumpster or VirusTotal).
2. Query SecurityTrails or Amass via remote API (if enabled).
3. If all fail, send the target and error message to the LLM (CodeStral), which may suggest:
   - alternative APIs,
   - typos in the domain name,
   - or identify that the domain is newly registered and lacks subdomain history.

All fallback results are merged with the original list (if partial output exists), and deduplicated before being returned.

Whois

WHOIS is a widely used Internet protocol for **searching databases that store information about registered users of domain names, IP addresses, and autonomous systems**. It helps you find the owner's contact details, registration dates, and name server information [33].

In Glitch_ai, Whois is used to gather contextual information about the target domain. It helps identify:

- Domain registration date and country,
- If the domain is managed by a known registrar,
- Whether there are signs of suspicious activity (e.g., very recent creation, anonymity).

This tool only works with domains, not IP addresses. If the target is an IP or an invalid TLD, the backend detects this and skips the execution, returning a message to the frontend.

The default command used is: '*whois [domain]*'

The tool is executed locally when available. On some Linux distributions, it requires installation via package manager ('*apt install whoi's*). If the tool is missing, users with superuser role can install it from the frontend using the integrated installer.

For certain TLDs (such as *.es*), Whois servers are not directly available. In those cases, Glitch_ai adds a message to the output pointing to external lookup services (e.g., dominios.es for *.es* domains).

Nikto

Nikto is an open-source web server scanner designed to **detect common vulnerabilities**, such as out-of-date software versions, incorrect configurations, exposed files, and insecure headers. It performs a thorough check for known issues and CVE-linked vulnerabilities on HTTP/HTTPS targets [28].

In Glitch_ai, Nikto is used to scan web targets for potential weaknesses in the application layer. Its integration requires specific handling, particularly regarding target validation and input formatting:

- The target must include the protocol (http:// or https://) for Nikto to function correctly.
- If the user provides a target without a protocol, the frontend shows a warning, and the backend prevents the command from being executed.
- This validation ensures that Nikto receives a fully qualified URL to avoid misfires and incomplete scans.

The default command template is, '*nikto -host [target]*'. This command is automatically adjusted depending on the user's input. If the scan fails with a known error, Glitch_ai attempts an adaptive fix, such as:

- Forcing SSL scanning (*-ssl*).
- Disabling redirects (*-no404*).
- Or switching the protocol if misconfigured.

If the scan fails after adaptive attempts, the LLM is queried with the full command and error output. The model (typically Mistral Nemo) may suggest a protocol correction, alternative flags, or even propose switching to another tool like WhatWeb.

Once the scan completes, the output is parsed and structured before being passed to the LLM for summary generation.

SQLmap

SQLmap is an open-source penetration testing tool specialized in **detecting and exploiting SQL injection vulnerabilities** in web applications. It automates the process of identifying vulnerable parameters, extracting database information, and even taking full control of the backend database under certain conditions [31].

In Glitch_ai, SQLmap is used as a high-risk vulnerability scanner focused on web targets. Its integration requires strict input validation, as it expects a full URL with protocol and at least one injectable parameter

The default command template used is: '*sqlmap -u [target] –batch*'

Additional options are added based on tool configuration and user settings. For example:

- *--level 5 --risk 3* for more aggressive scans.
- *--tamper=space2comment* for bypassing simple filters.
- *--dbs* to list available databases (when testing depth is enabled).

The backend verifies that the target includes, a valid protocol (http:// or https://).

When all retry and adaptive mechanisms fail, the system sends the output and command to the LLM (typically CodeStral), which may:

- Identify missing injection points,
- Suggest specific tamper scripts,
- Reformat the URL.

WhatWeb

WhatWeb is a web fingerprinting tool that **identifies technologies used by websites**, such as server type, CMS platform, frameworks, JavaScript libraries, and more. It analyzes headers, HTML content, and scripts to detect patterns associated with known platforms [32].

In Glitch_ai, WhatWeb serves both as a complement and fallback to tools like Nikto and Wpscan. It is especially useful when:

- Nikto fails due to SSL issues or misconfiguration.
- The user wants to validate if the target is running a CMS (e.g., WordPress, Joomla).
- A quick scan is needed with low risk of interruption or detection.

The default command executed is: '*whatweb [target]*'

Additional flags such as *--no-errors* or *--log-verbose* may be added depending on verbosity settings and error control. The backend ensures the target includes a protocol (http/https). If not, the tool is disabled, and the frontend alerts the user. Unlike Nikto, WhatWeb can handle more flexible inputs but may return less structured results.

If WhatWeb detects a WordPress installation, that information can trigger the suggestion of launching Wpscan in a subsequent step.

WhatWeb also plays a fallback role:

- If Nikto fails after retries and adaptive fixes, Glitch_ai may launch WhatWeb instead to gather partial insight about the webserver.
- In such cases, the LLM is prompted with both outputs (failed Nikto and successful WhatWeb) and asked to compare or complete the interpretation.

If WhatWeb itself fails (rare), Glitch_ai may query the LLM for insight based on headers manually extracted from previous scans.

Wpscan

Wpscan is a specialized **security scanner for WordPress websites**. It detects known vulnerabilities in core WordPress versions, themes, and plugins, and can also identify weak or exposed credentials and misconfigurations [34].

In Glitch_ai, Wpscan is reserved for superusers ("S") and executed only if the user provides:

- A valid URL that includes the protocol (http:// or https://), and
- Confirmation that the target runs WordPress (e.g., manually or based on prior WhatWeb results).

Due to its installation complexity and system-level dependencies, Wpscan is executed exclusively via the remote backend hosted on a Linux VM with all required packages pre-installed.

The default command template is: *wpscan --url [target] --disable-tls-checks*

Additional options may be added based on user configuration:

- *--enumerate p* to list vulnerable plugins.
- *--api-token [token]* for deeper vulnerability data (optional).
- *--random-user-agent* to avoid detection/blocking.

All commands are constructed on the backend, and the target is validated to ensure it contains a protocol. If the URL is invalid or points to a non-WordPress site, the tool is skipped, and a message is sent to the frontend.

If Wpscan fails (due to SSL issues, invalid certificate, or remote blocking), Glitch_ai applies internal fixes such as:

- using *--disable-tls-checks*,
- switching to http if the site is using a self-signed certificate,
- and lowering verbosity.

If all retries fail, the command and output are passed to the LLM (CodeStral) to generate a corrected version or suggest alternatives.

Hydra

Hydra is an open-source Python framework that simplifies the development of research. It **systematically tests multiple username and password combinations** against a selected service to identify weak or default credentials [27].

In Glitch_ai, Hydra is integrated as part of the exploitation phase. Its execution is carefully managed due to the potential legal and ethical implications of brute force testing. By default, the system only allows Hydra to be used on explicitly selected targets and under the superuser role ('S') to prevent improper use.

The default command template is: '*hydra -l [username] -P [dictionary_path] [protocol]://[target]:[port]*'

Hydra integration in Glitch_ai includes a flexible password profile system, allowing users to choose from different brute-force intensities:

- Fast: a small list of common passwords for quick scans.
- Standard: the most used 500–1000 passwords.
- Deep: a longer dictionary.

The password profile is passed to the backend, which then selects the corresponding dictionary file and updates the command accordingly.

Hydra may fail for multiple reasons:

- Timeout or connection reset,
- Service unresponsive,
- Authentication rate-limiting,
- Invalid protocol-port combinations.

Glitch_ai handles this with:

1. Retry logic with timeout reduction (-t 4).
2. Adaptive fix by trying a different port (e.g., 22 → 2222).
3. Final fallback: sending the error message to the LLM for analysis.

The model used for adaptive suggestions is CodeStral, which receives the failing command and error output. It may propose:

- Adding a -s flag for port,
- Switching to a different protocol (e.g., from ssh to ftp),
- Recommending a lower concurrency to avoid lockouts.

5.5.2. Execution Logic and Platform Adaptation

As seen throughout the tool integrations, Glitch_ai dynamically adjusts its execution logic based on the operating system, user role, and the availability of each tool.
Some tools, such as Nmap or WhatWeb, can run locally without restrictions.

Others, such as Wpscan or certain Hydra configurations, require either sudo privileges or a Linux-based environment and are therefore routed through the remote backend.
This adaptive configuration enables Glitch_ai to maintain reliable operation despite host system limitations.
The platform integrates established penetration testing tools within a workflow that is both flexible and automated.

5.5.3. Output Processing and Security Controls

Each tool integrated into **Glitch_ai** produces raw output through the command line usually via '*stdout*' and '*stderr*'. To ensure that these results are usable, readable, and secure, the system applies a standardized processing pipeline and several layers of execution control.

Every tool execution is wrapped using '*subprocess.run()*' with output redirection. The backend captures:

- Raw output (stdout),
- Error messages (stderr),
- Exit code, and
- Execution metadata (duration, target, timestamp, etc.).

Before sending the output to any LLM, the backend uses the *prompt_templates.py* module to:

- Attach contextual information (tool, options, error or success),
- Apply summarization prompts based on role (technical, executive, mitigation),
- Avoid redundant or noisy inputs.

If the output is incomplete or contains warnings, this is explicitly stated in the prompt to prevent hallucinated interpretations by the model.

And to protect the system and the user, Glitch_ai includes multiple security measures:

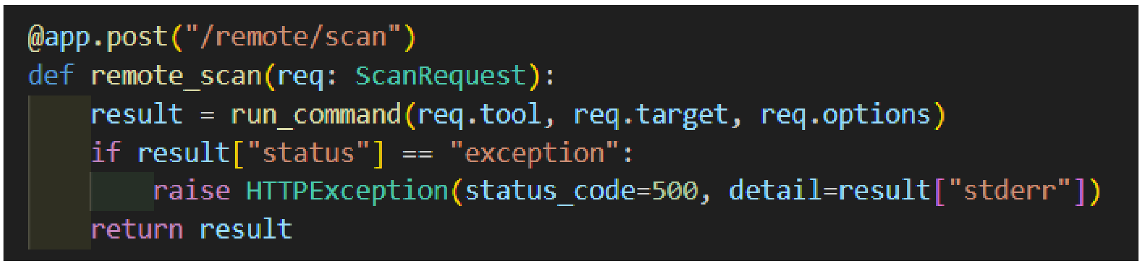
- Target sanitization: URLs and domains are cleaned before command insertion to prevent injection or formatting issues.
- Whitelisting of tools: Only predefined tools and commands can be executed. Arbitrary or dynamic execution is explicitly blocked.
- Role-based access: Only users with the “S” role can trigger sudo-level actions or install tools from the interface.
- Fallback prevention: Suggestions generated by the LLM are validated before execution to avoid executing malformed or unsafe commands.
- Timeouts and limits: Each command has a maximum execution time and resource cap to prevent system overload.

5.6. Remote backend

Glitch_ai includes a dedicated remote backend designed to run tools on a secure Linux-based virtual machine when the local environment is incompatible or has restricted permissions for setting up applications.

This architecture ensures full functionality even when users run Glitch_ai from a system that does not support local execution, such as a Windows machine without sudo or a device lacking certain tools.

The remote backend is implemented as a separate FastAPI service, hosted on a virtual machine with all basic security tools pre-installed. It exposes a single POST endpoint.



```python
@app.post("/remote/scan")
def remote_scan(req: ScanRequest):
    result = run_command(req.tool, req.target, req.options)
    if result["status"] == "exception":
        raise HTTPException(status_code=500, detail=result["stderr"])
    return result
```

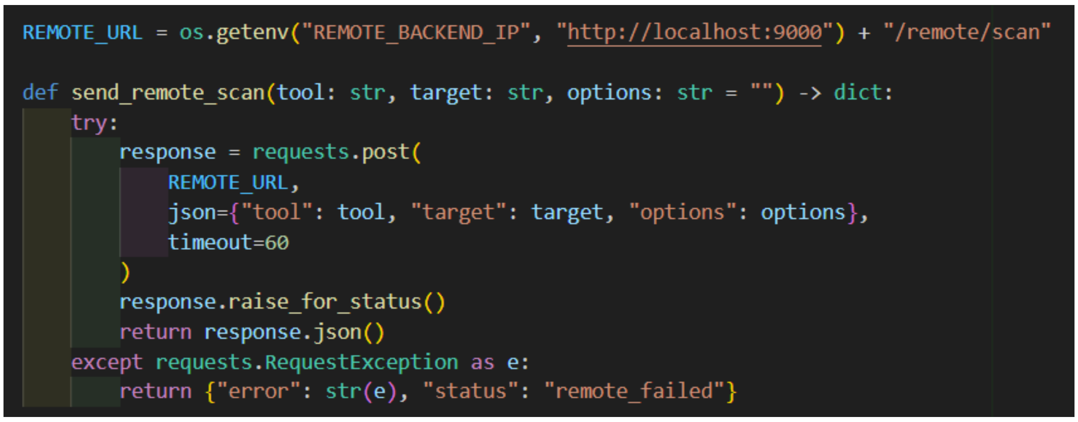
Illustration 5.6.1 - FastAPI implementation of the /remote/scan endpoint used for executing tools remotely

This endpoint receives the analysis request in JSON format, including:

- Tool name.
- Objective.
- Optional execution parameters.

It then calls a local function to execute the command on the Linux host and returns the result *(‘stdout’, ‘stderr’, ‘status’*, and *‘executed command’*) to the main backend.

For integration with the main backend, there is a module called remote_client.py that manages API calls to the remote backend. It uses the requests library to send the scan request via HTTP and return the result.



```
REMOTE_URL = os.getenv("REMOTE_BACKEND_IP", "http://localhost:9000") + "/remote/scan"

def send_remote_scan(tool: str, target: str, options: str = "") -> dict:
    try:
        response = requests.post(
            REMOTE_URL,
            json={"tool": tool, "target": target, "options": options},
            timeout=60
        )
        response.raise_for_status()
        return response.json()
    except requests.RequestException as e:
        return {"error": str(e), "status": "remote_failed"}
```

Illustration 5.6.2 - Function send_remote_scan() function in the main backend. Sends the tool, target, and options to the remote backend via a POST

The remote URL is dynamically loaded using environment variables (REMOTE_BACKEND_IP), allowing for flexible deployment across different networks or servers.

If the remote backend fails or becomes unavailable, a clear error is returned to the main backend so that it can be resolved appropriately and the user informed.

To ensure secure remote execution:

- Only predefined tools are supported; arbitrary commands are not allowed.
- The command is created from validated inputs and executed in a restricted shell context.
- The virtual machine is isolated from the main system, reducing the impact of tool-level errors or incorrect setups.

This remote architecture not only allows tools to be run in restricted environments, but also provides several operational advantages:

- **Cross-platform compatibility**: users running Glitch_ai on Windows or macOS, systems that often lack native compatibility with tools such as Nmap or Wpscan, can still access all features through the remote backend.
- **Privilege abstraction**: the need for local sudo privileges is eliminated, as tools requiring elevated access (e.g., Wpscan or certain Hydra scans) run within a secure, preconfigured environment.
- **Centralised maintenance**: Since all tools are pre-installed on the remote virtual machine, updates and changes to the environment can be managed in a single location, reducing configuration difficulties for users.
- **Execution resource**: If local execution fails due to missing tools, system incompatibilities, or permission issues, Glitch_ai automatically redirects execution to the remote backend, ensuring that scans can continue without user intervention.

Together, these advantages make the remote backend a fundamental component of Glitch_ai's adaptive architecture, enabling reliable penetration testing across diverse platforms and user roles.

5.7. Firebase Authentication

To manage user access and enforce role-based permissions, Glitch_ai integrates Firebase Authentication, a secure and scalable identity management platform developed by Google.

Firebase Authentication provides built-in support for email and password login, token-based session management, and secure access control, enabling developers to focus on application logic without needing to build an authentication system from scratch [45].

In Glitch_ai, this integration enables the platform to distinguish between regular users (“U”) and superusers (“S”), controlling which actions each user can perform across the system.

5.7.1. Frontend Integration

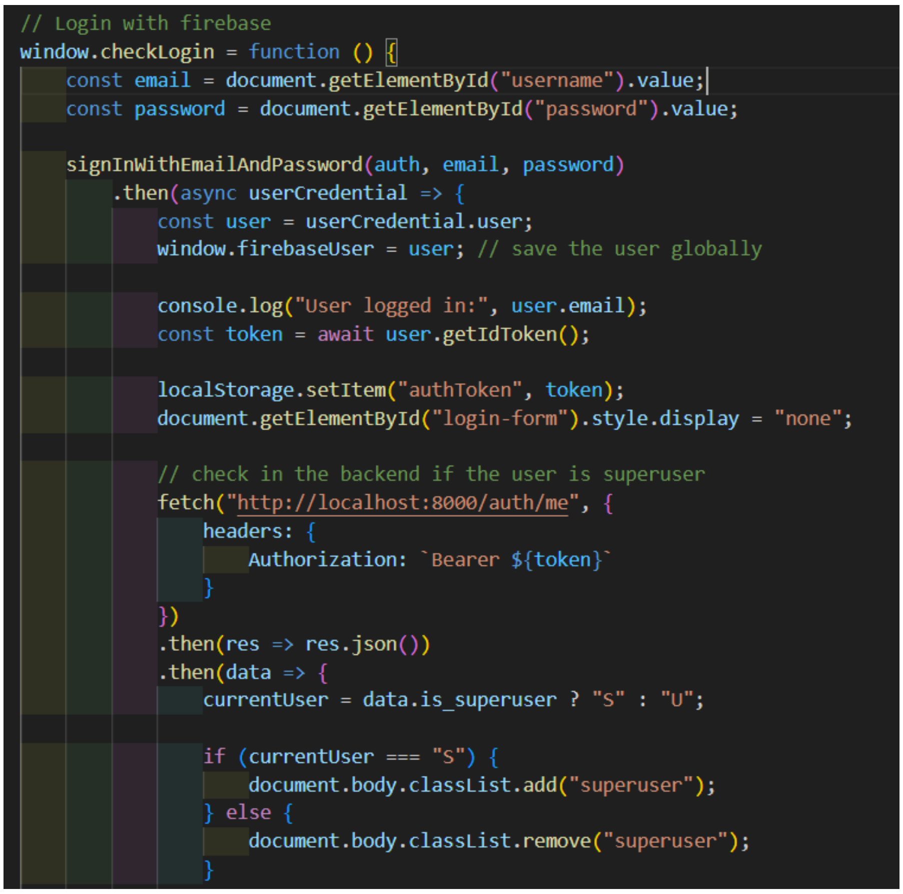
In the frontend interface, users log in using their email and password via Firebase's authentication SDK. Upon successful login, Firebase generates a JSON Web Token (JWT), which is stored locally in the browser and attached to all backend requests that require authentication.

The authentication process is implemented through the '*checkLogin()*' function, which:

- Sends the user's credentials to Firebase,
- Retrieves the corresponding JWT,
- stores it in '*localStorage'*,
- and sends it to the backend to verify the user's role.

Since the user's role is not included in the token, the frontend makes a follow-up request to the backend endpoint */auth/me*, which returns whether the user is a superuser ("S") or a standard user ("U"). Based on this role, the UI dynamically adapts:

- Users logged in as "U" see a limited dashboard (e.g., no tool installation).
- Superusers ("S") gain access to administrative features, including the ability to run *sudo* commands and manage missing tools.

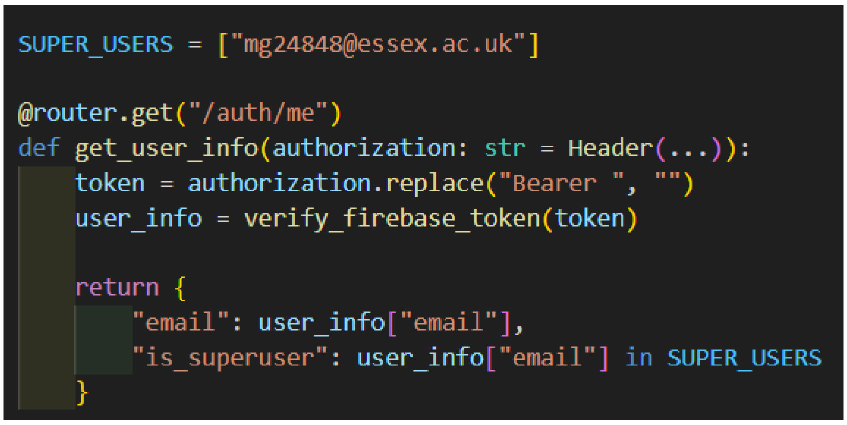


```javascript
// Login with firebase
window.checkLogin = function () {
    const email = document.getElementById("username").value;
    const password = document.getElementById("password").value;

    signInWithEmailAndPassword(auth, email, password)
        .then(async userCredential => {
            const user = userCredential.user;
            window.firebaseUser = user; // save the user globally

            console.log("User logged in:", user.email);
            const token = await user.getIdToken();

            localStorage.setItem("authToken", token);
            document.getElementById("login-form").style.display = "none";

            // check in the backend if the user is superuser
            fetch("http://localhost:8000/auth/me", {
                headers: {
                    Authorization: `Bearer ${token}`
                }
            })
            .then(res => res.json())
            .then(data => {
                currentUser = data.is_superuser ? "S" : "U";

                if (currentUser === "S") {
                    document.body.classList.add("superuser");
                } else {
                    document.body.classList.remove("superuser");
                }
```

Illustration 5.7.1 - checkLogin() function in the frontend. It manages Firebase authentication and role-based UI adaptation

5.7.2. Backend Role Verification

All backend-protected routes that require role-based decisions begin by verifying the Firebase token using the '*auth.py*' module.



```python
SUPER_USERS = ["mg24848@essex.ac.uk"]

@router.get("/auth/me")
def get_user_info(authorization: str = Header(...)):
    token = authorization.replace("Bearer ", "")
    user_info = verify_firebase_token(token)

    return {
        "email": user_info["email"],
        "is_superuser": user_info["email"] in SUPER_USERS
    }
```

Illustration 5.7.2 - Endpoint /auth/me used to verify Firebase tokens and assign user roles

The key endpoint is '*/auth/me*', which receives the token from the frontend (via the *Authorization: Bearer <token>* header), decodes it, validates it with Firebase public keys, and checks whether the user's email address appears in the predefined list of superusers.

If the token is invalid or missing, the request is rejected with a 401 Unauthorised response. If it is valid, the backend returns the user's email address and role, which the frontend uses to adapt the interface accordingly.

5.8. Reporting Module (JSON & PDF)

At the end of every scan session, Glitch_ai provides users with the option to generate and download a structured report. These reports are available in two formats:

- **JSON**, for structured, machine-readable data that can be programmatically parsed or integrated into dashboards.
- **PDF**, for professional, human-readable documents designed for audits, documentation, or executive review.

Both formats are generated from the same backend data, ensuring consistency between technical output and high-level summaries.

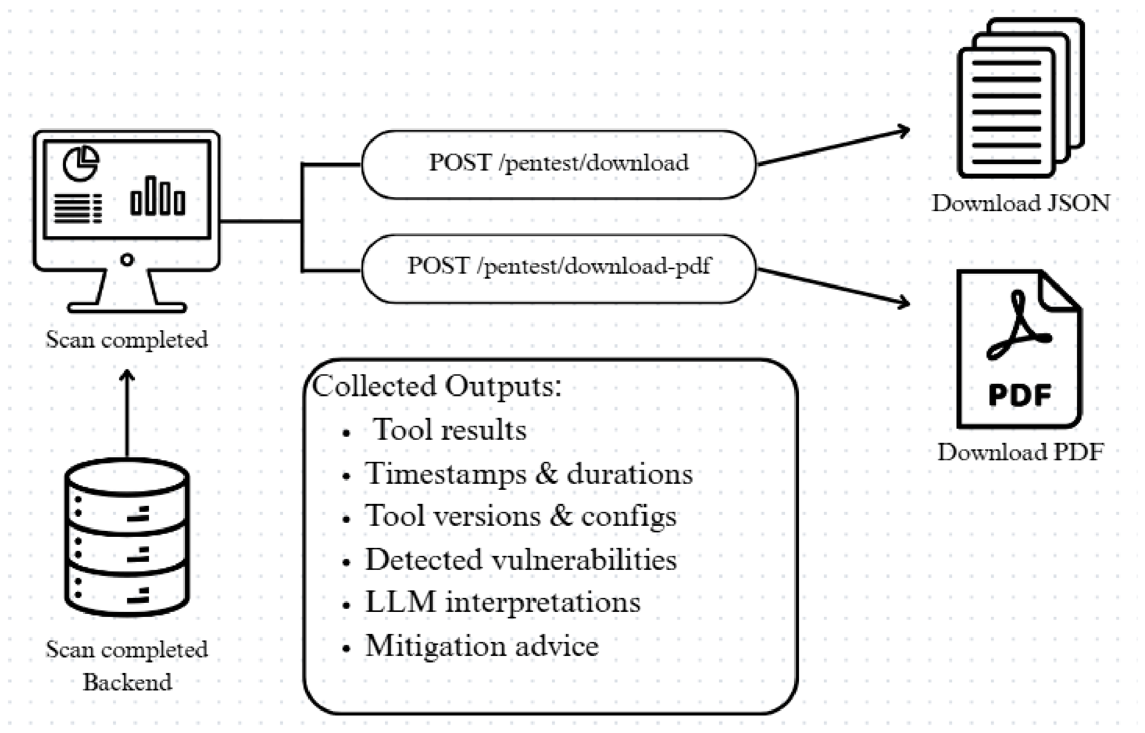
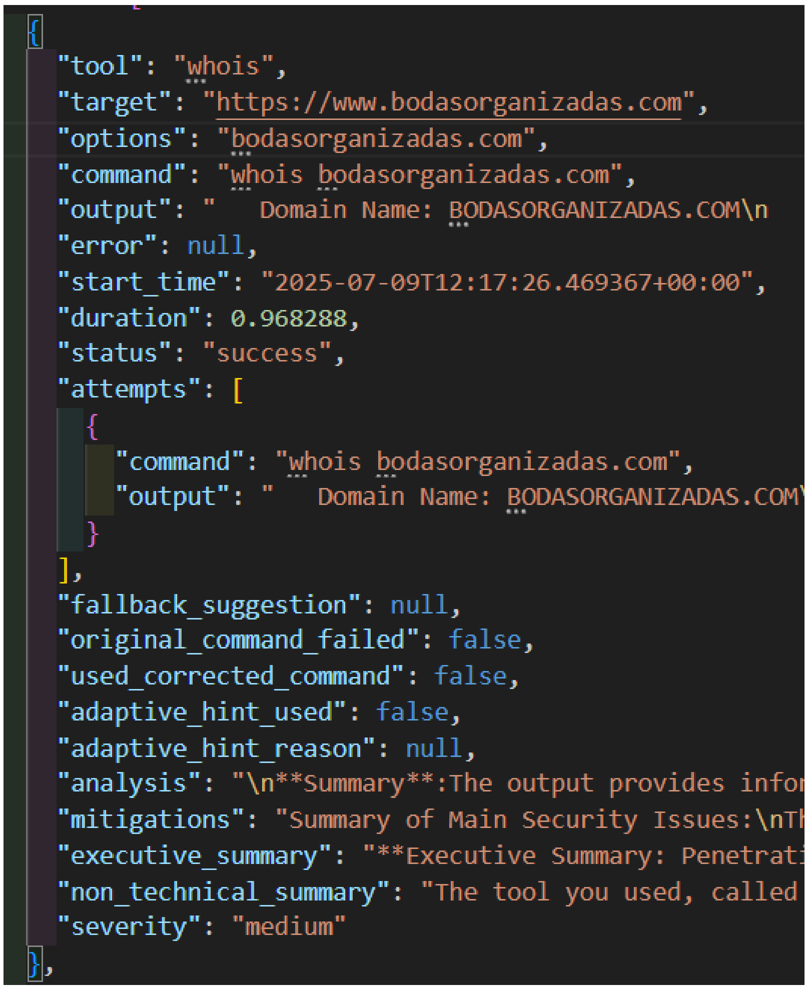


Illustration 5.8.1 - Diagram of the report generation flow in Glitch_ai, showing how JSON and PDF reports are created after scan completion

The JSON report is an unprocessed, hierarchical data structure intended for developers, analysts, or data dashboards. Each entry includes:

- Tool name and command executed.
- Target scanned.
- Standard output and error (if any).
- Execution status and duration.

- Optional LLM interpretation block (summary, mitigation, risk level).



```
{
  "tool": "whois",
  "target": "https://www.bodasorganizadas.com",
  "options": "bodasorganizadas.com",
  "command": "whois bodasorganizadas.com",
  "output": "   Domain Name: BODASORGANIZADAS.COM\n
  "error": null,
  "start_time": "2025-07-09T12:17:26.469367+00:00",
  "duration": 0.968288,
  "status": "success",
  "attempts": [
    {
      "command": "whois bodasorganizadas.com",
      "output": "   Domain Name: BODASORGANIZADAS.COM\
    }
  ],
  "fallback_suggestion": null,
  "original_command_failed": false,
  "used_corrected_command": false,
  "adaptive_hint_used": false,
  "adaptive_hint_reason": null,
  "analysis": "\n**Summary**:The output provides infor
  "mitigations": "Summary of Main Security Issues:\nTh
  "executive_summary": "**Executive Summary: Penetrati
  "non_technical_summary": "The tool you used, called
  "severity": "medium"
},
```

Illustration 5.8.2 - JSON report structure generated after a scan

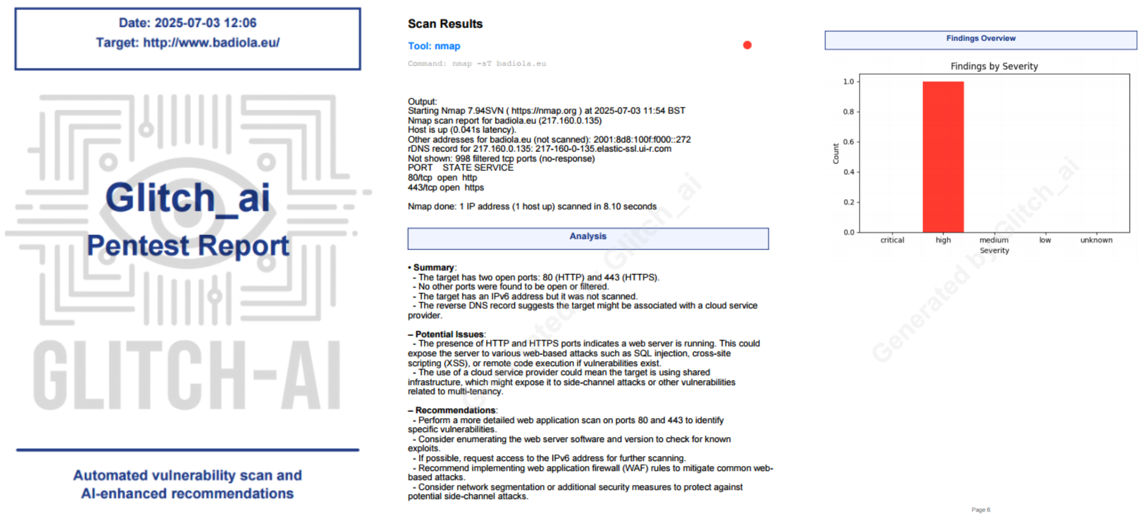
This format allows users to run post-processing scripts or load results into security dashboards for long-term tracking and correlation.

And for the PDF report is designed for professional presentation and non-technical audiences. It is structured into the following sections:

1. **Cover page** (title, date, target).
2. **Scan summary** (tools used, duration, system type).
3. **Results per tool**, each with: Detected vulnerabilities or outputs, Interpretations by the LLM and Recommended mitigations.
4. **Executive summary**, written in natural language, adapted to stakeholders without cybersecurity expertise.

The PDF is styled with clear headings, color-coded severity levels, and structured tables to improve readability.

PDFs are generated using the '*reportlab*' library in Python and are available for download directly after the scan completes.


Date: 2025-07-03 12:06
Target: http://www.badiola.eu/

Glitch_ai
Pentest Report
GLITCH-AI

Automated vulnerability scan and
AI-enhanced recommendations

Scan Results

Tool: nmap

```
Command: nmap -sT badiola.eu
```

```
Output:
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-07-03 11:54 BST
Nmap scan report for badiola.eu (217.160.0.135)
Host is up (0.041s latency).
Other addresses for badiola.eu (not scanned): 2001:8d8:100f:f000::272
rDNS record for 217.160.0.135: 217-160-0-135.elastic-ssl.ui-r.com
Not shown: 998 filtered tcp ports (no-response)
PORT    STATE SERVICE
80/tcp  open  http
443/tcp open  https

Nmap done: 1 IP address (1 host up) scanned in 8.10 seconds
```

Analysis

• **Summary**:
 - The target has two open ports: 80 (HTTP) and 443 (HTTPS).
 - No other ports were found to be open or filtered.
 - The target has an IPv6 address but it was not scanned.
 - The reverse DNS record suggests the target might be associated with a cloud service provider.

– **Potential Issues**:
 - The presence of HTTP and HTTPS ports indicates a web server is running. This could expose the server to various web-based attacks such as SQL injection, cross-site scripting (XSS), or remote code execution if vulnerabilities exist.
 - The use of a cloud service provider could mean the target is using shared infrastructure, which might expose it to side-channel attacks or other vulnerabilities related to multi-tenancy.

– **Recommendations**:
 - Perform a more detailed web application scan on ports 80 and 443 to identify specific vulnerabilities.
 - Consider enumerating the web server software and version to check for known exploits.
 - If possible, request access to the IPv6 address for further scanning.
 - Recommend implementing web application firewall (WAF) rules to mitigate common web-based attacks.
 - Consider network segmentation or additional security measures to protect against potential side-channel attacks.

Illustration 5.8.3 - PDF report excerpt showing executive summary and tool-specific result

- **Format Comparison and Use Cases**

To highlight the differences between the two output formats and guide their use in different contexts, the following table compares the key features of JSON and PDF reports generated by Glitch_ai. It describes their format, target audience, automation potential, and presentation style.

Feature	**JSON**	**PDF**
Format type	Machine-readable (raw data)	Human-readable (formatted)
Ideal audience	Developers, analysts	Executives, clients, auditors
Supports automation	Yes	No
Supports visual presentation	No	Yes
Includes LLM summaries	Yes	Yes
Styling and structure	Plain text / key-value	Structured layout with sections

Table 5.8.1 - Comparison between the JSON and PDF formats generated by Glitch_ai

The availability of both JSON and PDF reports ensures that Glitch_ai delivers comprehensive and accessible results for diverse user groups. This reporting capability supports transparent, practical, and adaptable security testing throughout the assessment process.

6. Evaluations and Results

This chapter explains how Glitch_ai was tested and evaluated to ensure that it works as planned and meets its goals. It outlines the main types of tests used—unit, integration, and validation—and describes the key performance metrics, including accuracy, false positives, and execution time. The results are then discussed in context, comparing Glitch_ai to traditional penetration testing tools, emphasizing both strengths and areas for potential improvement.

6.1. Testing Methodology

To evaluate the reliability, adaptability, and accuracy of Glitch_ai, a structured, multi-level testing methodology was applied. The process was divided into three complementary stages: **unit testing**, **integration testing,** and **validation testing**, each focusing on different aspects of the system architecture.

Unit Testing

Unit tests focus on independently verifying the smallest, most testable parts of an application, helping to detect errors early and ensure correctness at the component level [7].

Tests were performed on isolated backend modules using simulated inputs and controlled scenarios. The main components tested included:

- The **command builder module**, responsible for dynamically generating CLI instructions based on tool selection, user role, and target input.
- The **role validation logic**, which applied permission-based restrictions to tool execution (e.g., sudo tools restricted to superusers).
- The **LLM output handler**, which analysed suggestions generated by the model and directed them to execution decisions or report summaries.

These tests ensured functional correctness under various input combinations, including invalid targets, missing parameters, and role discrepancies. The tests were implemented using **pytest**, a popular Python testing tool, to write and run the tests in a local development setup with version control to catch any issues [46].

Integration Testing

Integration testing ensures that different modules or services work together as expected and validates data consistency [47].

- **Frontend - Backend**: User selections in the user interface (e.g., selected tools, target input, report generation) were verified to trigger the appropriate API calls and receive the correct responses.
- **Backend - Remote backend**: It was confirmed that Glitch_ai correctly delegated the execution to the remote backend when local permissions or missing installations prevented local execution.
- **Backend - LLM module**: It was evaluated whether failed commands triggered contextual suggestions from Mistral or CodeStral, and whether the corresponding backup strategies were executed.

The test cases simulated real-world usage patterns, such as launching a penetration test as a user without sudo and without the necessary tools or requesting a report after several tool failures.

Validation Testing

Validation testing checks whether the system meets user expectations and functional requirements under realistic conditions [48].

This involved deploying Glitch_ai against intentionally vulnerable targets, including:

- **DVWA (Damn Vulnerable Web Application)**: A deliberately insecure web application developed for security professionals to test tools and techniques in a legal environment. It includes a wide range of common web vulnerabilities, such as SQL injection, XSS, and weak authentication mechanisms [49].
- Web servers with known misconfigurations.
- Dockerised testbeds containing open ports or weak login forms.

Each test replicated and end-to-end workflow path from tool selection to command execution, result interpretation, and report generation. Validation metrics included:

- Successful detection of known vulnerabilities.
- Proper fallback when tools failed (e.g., Sublist3r switching to SecurityTrails).

- Generation of readable and actionable JSON/PDF reports with mitigation advice.

Scenarios were executed using both **user (U)** and **superuser (S)** roles to validate permission logic and hybrid execution capabilities (local vs. remote).

6.2. Evaluation criteria

To measure how well Glitch_ai performs, we set up several evaluation metrics. These cover both how each part works on its own and how the whole system performs in real-world testing. We looked at the following areas:

- **Functional Accuracy:** Measures whether each component of the system behaves as expected under a variety of inputs and execution contexts. For instance, whether the correct command is constructed based on user role and target, or whether fallback logic triggers when required.
- **False Positives and Error Handling:** Evaluates the system's ability to minimise false positives, such as wrongly flagged vulnerabilities or tool errors, and its responsiveness to incorrect or ambiguous outputs. Particular attention is given to how Glitch_ai reacts to failed scans or incomplete data through its LLM-guided recovery logic.
- **Execution Time:** Captures the time taken from initiating a scan to generating the final report, including tool execution, LLM processing, and result formatting. This helps benchmark performance across different environments and tool configurations (e.g., local vs. remote backend).
- **Orchestration Robustness:** Assesses the system's ability to maintain workflow integrity when faced with incomplete configurations (e.g., missing tools or lack of sudo privileges). Successful rerouting to the remote backend or skipping unavailable steps is considered a positive outcome.
- **Report Quality and Interpretability:** Focuses on the clarity and usefulness of the output generated, both in JSON and PDF formats. This includes evaluating whether the mitigation suggestions and summaries are technically accurate, concise, and understandable by both technical and non-technical users.

These metrics provide a practical foundation for analysing system performance in the next section. Where relevant, thresholds or qualitative interpretations will be included to assess whether Glitch_ai meets the goals defined early in development.

6.3. Results and Analysis

This section presents the results of Glitch_ai testing, organized according to the primary evaluation metrics. The findings are based on realistic scenarios involving both standard and superuser roles.

Functional Accuracy

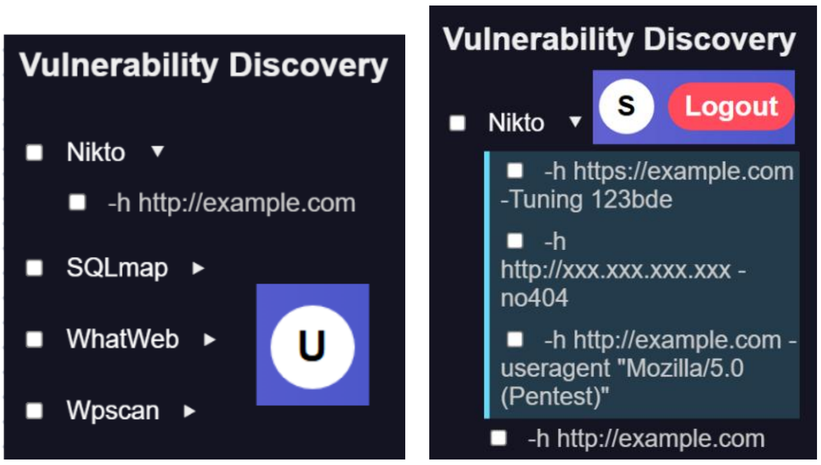
To evaluate the core reliability of Glitch_ai under expected usage scenarios, several functional accuracy tests were conducted. These focused on verifying whether the system behaves correctly under realistic operational conditions, with special emphasis on permission logic, remote execution fallback, and LLM-assisted error handling.

- **Test Case 1: Role-based Tool Visibility**

Objective: Verify that standard users (U) only see tools and command options appropriate to their role, while superusers (S) can access advanced tools and configurations that require elevated permissions.

Method: Launch the Glitch_ai interface first without logging in (User mode), and then again after logging in with superuser credentials. Observe the visibility and availability of tools such as *Nikto* and their corresponding commands in each case.

<u>Expected Result</u>: In User (U) mode, only non-sudo tools and basic commands should be visible and selectable. In Superuser (S) mode, advanced tools (e.g., *Nikto* full options) should be accessible, including additional command variations.



```
-h http://example.com
```

```
-h https://example.com -Tuning 123bde
-h http://xxx.xxx.xxx.xxx -no404
-h http://example.com -useragent "Mozilla/5.0 (Pentest)"
-h http://example.com
```

Illustration 6.3.1 - Tool visibility by user role

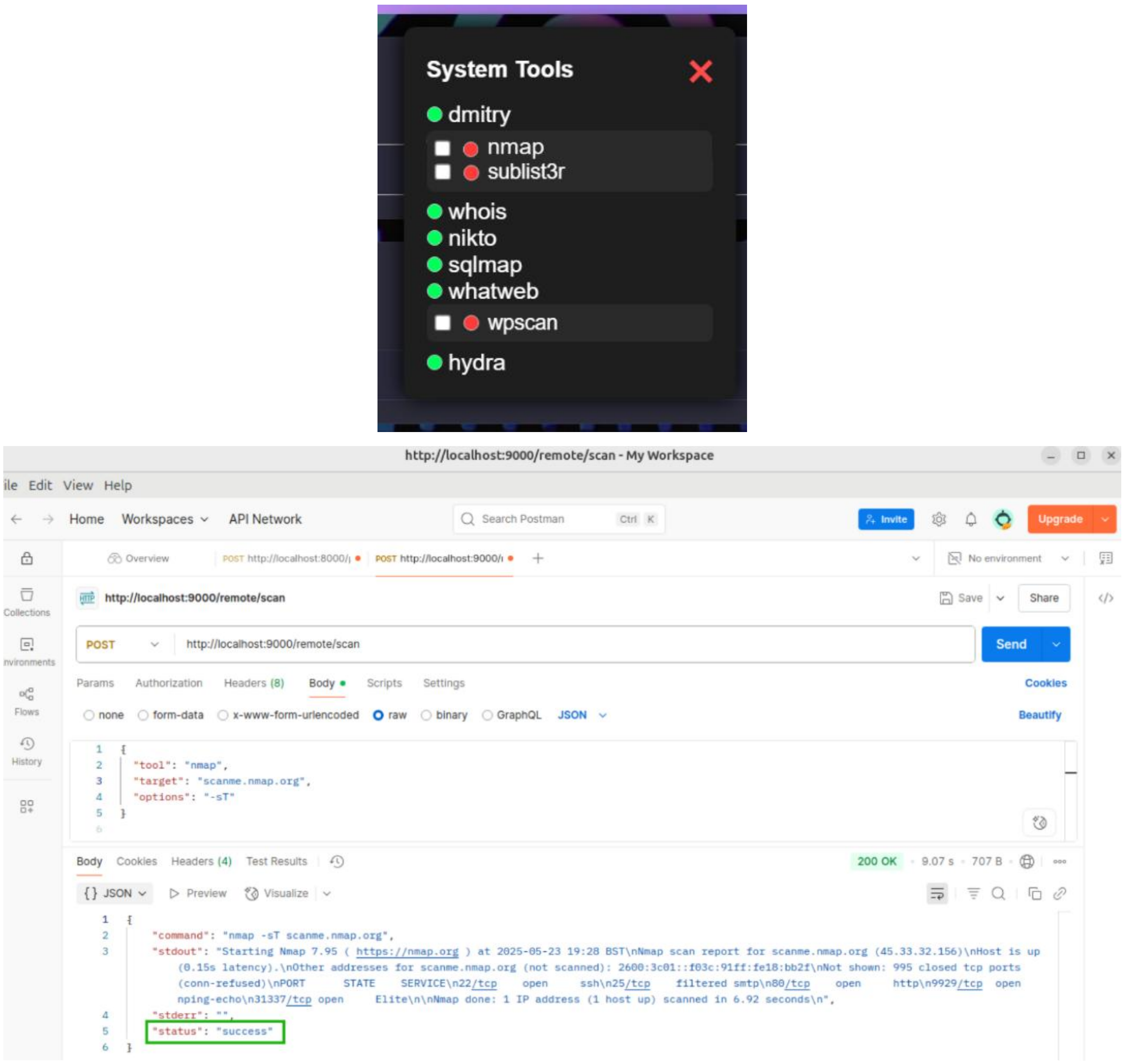
<u>Outcome</u>: **Passed**. The interface correctly adapts tool visibility based on the user's role. In User mode, only a single Nikto command is displayed, while in Superuser mode, multiple advanced command variants become available.

- **Test Case 2: Remote Execution Fallback for Missing Tools**

<u>Objective</u>: Confirm that when a standard user (U) selects a tool that is not installed locally, the backend correctly offloads execution to the remote Virtual Machine.

<u>Method</u>: A standard user (U) launched a scan using *Nmap*, which was disabled locally due to lack of installation or permissions (as shown by the red indicators in the "System Tools" panel). The backend detected the restriction and automatically offloaded execution to the remote server.

<u>Expected Result</u>: The scan should be executed on the remote Virtual Machine, and the results should be returned seamlessly to the user interface.



```
{
  "tool": "nmap",
  "target": "scanme.nmap.org",
  "options": "-sT"
}
```

```
{
  "command": "nmap -sT scanme.nmap.org",
  "stdout": "Starting Nmap 7.95 ( https://nmap.org ) at 2025-05-23 19:28 BST\nNmap scan report for scanme.nmap.org (45.33.32.156)\nHost is up
    (0.15s latency).\nOther addresses for scanme.nmap.org (not scanned): 2600:3c01::f03c:91ff:fe18:bb2f\nNot shown: 995 closed tcp ports
    (conn-refused)\nPORT      STATE    SERVICE\n22/tcp    open     ssh\n25/tcp    filtered smtp\n80/tcp    open     http\n9929/tcp  open
    nping-echo\n31337/tcp open     Elite\n\nNmap done: 1 IP address (1 host up) scanned in 6.92 seconds\n",
  "stderr": "",
  "status": "success"
}
```

Illustration 6.3.2 - Remote execution of Nmap triggered from a local system with restricted access (<u>top</u>: local unavailability; <u>bottom</u>: successful remote scan and output)

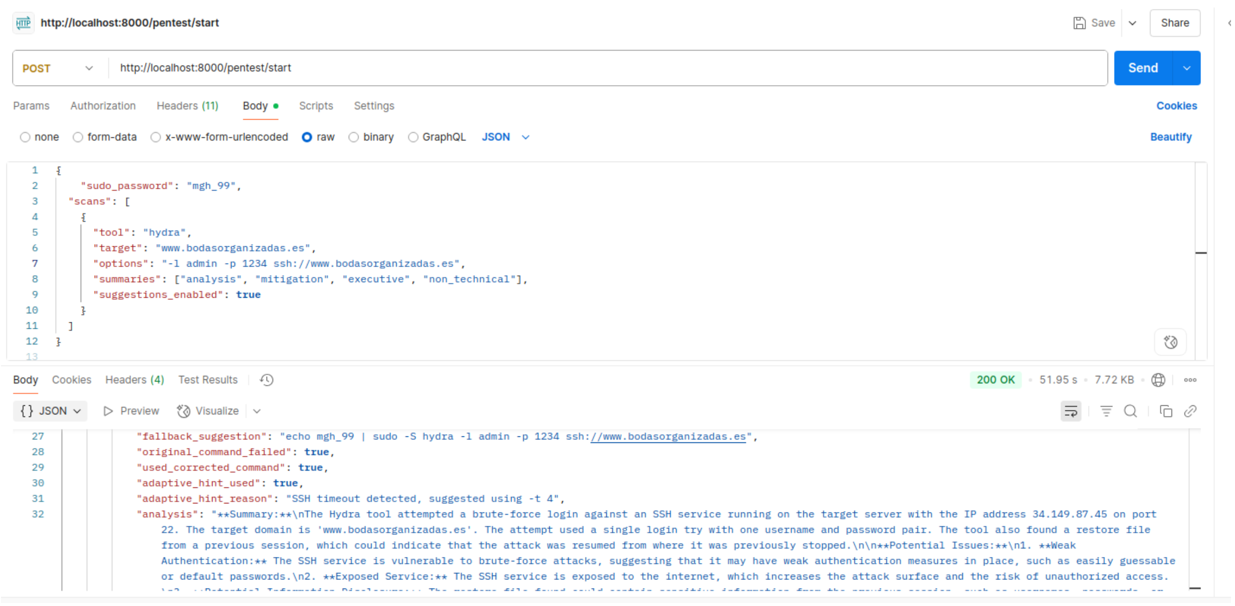
<u>Outcome</u>: **Passed**. The test confirmed that Glitch_ai successfully evades local restrictions by redirecting incompatible tool executions to the remote backend. The Postman interface (second image) shows a successful execution of Nmap with valid results.

- **Test Case 3: LLM-based Retry After Tool Failure**

<u>Objective</u>: Validate that Glitch_ai leverages its integrated LLM (e.g., Mistral or CodeStral) to handle tool execution failures and dynamically generate alternative strategies.

<u>Method</u>: Run a Hydra analysis with incomplete or problematic parameters (e.g., low timeout or inaccessible service) to intentionally trigger a failure. Ensure that the *'suggestions_enabled'* option is set to *'true'*. Observe the adaptive response from the backend.

<u>Expected Result</u>: The LLM identifies the context of the failure (e.g., SSH timeout) and recommends a corrected version of the command (e.g., increase the timeout with '*-t 4*'). The backend captures the alternative resource, applies the corrected command, and continues execution.


http://localhost:8000/pentest/start

POST http://localhost:8000/pentest/start

```
{
  "sudo_password": "mgh_99",
 "scans": [
  {
   "tool": "hydra",
   "target": "www.bodasorganizadas.es",
   "options": "-l admin -p 1234 ssh://www.bodasorganizadas.es",
   "summaries": ["analysis", "mitigation", "executive", "non_technical"],
   "suggestions_enabled": true
  }
 ]
}
```

200 OK · 51.95 s · 7.72 KB

```
"fallback_suggestion": "echo mgh_99 | sudo -S hydra -l admin -p 1234 ssh://www.bodasorganizadas.es",
"original_command_failed": true,
"used_corrected_command": true,
"adaptive_hint_used": true,
"adaptive_hint_reason": "SSH timeout detected, suggested using -t 4",
"analysis": "**Summary:**\nThe Hydra tool attempted a brute-force login against an SSH service running on the target server with the IP address 34.149.87.45 on port
    22. The target domain is 'www.bodasorganizadas.es'. The attempt used a single login try with one username and password pair. The tool also found a restore file
    from a previous session, which could indicate that the attack was resumed from where it was previously stopped.\n\n**Potential Issues:**\n1. **Weak
    Authentication:** The SSH service is vulnerable to brute-force attacks, suggesting that it may have weak authentication measures in place, such as easily guessable
    or default passwords.\n2. **Exposed Service:** The SSH service is exposed to the internet, which increases the attack surface and the risk of unauthorized access.
```

Illustration 6.3.3 - Adaptive retry in Hydra after timeout failure, with fallback suggested by the LLM and incorporated into execution

Outcome: **Passed.** The LLM detected the initial error, issued an alternative suggestion *('-t 4')*, and executed the corrected command. Additionally, the analysis section confirms that the tool completed and reported the relevant findings.

The results of the functional accuracy tests demonstrate that Glitch_ai fulfills the defined performance specifications when operating in restricted environments.

Execution time

Execution times changed depending on whether tools operated in local or remote environments. On average:

Tool	Target	Options	Execution Time (Local)	Execution Time (Remote)
Dmitry	www.bodasorganizadas.com	-w	0.126688 sec	0.06022 sec
Nmap	www.bodasorganizadas.com	-p 80,443	0.548404 sec	0.68943 sec
Whois	www.bodasorganizadas.com	| grep \"Registrar\	0.220297 sec	0.24005 sec
SQLMap	www.bodasorganizadas.com	--banner --level=1 --risk=1 --batch --timeout=20	27.289663 sec	40.3468 sec

Table 6.3.1 - Execution times local vs. remote environment

These results confirm that:

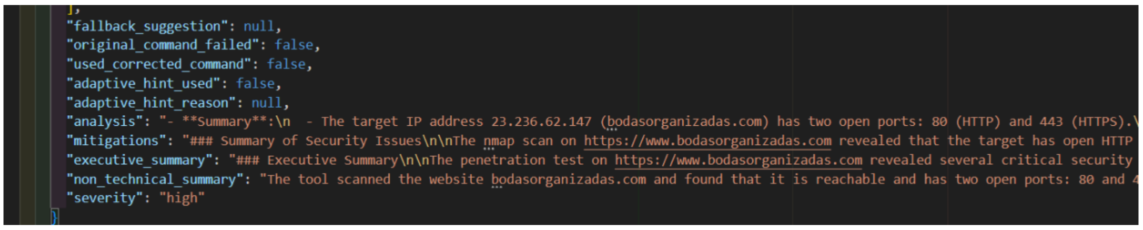
- Execution times remained **similar** in both local and remote environments for lightweight tools such as Dmitry, Whois, and Nmap.
- More complex tools, such as SQLMap, exhibited increased execution times in remote environments, likely due to greater computational demands and network latency.
- In all cases, the remote backup feature worked as expected, maintaining consistent output formats and reliable results.

This comparison indicates that remote execution introduces moderate delays, particularly for tools with extended execution times. However, it maintains **full functionality** in restricted environments, which is a primary objective of Glitch_ai's hybrid architecture.

Report Quality and Interpretability

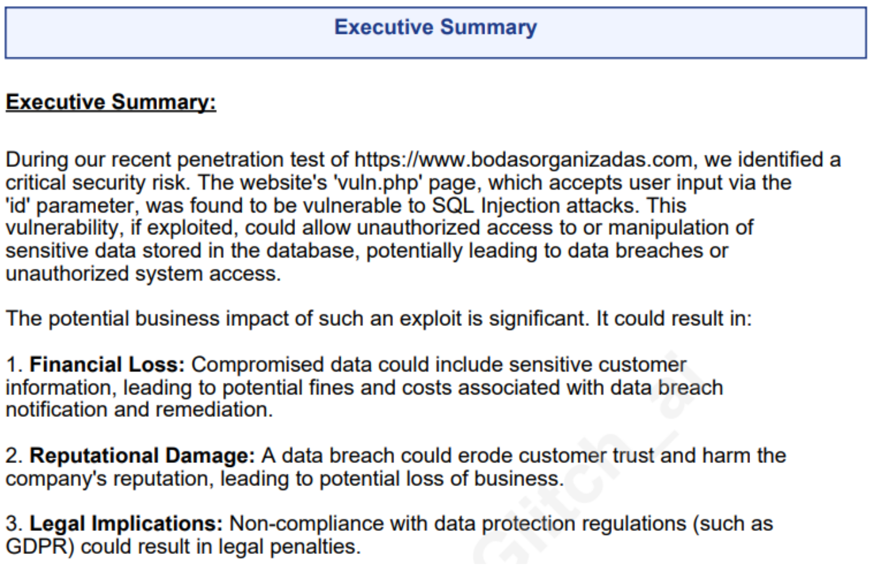
A distinctive capability of Glitch_ai is its structured reporting system, which adjusts **technical detail to match the intended audience**. Reports are generated in JSON and PDF formats and are divided into clearly defined sections for quick interpretation and decision-making. During testing, the reporting module was evaluated using four main criteria:

- **Analysis**: Detailed technical interpretation of the scan results, including IP, port status, protocol types, and vulnerability context. As shown in *Illustration 6.3.4*, the tool correctly identified and described exposed ports on a given target, such as ports 80 (HTTP) and 443 (HTTPS), using structured markdown formatting.
- **Mitigation**: Actionable remediation strategies tailored to the detected vulnerabilities. These were generated through LLM-based reasoning and adapted to each specific case. *Illustration 6.3.6* illustrates a scenario in which SQL injection was detected, with the corresponding mitigation section outlining specific steps, such as using parameterized queries and implementing input validation.
- **Executive Summary**: Designed for management or decision-makers, this section communicates the impact of findings without requiring technical knowledge. *Illustration 6.3.5* shows a clear example of a professional-grade executive summary, listing business risks such as financial loss, reputational damage, and legal implications in response to a critical web vulnerability.
- **Non-technical Summary**: A plain-language explanation designed to inform users without a cybersecurity background. For example, the JSON output (*Illustration 6.3.4*) describes the website's accessibility and clarifies the significance of open ports.



```
    ],
    "fallback_suggestion": null,
    "original_command_failed": false,
    "used_corrected_command": false,
    "adaptive_hint_used": false,
    "adaptive_hint_reason": null,
    "analysis": "- **Summary**:\n  - The target IP address 23.236.62.147 (bodasorganizadas.com) has two open ports: 80 (HTTP) and 443 (HTTPS).\
    "mitigations": "### Summary of Security Issues\n\nThe nmap scan on https://www.bodasorganizadas.com revealed that the target has open HTTP
    "executive_summary": "### Executive Summary\n\nThe penetration test on https://www.bodasorganizadas.com revealed several critical security
    "non_technical_summary": "The tool scanned the website bodasorganizadas.com and found that it is reachable and has two open ports: 80 and 4
    "severity": "high"
}
```

Illustration 6.3.4 - An extract from the JSON results generated by Glitch_ai



Executive Summary

Executive Summary:

During our recent penetration test of https://www.bodasorganizadas.com, we identified a critical security risk. The website's 'vuln.php' page, which accepts user input via the 'id' parameter, was found to be vulnerable to SQL Injection attacks. This vulnerability, if exploited, could allow unauthorized access to or manipulation of sensitive data stored in the database, potentially leading to data breaches or unauthorized system access.

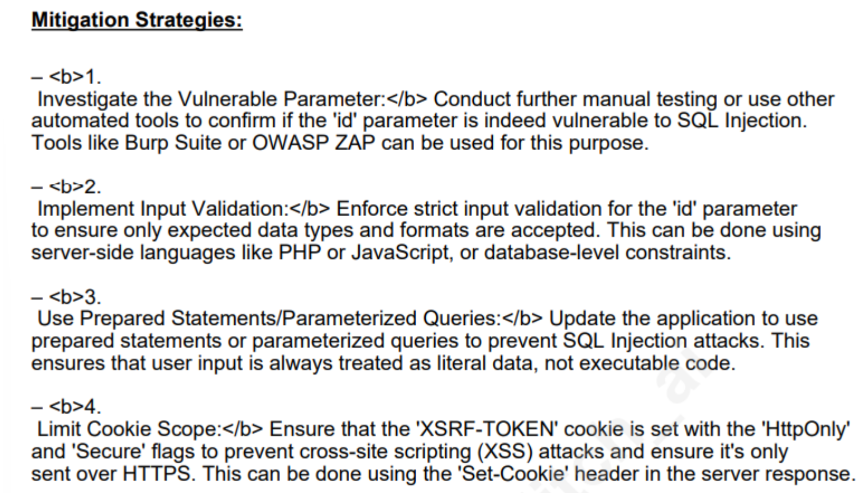
The potential business impact of such an exploit is significant. It could result in:

1. **Financial Loss:** Compromised data could include sensitive customer information, leading to potential fines and costs associated with data breach notification and remediation.

2. **Reputational Damage:** A data breach could erode customer trust and harm the company's reputation, leading to potential loss of business.

3. **Legal Implications:** Non-compliance with data protection regulations (such as GDPR) could result in legal penalties.

Illustration 6.3.6 - An extract from the PDF report: executive summary generated for a SQL injection scenario.



Mitigation Strategies:

– <b>1.
 Investigate the Vulnerable Parameter:</b> Conduct further manual testing or use other automated tools to confirm if the 'id' parameter is indeed vulnerable to SQL Injection. Tools like Burp Suite or OWASP ZAP can be used for this purpose.

– <b>2.
 Implement Input Validation:</b> Enforce strict input validation for the 'id' parameter to ensure only expected data types and formats are accepted. This can be done using server-side languages like PHP or JavaScript, or database-level constraints.

– <b>3.
 Use Prepared Statements/Parameterized Queries:</b> Update the application to use prepared statements or parameterized queries to prevent SQL Injection attacks. This ensures that user input is always treated as literal data, not executable code.

– <b>4.
 Limit Cookie Scope:</b> Ensure that the 'XSRF-TOKEN' cookie is set with the 'HttpOnly' and 'Secure' flags to prevent cross-site scripting (XSS) attacks and ensure it's only sent over HTTPS. This can be done using the 'Set-Cookie' header in the server response.

Illustration 6.3.5 - An extract from the PDF report: mitigation strategies generated by Glitch_ai

6.4. Comparative Observations

During testing, Glitch_ai showed several features that make it stand out from other automated or AI-based penetration testing tools. Previous solutions typically provide limited automation or machine learning, but their workflows are often fixed, cloud-only, or restricted to specific tasks.

In contrast, Glitch_ai's hybrid architecture enables both **local and remote execution**, dynamically adapting to user permissions and system capabilities. This flexibility addresses a limitation observed in tools such as ImmuniWeb, Bright Security, and Detectify, which operate almost exclusively in the cloud and do not support custom execution environments [8] [7] [6].

Another important difference is **how AI is used**. Most platforms employ machine learning for tasks such as sorting, prioritization, or reducing false positives, but they rarely use it for reasoning. Glitch_ai uses LLMs not only for language generation but also for decision-making, backup management, and adaptation to diverse scenarios. This approach extends beyond conventional uses of AI in security tools.

In addition, the inclusion of **role-based execution constraints** and **natural language reports** tailored to both technical and non-technical audiences improve usability and transparency, aspects that are often lacking in traditional tools or even AI-enhanced ones.

Although this comparison is not based on empirical testing of third-party tools, the conclusions have been drawn from public documentation, technical reviews, and direct observation of Glitch_ai's capabilities in scenarios designed to reflect real-world limitations.

7. Project Management

This section describes the tools, methodologies, and strategies used to manage the development of Glitch_ai. Task planning and time management were achieved through the integration of GitLab and the maintenance of a structured Gantt chart.

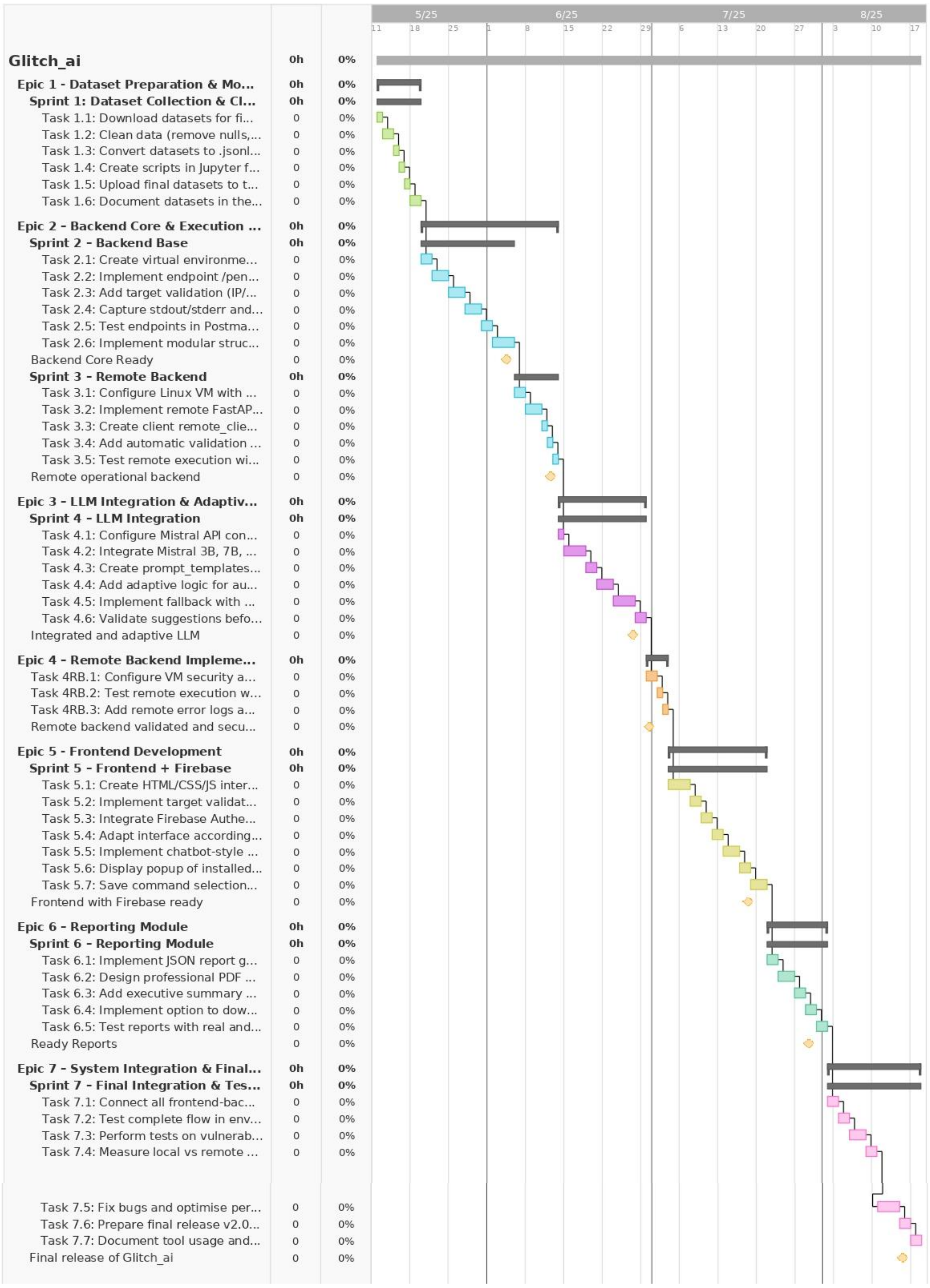
7.1. GitLab

Although GitHub was initially used for local version control during the development of Glitch_ai, the final code base was uploaded to GitLab to meet academic submission requirements. GitLab served as the official evaluation platform, hosting the complete frontend and backend source code, deployment scripts, test data, and documentation. All files include comments and a well-structured commit history, ensuring transparency and reproducibility.

The repository can be accessed at: https://cseegit.essex.ac.uk/24-25-ce901-su-ce902-sp/24-25_CE901-SU_CE902-SP_gonzalez_herrero

7.2. Gantt chart

A detailed Gantt chart was created using TeamGantt, an online project scheduling and collaboration tool [50], to plan and coordinate the development of Glitch_ai. The chart segmented the project into seven epics, each representing a core functional area, and distributed them over a four-month period from mid-May to mid-August 2025. Each epic was further divided into sprints and specific tasks, ensuring clear structure and traceability from data preparation to system launch.



Glitch_ai	**0h**	**0%**
Epic 1 - Dataset Preparation & Mo...	**0h**	**0%**
Sprint 1: Dataset Collection & Cl...	**0h**	**0%**
Task 1.1: Download datasets for fi...	0	0%
Task 1.2: Clean data (remove nulls,...	0	0%
Task 1.3: Convert datasets to .jsonl...	0	0%
Task 1.4: Create scripts in Jupyter f...	0	0%
Task 1.5: Upload final datasets to t...	0	0%
Task 1.6: Document datasets in the...	0	0%
Epic 2 - Backend Core & Execution ...	**0h**	**0%**
Sprint 2 - Backend Base	**0h**	**0%**
Task 2.1: Create virtual environme...	0	0%
Task 2.2: Implement endpoint /pen...	0	0%
Task 2.3: Add target validation (IP/...	0	0%
Task 2.4: Capture stdout/stderr and...	0	0%
Task 2.5: Test endpoints in Postma...	0	0%
Task 2.6: Implement modular struc...	0	0%
Backend Core Ready	0	0%
Sprint 3 - Remote Backend	**0h**	**0%**
Task 3.1: Configure Linux VM with ...	0	0%
Task 3.2: Implement remote FastAP...	0	0%
Task 3.3: Create client remote_clie...	0	0%
Task 3.4: Add automatic validation ...	0	0%
Task 3.5: Test remote execution wi...	0	0%
Remote operational backend	0	0%
Epic 3 - LLM Integration & Adaptiv...	**0h**	**0%**
Sprint 4 - LLM Integration	**0h**	**0%**
Task 4.1: Configure Mistral API con...	0	0%
Task 4.2: Integrate Mistral 3B, 7B, ...	0	0%
Task 4.3: Create prompt_templates...	0	0%
Task 4.4: Add adaptive logic for au...	0	0%
Task 4.5: Implement fallback with ...	0	0%
Task 4.6: Validate suggestions befo...	0	0%
Integrated and adaptive LLM	0	0%
Epic 4 - Remote Backend Impleme...	**0h**	**0%**
Task 4RB.1: Configure VM security a...	0	0%
Task 4RB.2: Test remote execution w...	0	0%
Task 4RB.3: Add remote error logs a...	0	0%
Remote backend validated and secu...	0	0%
Epic 5 - Frontend Development	**0h**	**0%**
Sprint 5 - Frontend + Firebase	**0h**	**0%**
Task 5.1: Create HTML/CSS/JS inter...	0	0%
Task 5.2: Implement target validat...	0	0%
Task 5.3: Integrate Firebase Authe...	0	0%
Task 5.4: Adapt interface according...	0	0%
Task 5.5: Implement chatbot-style ...	0	0%
Task 5.6: Display popup of installed...	0	0%
Task 5.7: Save command selection...	0	0%
Frontend with Firebase ready	0	0%
Epic 6 - Reporting Module	**0h**	**0%**
Sprint 6 - Reporting Module	**0h**	**0%**
Task 6.1: Implement JSON report g...	0	0%
Task 6.2: Design professional PDF ...	0	0%
Task 6.3: Add executive summary ...	0	0%
Task 6.4: Implement option to dow...	0	0%
Task 6.5: Test reports with real and...	0	0%
Ready Reports	0	0%
Epic 7 - System Integration & Final...	**0h**	**0%**
Sprint 7 - Final Integration & Tes...	**0h**	**0%**
Task 7.1: Connect all frontend-bac...	0	0%
Task 7.2: Test complete flow in env...	0	0%
Task 7.3: Perform tests on vulnerab...	0	0%
Task 7.4: Measure local vs remote ...	0	0%
Task 7.5: Fix bugs and optimise per...	0	0%
Task 7.6: Prepare final release v2.0...	0	0%
Task 7.7: Document tool usage and...	0	0%
Final release of Glitch_ai	0	0%

Illustration 7.2.1 - Gantt chart of the project

8. Future Work and Conclusions

This final chapter describes possible avenues for future development and concludes with a reflection on the overall **contributions and results** of the project. The first section presents proposed improvements to increase the flexibility, intelligence, and scalability of the tool. The second section summarises the main achievements, analyses their impact on the field of cybersecurity, and highlights how Glitch_ai bridges the gap between traditional penetration testing and modern AI-assisted workflows.

8.1. Future Work

Several areas for further improvement of Glitch_ai have been identified during its development. The following proposals specify future work to expand functionality, optimize deployment across environments, and enhance automation and adaptive reasoning.

1. **Dynamic dictionary selection for Hydra:** Currently, Glitch_ai uses a standard password list by default when launching brute force attacks through Hydra. Future iterations will support contextual selection between three dictionary profiles: fast, standard and deep. This will allow the system to adapt to different attack scenarios, opting for speed or thoroughness depending on user preferences, tool configuration, or target criticality.
2. **Tool expansion based on rigorous evaluation:** Rather than arbitrarily expanding the toolset, Glitch_ai will integrate new penetration testing utilities after a structured evaluation. Tools will be evaluated based on their compatibility with the CLI, their added value, their redundancy, and their feasibility for integration with the existing backend logic and user interface. This ensures that future additions remain useful and maintain the consistency of the system.
3. **Real-time execution flow and feedback loop:** To enhance transparency and user engagement, the chatbot interface should incorporate real-time status updates for each tool execution, including load status, error detection, and retry logic managed by LLMs. Additionally, users could rate the usefulness of AI-generated suggestions, enabling the backend to collect feedback and adjust system performance accordingly.
4. **Docker-based implementation:** A containerised version of Glitch_ai would be developed using Docker, allowing the platform to be easily deployed in different environments. This would simplify installation, eliminate dependency issues, and improve reproducibility, especially for remote teams or academic environments.
5. **Parallel execution engine:** Glitch_ai will implement concurrency for supported tools, allowing simultaneous execution when commands do not depend on shared resources. This improvement will significantly reduce the total analysis time, especially in multi-target workflows, while maintaining consistency of results and system stability.
6. **Improved report visualisation:** Future reports will include new visual elements, such as bar charts and heat maps, to represent detected vulnerabilities and tool results.
7. **Advanced threat intelligence summary**: Using LLM, Glitch_ai could summarise external threat intelligence sources (e.g. CVE feeds, blogs, security reports) into short, actionable reports. These summaries would help contextualise vulnerabilities identified during scans, providing users with up-to-date prior knowledge without the need for manual research.
8. **Exploit simulation (controlled):** A strictly regulated module will allow Glitch_ai to simulate potential exploits for detected vulnerabilities, offering ethical red teaming capabilities. Exploit generation will be limited to sandbox environments and will only be accessible to authorised users. This feature aims to bridge the gap between detection and real-world impact assessment in a safe and responsible way.

These future directions encompass both user-focused improvements and system-level enhancements that will establish Glitch_ai as a more autonomous and intelligent platform.

8.2. Conclusions

This project introduced **Glitch_ai**, a modular and hybrid web-based platform designed to optimise and improve penetration testing workflows by integrating traditional security tools with LLMs. The platform aims to bridge the gap between automation and contextual reasoning, enabling a more intelligent and adaptable approach to vulnerability analysis.

Throughout the development process, **several key challenges were addressed**. A notable achievement was the implementation of a remote backend system that allows users without elevated permissions to perform scans through a virtual environment. This design choice significantly expanded the platform's accessibility, ensuring that effective testing could be performed even in restricted environments.

In addition, the system incorporated a coordination engine capable of responding to tool failures with LLM-powered fallback logic, as well as a role-based interface that dynamically adjusts tool visibility. The reporting module was also enhanced with customised results for both technical analysts and non-technical stakeholders, improving the interpretability and communication of results.

The testing phase validated Glitch_ai's core functionalities in unit, integration, and real-world validation scenarios. The results demonstrated reliable behaviour in complex workflows, effective handling of execution errors, and overall improvement in the clarity and usability of reports. Compared to traditional and AI-assisted alternatives, the platform offers a unique combination of adaptability, modularity, and explainability.

Although the system met its main objectives, several areas for improvement have been identified. These include optimising execution performance, expanding tool support based on rigorous evaluation, and incorporating real-time user feedback to improve the accuracy and usefulness of LLM responses.

In conclusion, Glitch_ai is a practical and forward-looking contribution to the field of automated penetration testing. Its architecture lays a solid foundation for future developments in AI-driven cybersecurity, where flexibility, intelligence, and collaboration between humans and AI will be critical to defending against increasingly complex threats.

9. References

[1] Al, Hessa Mohammed Zaher Shebli and Babak D. Beheshti, "A study on penetration testing process and tools," Long Island Systems, Applications and Technology Conference (LISAT), Farmingdale, NY, USA, 2018. [Online]. https://ieeexplore.ieee.org/abstract/document/8378035

[2] Deepanshu Garg and Nishu Bansal, "A Systematic Review on Penetration Testing," Global Conference for Advancement in Technology (GCAT), Bangalore, India, 2021. [Online]. https://ieeexplore.ieee.org/abstract/document/9587771

[3] Prashant Vats, Manju Mandot, and Anjana Gosain, "A Comprehensive Literature Review of Penetration Testing & Its Applications," (Trends and Future Directions) (ICRITO), Noida, India, 2020. [Online]. https://ieeexplore.ieee.org/abstract/document/9197961

[4] Chiem Trieu Phong, "A Study of Penetration Testing Tools," School of Computing and Mathematical Sciences, Auckland, New Zeland, 2014. [Online]. https://openrepository.aut.ac.nz/server/api/core/bitstreams/a86ef9ec-82aa-40f1-bf64-a48a3f8b86b0/content

[5] Daniel Dalalana Bertoglio and Avelino Francisco Zorzo, *Overview and open issues on penetration test*. Brazil: Journal of the Brazilian Computer Society, 2017. [Online]. https://link.springer.com/article/10.1186/s13173-017-0051-1

[6] Detectify. detectify. [Online]. https://detectify.com/

[7] Brightsec. bright. [Online]. https://brightsec.com/

[8] Immuniweb. immuniweb. [Online]. https://www.immuniweb.com/

[9] Acunetix. acunetix. [Online]. https://www.acunetix.com/

[10] Invicti. invicti. [Online]. https://www.invicti.com/

[11] ProjectDiscovery. projectdiscovery.io. [Online]. https://projectdiscovery.io/

[12] Checkmarx. zaproxy.org. [Online]. https://www.zaproxy.org/

[13] Muhammad Ismaeel Khan, Aftab Arif, and Ali Raza A Khan, "The Most Recent Advances and Uses of AI in Cybersecurity," MIST at Washington University of Sciencie and technology; Virginia University of Sciencie & Technology, Washington; Virginia, 2024. [Online]. https://www.researchgate.net/profile/Muhammad-Ismaeel-Khan/publication/390740851_The_Most_Recent_Advances_and_Uses_of_AI_in_Cybersecurity/links/67fb8256bd3f1930dd5d5d8f/The-Most-Recent-Advances-and-Uses-of-AI-in-Cybersecurity.pdf

[14] Naveed Naeem Abbas, Tanveer Ahmed, Syed Habib Ullah Shah, and Muhammad Omar, *Investigating the applications of artificial intelligence in cyber security*., 2019. [Online]. https://link.springer.com/article/10.1007/s11192-019-03222-9

[15] Sunriz Islam, Md. Abul Hayat, and Md. Fokhray Hossain, "Artificial Intelligence for Cibersecurity: Impact, Limitations and Future Research Directions," Hajee Mohammad Danesh Science and Technology University; Daffodil International University, 2023. [Online]. https://www.researchgate.net/publication/377019141_ARTIFICIAL_INTELLIGENCE_FOR_CYBERSECURITY_IMPACT_LIMITATIONS_AND_FUTURE_RESEARCH_DIRECTIONS

[16] Dinesh Kalla, Sivaraju Kuraku, and Fnu Samaah, "Advantages, Disadvantages and Risks Associated with ChatGPT and AI on Cybersecurity," Colorado Technical University; University of the Cumberlands; Northeastern Illinois University, 2023. [Online]. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4619204

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, and Jakob Uszkoreit, "Attention Is All You Need," Google Research; University of Toronto, 2023. [Online]. https://arxiv.org/pdf/1706.03762

[18] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, and Meysam Chenaghlu, "Large Language Models: A Survey," 2024. [Online]. https://arxiv.org/html/2402.06196v2

[19] Jie Zhang, Haoyu Bu, Hui Wen, and Yongji Liu, "When LLMs meet cybersecurity: a systematic literature review," 2025. [Online]. https://cybersecurity.springeropen.com/articles/10.1186/s42400-025-00361-w

[20] Lei Huang, Weijiang Yu, Weitao Ma, and Weihong Zhong, "A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions," 2024. [Online]. https://arxiv.org/abs/2311.05232

[21] GenAI Security Project. (2025) LLM01:2025 Prompt Injection. [Online]. https://genai.owasp.org/llmrisk/llm01-prompt-injection/

[22] Kevin E. Awoufack, "Adversarial Prompt Transformation for Systematic Jailbreaks of LLMs," Massachusetts Institute of Technology, 2024. [Online]. http://dspace.mit.edu/handle/1721.1/157167

[23] Jiaxin Fan, Qi Yan, Mohan Li, Guanqun Qu, and Yang Xiao, "A Survey on Data Poisoning Attacks and Defenses," IEEE International Conference on Data Science in Cyberspace (DSC), Guilin, China, 2022. [Online]. http://dspace.mit.edu/handle/1721.1/157167

[24] Alexander Wan, Eric Wallace, Sheng Shen, and Dan Klein, "Poisoning Language Models During Instruction Tuning," 2023. [Online]. https://proceedings.mlr.press/v202/wan23b

[25] Zichuan Fu, Wentao Song, Yejing Wang, Xian Wu, and et al, "Sliding Window Attention Training for Efficient Large Language Models," 2025. [Online]. https://arxiv.org/abs/2502.18845

[26] Jie Zhang, Haoyu Bu, Hui Wen, Yongji Liu, and et al., "When LLMs meet cybersecurity: a systematic literature review," 2025. [Online]. https://link.springer.com/article/10.1186/s42400-025-00361-w

[27] Hanxiang Xu, Shenao Wang, Ningke Li, Kailong Wang, and et al., "Large Language Models for Cyber Security: A Systematic Literature Review," 2025. [Online]. https://arxiv.org/abs/2405.04760

[28] Kali. Dmitry. [Online]. https://www.kali.org/tools/dmitry/

[29] Hydra. (2025) hydra. [Online]. https://hydra.cc/docs/intro/

[30] cisa.gov. Nikto. [Online]. https://www.cisa.gov/resources-tools/services/nikto

[31] Nmap. nmap. [Online]. https://nmap.org/

[32] Kali. Sublist3r. [Online]. https://www.kali.org/tools/sublist3r/

[33] Bernardo Damele A. G. and Miroslav Stampar. sqlmap. [Online]. https://sqlmap.org/

[34] WhatWeb. whatweb. [Online]. https://whatweb.net/

[35] who.is. Domain Name Information Lookup. [Online]. https://who.is/

[36] WPScan. WPScan. [Online]. https://wpscan.com/

[37] FastAPI. FastAPI. [Online]. https://fastapi.tiangolo.com/

[38] Mistral AI. Mistral AI Documentation. [Online]. https://docs.mistral.ai/

[39] Mistral AI. Models Overview. [Online]. https://docs.mistral.ai/getting-started/models/models_overview/

[40] Scrapfly. (2024) JSONL vs JSON. [Online]. https://dev.to/scrapfly_dev/jsonl-vs-json-hb0

[41] Dave Bergmann. (2024) What is fine-tuning? [Online]. https://www.ibm.com/think/topics/fine-tuning

[42] IBM. (2025) Parameters for tuning foundation models. [Online]. https://www.ibm.com/docs/en/watsonx/w-and-w/2.1.0?topic=models-tuning-experiment-parameters

[43] Hugging Face. Fine-tuning. [Online]. https://huggingface.co/docs/transformers/training

[44] Milvus. How does the number of training epochs during fine-tuning affect the quality of a Sentence Transformer model versus the risk of overfitting? [Online]. https://milvus.io/ai-quick-reference/how-does-the-number-of-training-epochs-during-finetuning-affect-the-quality-of-a-sentence-transformer-model-versus-the-risk-of-overfitting

[45] Miha Cacic. (2023) What are Fine-tuning Hyperparameters and How to Set Them Just Right. [Online]. https://www.entrypointai.com/blog/fine-tuning-hyperparameters/

[46] Pydantic. Pydantic. [Online]. https://docs.pydantic.dev/latest/

[47] Firebase. Firebase. [Online]. https://firebase.google.com/?gad_source=1&gad_campaignid=12300214754&gbraid=0AAAAADpUDOjLkUDuDHH74RzQTYi1SsbBX&gclid=CjwKCAjwyb3DBhBlEiwAqZLe5Bmj-AsXqVA6MrkGJSpLY0U2yJbPDfLNBX0kitQybq8bnWeu0CAeZxoCW7IQAvD_BwE&gclsrc=aw.ds

[48] Pytest. pytest documentation. [Online]. https://docs.pytest.org/en/stable/

[49] Microsoft GitHub. Integration Testing. [Online]. https://microsoft.github.io/code-with-engineering-playbook/automated-testing/integration-testing/

[50] Indeed Editorial Team. (2025) What Is Validation Testing In Software Development? [Online]. https://www.indeed.com/career-advice/starting-new-job/what-is-validation-testing

[51] OWASP. OWASP Damn Vulnerable Web Sockets. [Online]. https://owasp.org/www-project-damn-vulnerable-web-sockets/

[52] Team Gantt. team gantt. [Online]. https://www.teamgantt.com/