POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Tesi di Laurea Magistrale in
Computer Science and Engineering
Ingegneria Informatica

An AI-Driven Penetration Testing Framework for Autonomous Offensive Security

Author:
Lorenzo Ladina

Student ID:
251841

Advisor:
Prof. Stefano Zanero

Academic Year:
2025-2026

"All we have to decide is what to do with the time that is given us."

— J.R.R. Tolkien

Abstract

The increasing complexity of modern IT infrastructures renders traditional penetration testing approaches, which rely heavily on manual enumeration and static scanners, difficult to scale. While Large Language Models (LLMs) have demonstrated significant reasoning capabilities in offensive security, their deployment as autonomous agents is hindered by severe operational limitations: context window saturation (Vertical Bloat) from ingesting raw logs, the resulting Cognitive Drag, and a critical vulnerability to Indirect Prompt Injection (IPI) attacks originating from the target environment.

This thesis proposes *An AI-Driven Penetration Testing Framework for Autonomous Offensive Security*, a middleware architecture designed to address these systemic bottlenecks. The research introduces a Semantic Compression layer capable of distilling complex security telemetry (such as Active Directory [5] relational graphs and web DOMs [37]) and a strategic memory management system. Furthermore, the framework implements a Defense-in-Depth architecture that decouples the logical reasoning engine from execution via deterministic command validation.

Ablation studies demonstrate that semantic distillation reduces the token footprint by over 85% and significantly decreases the "Thinking Tokens" expended on redundant processing. From a security perspective, empirical tests confirm that the proposed architecture effectively mitigates environmental vulnerabilities, reducing the IPI success rate from 100% to a residual 10%. Ultimately, this research establishes that the resilience and economic scalability of autonomous offensive agents depend not solely on the underlying model's capacity, but require rigorous engineering of the orchestration architecture.

Keywords: Large Language Models, Offensive Security, Penetration Testing, Indirect Prompt Injection, Semantic Compression.

Abstract in lingua italiana

La crescente complessità delle moderne infrastrutture IT rende i tradizionali approcci di penetration testing, basati sull'analisi manuale e su scanner statici, difficilmente scalabili. Sebbene i Large Language Models (LLM) abbiano dimostrato notevoli capacità di ragionamento nel campo della sicurezza offensiva, il loro impiego come agenti autonomi è ostacolato da severi limiti operativi: la saturazione della finestra di contesto (Vertical Bloat) dovuta all'ingestione di log grezzi, il conseguente attrito cognitivo (Cognitive Drag) e una marcata vulnerabilità agli attacchi di Indirect Prompt Injection (IPI) provenienti dall'ambiente bersaglio.

Questa tesi propone *An AI-Driven Penetration Testing Framework for Autonomous Offensive Security*, un'architettura middleware progettata per risolvere questi limiti sistemici. Il lavoro introduce un layer di Compressione Semantica in grado di distillare l'output di tool di sicurezza complessi (come i grafi relazionali in ambienti Active Directory [5] o i DOM web [37]) e un sistema di gestione strategica della memoria. Inoltre, il framework implementa un'architettura Defense-in-Depth che separa il motore di ragionamento logico dall'esecuzione, applicando una validazione deterministica dei comandi.

Gli studi di ablazione condotti dimostrano che la distillazione semantica riduce il consumo di token di oltre l'85% e abbatte significativamente i "Thinking Tokens" spesi in processi ridondanti. Dal punto di vista della sicurezza, i test empirici confermano che l'architettura proposta mitiga efficacemente le vulnerabilità ambientali, riducendo il tasso di successo degli attacchi IPI dal 100% a un rischio residuo del 10%. In conclusione, questa ricerca dimostra che la resilienza e la scalabilità economica degli agenti offensivi autonomi richiedono una rigorosa ingegnerizzazione dell'architettura di orchestrazione.

Parole chiave: Large Language Models, Offensive Security, Penetration Testing, Indirect Prompt Injection, Semantic Compression.

Contents

1 | Introduction

The landscape of offensive security is undergoing a rapid transformation. As enterprise IT infrastructures become increasingly complex, dynamic, and distributed, the traditional approach to penetration testing—relying heavily on manual enumeration and static automated scanners—is struggling to scale. Human security professionals face significant cognitive loads when parsing massive amounts of network telemetry, while conventional automated tools lack the strategic reasoning required to chain complex, multi-step vulnerabilities [31].

Recently, the advent of Large Language Models (LLMs) has introduced a paradigm shift in cybersecurity. These models possess unprecedented capabilities in natural language understanding, code analysis, and contextual reasoning, making them ideal candidates for autonomous Red Teaming operations [9, 15, 39]. However, deploying an LLM as a fully autonomous penetration testing agent introduces severe operational and architectural challenges that remain unresolved by current general-purpose solutions.

While LLMs demonstrate strong theoretical knowledge of offensive security, their practical application as autonomous agents is hindered by three fundamental technical limitations:

1. **Context Window Saturation (Vertical Bloat):** Security tools such as network scanners and web fuzzers generate massive, unstructured outputs. Feeding raw logs, JSON data, or full HTML Document Object Models (DOMs) directly into an LLM quickly exhausts its context window, degrading its reasoning capabilities and causing it to lose track of the mission's primary objectives.
2. **Cognitive Drag (Horizontal Bloat):** Autonomous agents operate in continuous loops. Without a mechanism to summarize and distill past actions, the linear accumulation of command history forces the model to expend significant computational resources merely to maintain state awareness, drastically increasing latency and any API costs.
3. **Instruction-Data Confusion and Security Risks:** Current generative models

lack a deterministic boundary between trusted system instructions and untrusted environmental data. This architectural flaw leaves autonomous agents highly vulnerable to Indirect Prompt Injection (IPI), where malicious payloads embedded within the target environment can hijack the agent's cognitive loop and force unauthorized actions.

The primary objective of this thesis is to bridge the gap between high-level AI reasoning and low-level security execution by designing, implementing, and evaluating a specialized middleware architecture. To address the limitations of deploying LLMs as standalone tools, this research introduces an orchestration framework designed to improve overall system resilience.

The main contributions of this work are:

- The development of a **Semantic Compression Layer** that programmatically distills raw tool outputs into highly dense, actionable intelligence. As evaluated, this approach reduces the context footprint by over 85%, effectively mitigating context saturation.
- The implementation of **Strategic Memory Management** (via a dynamic Recon Summary) to eliminate cognitive drag. This architecture allows the agent to maintain long-term goal persistence, drastically reducing the "Thinking Tokens" expended on historical parsing.
- An **Evaluation of the System's Robustness**, detailing a layered defense strategy aimed at mitigating Indirect Prompt Injection (IPI) attacks. Through syntactic escaping and deterministic command validation, the framework reduces the attack success rate from a baseline of 100% to a residual 10%.

The remainder of this thesis is organized as follows: Chapter 2 explores the State of the Art, analyzing current applications of LLMs in cybersecurity and their inherent limitations. Chapter 3 details the Proposed System Architecture, introducing the orchestrator's design and the theory behind Semantic Compression. Chapter 4 covers the Implementation and System Engineering, providing insights into the asynchronous framework and the dashboard. Chapter 5 presents the Experimental Evaluation, quantifying efficiency gains through ablation studies. Chapter 6 focuses on the Robustness and Security Analysis regarding IPI attacks. Finally, Chapter 7 outlines the Conclusion and Future Work.

2 | State of the Art

The integration of Artificial Intelligence in offensive security has historically been dominated by statistical machine learning models tailored for specific, narrow tasks, such as malware classification or anomaly detection. The release of advanced Large Language Models (LLMs), such as the GPT-4, Claude Code, and Gemini 3 series [12, 19, 25], has expanded these boundaries. Because these models are trained on vast corpora of code, vulnerability reports, and technical documentation, they possess an inherent capability to analyze source code, identify security misconfigurations, and suggest exploitation vectors [1, 2, 38]. However, transitioning these models from passive assistants to active, autonomous agents represents a significant engineering hurdle.

2.1. Autonomous Agents and Reasoning Architectures

To achieve autonomy, researchers have adapted cognitive architectures to guide LLM execution. The most prominent is the *ReAct* (Reason + Act) paradigm, which forces the model to interleave strategic reasoning traces [36] with actionable commands. This framework is closely aligned with the military *OODA loop* (Observe, Orient, Decide, Act), forming the basis for modern autonomous systems.

The foundational concept of ReAct was formally introduced by Yao et al. [41], demonstrating that prompting an LLM to generate both reasoning traces and task-specific actions significantly improves its success rate in dynamic environments. Following this paradigm, several open-source initiatives, such as AutoGPT [7] and PentestGPT [8], have attempted to fully automate the penetration testing lifecycle, making significant pioneering contributions to the field.

However, applying these generalized ReAct frameworks to complex offensive security tasks often exposes significant operational bottlenecks. Because these early systems typically rely on a direct-coupling approach—where the LLM interacts directly with raw command-line outputs without an intermediate distillation layer—they are highly susceptible to data saturation. As documented by recent empirical studies on LLM agents [35],

when these systems are confronted with the noisy, verbose telemetry typical of security tools like NetExec [32] or Nmap [21], their reasoning traces can become diluted. Consequently, agents operating without a dedicated context-management middleware often experience severe operational brittleness, entering infinite execution loops or hallucinating parameters as they struggle to process uncompressed telemetry.

2.2. Context Management and the "Lost in the Middle" Phenomenon

A well-documented limitation of Transformer-based architectures [34] is the "Lost in the Middle" phenomenon, where an LLM's ability to recall and utilize information degrades significantly when relevant data is buried within a massive context window. The phenomenon was empirically formalized by Liu et al. [20], who demonstrated that LLMs excel at retrieving information positioned at the very beginning or end of their context window, but suffer a dramatic degradation in recall when the relevant data is located in the center of the prompt. In autonomous penetration testing, this manifests as *Horizontal Bloat*—the linear accumulation of iterative command histories and massive tool outputs.

While the broader NLP (Natural Language Processing) community has heavily invested in Retrieval-Augmented Generation (RAG) [17] to manage context limits, RAG is often inadequate for offensive security. Standard vector embeddings are optimized for semantic text search, but they frequently discard critical syntactic nuances—such as specific port numbers, hidden HTML tokens, or exact Active Directory ACL [4] relationships—necessary for exploitation. The state of the art lacks a domain-specific, programmatic distillation mechanism capable of preserving these security primitives while discarding structural noise.

2.3. Vulnerabilities in Autonomous Agents: Indirect Prompt Injection

As LLMs are granted autonomy to interact with external environments [23, 29], a novel class of security threats has emerged: Indirect Prompt Injection (IPI). The concept was first conceptualized by Greshake et al. [14], highlighting how adversaries can weaponize the target environment against the AI itself. Unlike traditional prompt injection—where a user directly inputs malicious instructions—IPI occurs when the LLM ingests untrusted data from a third-party source, such as a compromised website, a database field, or a

manipulated file on a target server.

Recent security evaluations [18] show that because LLMs lack a dedicated execution space separate from their data space, they process both system instructions and ingested target data through the same attention mechanism. If an adversary implants a payload within a target environment that the agent is programmed to scan, the model may elevate the ingested text to the status of a directive, leading to *Mission Hijacking*. While theoretical defenses such as data delimiters have been proposed, practical implementations demonstrating robust, Defense-in-Depth architectures for autonomous security agents remain an underexplored area of research.

3 | Proposed System Architecture

This chapter details the design and internal logic of the autonomous penetration testing orchestrator. The system is engineered to solve the efficiency gap in current LLM-based agents by introducing two primary innovation layers: Strategic Memory Management and Semantic Compression.

3.1. Framework Overview: The Autonomous Orchestrator

The proposed framework is an autonomous system designed to emulate the cognitive and operational workflow of a human penetration tester. At its core, the system acts as an *intelligent middleware* that bridges the gap between high-level strategic reasoning provided by Large Language Models (LLMs) and low-level execution via standard offensive security tools.

Unlike traditional automated scanners that follow a static execution tree, this orchestrator implements a dynamic **closed-loop architecture**. The system does not merely execute a sequence of commands; it interprets tool outputs, updates its internal state, and adapts its strategy based on the evolving security posture of the target environment.

As illustrated in Figure 3.1, the architecture is divided into three functional macro-layers:

- **Reasoning Engine (The LLM):** The engine receives the optimized context and generates the next logical step in the attack path. It operates as a "stateless" brain that relies entirely on the quality and relevance of the context provided by the orchestrator.
- **Context Management Layer (The Mediation Layer):** Often referred to as the "digestive system" of the framework, this layer receives raw data from the tools and applies the *Semantic Compression* and *Memory Hygiene* techniques discussed in the following sections. Its role is to ensure that only high-signal, low-noise information

reaches the LLM.

- **Tool Execution Layer:** This layer manages the interaction with the external security ecosystem (e.g., *Nmap*, *NetExec*, *BloodHound*). It is responsible for executing commands in the target environment, handling authentication sessions, and capturing raw output.

3.1.1. Design Evolution: From Naive Approaches to Orchestration

The current architecture is the result of an iterative research process, moving beyond simple script-based automation toward a structured orchestration. Documenting this evolution is essential to justify the complexity of the proposed framework and to highlight the specific challenges encountered during the development of autonomous security agents.

In the early stages of this research, a *Naive Agent* approach was implemented, where the LLM was directly exposed to raw tool outputs (e.g., long raw HTML response or extensive BloodHound [30] JSON dumps). This baseline testing highlighted two critical operational failure modes:

1. **Syntactic Saturation and Vertical Bloat:** Large outputs quickly consumed the model's *context window*, leading to the "Lost in the Middle" phenomenon. This caused the model to lose track of the initial mission objectives and the overall tactical strategy.
2. **Instruction-Data Confusion**: Without rigorous prompt structuring and semantic compression, the model fails to distinguish between trusted system instructions and raw tool outputs.This lack of structural formatting allows untrusted data to be misinterpreted as valid directives. As demonstrated in Chapter 6, this vulnerability leads to *Mission Hijacking*, where the agent erroneously follows termination commands found within the target environment, resulting in premature mission failure.

These empirical findings justified the rejection of a direct-coupling approach in favor

Figure 3.1: High-level architecture of the operational loop.

of a three-layered architecture, where the orchestrator acts as a deterministic filter and a guardian of context integrity.

3.2. Semantic Compression Layer

The *Semantic Compression Layer* is the architectural component designed to mitigate *Vertical Bloat*. In offensive security contexts, tool outputs often contain a high volume of metadata, syntax, and redundant information that is irrelevant to high-level strategic reasoning. By filtering this noise before it reaches the LLM, the framework ensures that the model's limited context window is occupied only by useful information coming from the tool's output. Instead of a generic summarization, the parser uses a **Domain-Specific Dispatcher** approach: it identifies the tool's signature and applies localized regex-based heuristics to extract only actionable security primitives. Algorithm 3.1 illustrates this high-level distillation process.

Algorithm 3.1 Structural Parsing and Semantic Distillation

```
1: procedure OutputParser(tool_id, raw_stdout)
2:     structured_results ← new Dictionary()
3:     clean_text ← RemoveANSIAndHeaders(raw_stdout)
4:     if tool_id = "nmap" then
5:         structured_results ← ExtractPortsAndStates(clean_text)
6:     else if tool_id = "nxc" ∨ tool_id = "smbclient" then
7:         structured_results ← IdentifySharesAndCredentials(clean_text)
8:     else if tool_id = "gobuster" ∨ tool_id = "nikto" then
9:         structured_results ← MapFoundResourcesAndStatus(clean_text)
10:    else if tool_id = "GetUserSPNs" ∨ tool_id = "GetNPUsers" then
11:        structured_results ← ExtractCryptographicHashes(clean_text)
12:    else if tool_id = "..." then
13:        ...
14:    end if
15:    return structured_results
16: end procedure
```

Although a specific parser function has been implemented for every security tool within the framework, this analysis focuses on the BloodHound and Web parser modules. These two cases are particularly representative, as they target the most significant sources of *Vertical Bloat*, demonstrating the highest impact in terms of token reduction and information density.

3.2.1. BloodHound Parser: JSON Distillation

Active Directory (AD) reconnaissance via BloodHound [28] typically generates massive JSON files representing thousands of nodes and relationships. Injecting such raw data into an LLM prompt is infeasible due to token limits and the model's inability to parse complex nested structures efficiently at scale.

To solve this, we implemented a specialized **BloodHound Parser**. This module performs a programmatic extraction of strategic attack paths. Instead of the raw files, the LLM receives a distilled textual representation focusing on:

- **High-Value Targets:** Identification of Domain Admins, Domain Controllers, and Tier-0 assets.
- **Exploitable Relationships:** Direct paths such as *MemberOf*, *HasSession*, or *AdminTo*.
- **Complex ACL Paths:** Transitive permissions like *GenericAll*, *WriteDacl*, or *AddMember* that allow for privilege escalation.

As shown in our experimental results (Figure 5.1), this distillation reduces the cumulative token footprint of the AD reconnaissance phase by over 85%. Upon executing the enumeration tool at step 5, the ablated framework experiences a massive context spike up to 37,500 tokens, whereas our optimized BloodHound parser keeps the overall prompt size under 4,500 tokens.

3.2.2. Web Semantic Wrapper: HTML Distillation

Similarly, in web penetration testing, raw HTTP responses (HTML/JS/CSS) introduce significant noise. The **Web Semantic Wrapper** acts as a pre-processor for web interactions.

The wrapper does not merely forward the HTTP body to the LLM. Instead, it performs a **Semantic Distillation** of the page content:

- **Noise Removal:** Stripping of CSS styles, script blocks, and repetitive boilerplate (header/footer).
- **Feature Extraction:** Programmatic identification of HTML forms, input fields, and interactive links.
- **Context Padding:** Truncating large bodies while preserving security-relevant headers (e.g., Set-Cookie, Security Headers).

By supplying the LLM with a structured inventory of *Interactable Elements* instead of the raw DOM, the framework effectively prevents the model from losing focus during complex, multi-step exploitation chains. As demonstrated by our experimental results (Figure 5.2), performing this semantic extraction on the raw HTTP body response significantly reduces token consumption, ensuring a more efficient and cost-effective web assessment.

3.3. Strategic Memory Management

A critical limitation of current LLM-based agents is the *Horizontal Bloat*, characterized by the linear accumulation of command history and tool outputs within the prompt context. As the assessment progresses, the inclusion of redundant or irrelevant historical data leads to the "Lost in the Middle" phenomenon, where the model's reasoning capabilities degrade as key information is buried under tactical noise. To counter this, the framework implements a two-tiered memory architecture.

3.3.1. The Recon Summary Concept

Instead of providing the LLM with a full, unedited history of every interaction, the system maintains a dynamic **Recon Summary**. This component acts as a continuously updated state-of-the-art of the assessment, including but not limited to:

- **Identified Assets:** A distilled list of discovered hosts, services, and vulnerabilities.
- **Proven Attack Paths:** Successful pivots or credentials obtained during the session.
- **Current Objective:** The high-level goal derived from the initial tasking.

By replacing the raw command history with this summarized state, the framework maintains a near-constant prompt size, as demonstrated in our ablation studies (see Figures 5.3 and 5.4), effectively lowering the risk of context saturation in long-duration engagements.

3.3.2. Memory Hygiene

The summary is managed via a specialized synthesis function that implements what we define as *Memory Hygiene*. The retrieval logic, formalized in Algorithm 3.2, does not simply query the latest database entries; it applies a heuristic filter to balance the density of information against the available context window.

To prevent the agent from entering infinite loops—a common failure mode in au-

tonomous systems—the orchestrator specifically highlights the last n successful and failed attempts. By explicitly setting a small value for $limit_{err}$, we ensure that the model is confronted with its immediate mistakes, forcing it to "reason" over the causes of failure (e.g., "Syntax Error" or "Command Reuse") and pivot to an alternative strategy rather than repeating the same unsuccessful command.

Algorithm 3.2 Strategic Context Retrieval (Recon Summary)

```
1: procedure GetReconSummary(memory_db)
2:     context ← new Dictionary()
3:     limit_err ← n_1                          ▷ Heuristic limit for recent failed attempts
4:     limit_succ ← n_2                         ▷ Heuristic limit for recent successful actions
5:     bloat_keywords ← {"nmap", "gobuster", "nikto", "shares", ...}
6:     errors ← QueryLatestErrors(memory_db, limit_err)
7:     successes ← QueryLatestSuccesses(memory_db, limit_succ)
8:     for all entry ∈ successes do
9:         if ∃k ∈ bloat_keywords such that k ∈ entry.command then
10:            entry.results ← "Omitted: Context preserved in global state"
11:        else
12:            entry.results ← ParseStructuredData(entry.results)
13:        end if
14:        context.add(entry)
15:    end for
16:    return context ∪ errors
17: end procedure
```

Furthermore, the keyword-based omission of results for high-verbosity tools like `nmap` or `gobuster` is a critical architectural choice. Since the core findings (e.g., open ports) are already integrated into the *Assessment Knowledge Base* section of the summary, repeating the raw tool output in the command history would be redundant. This programmatic "muting" of redundant data is what allows the framework to maintain operational focus even in complex, multi-stage assessments.

3.4. The Decision Loop: Command Chain Integrity and Input Validation

The framework's execution follows an iterative cycle inspired by the **OODA loop** (Observe, Orient, Decide, Act), specifically adapted for autonomous cyber operations, as illustrated in Figure 3.2:

Observe: Execution of a security tool and collection of raw results from the target network or application.

Orient: Pre-processing and parsing of the raw output via the *Semantic Compression Layer* and updating the *Recon Summary* to maintain an up-to-date state of the assessment.

Decide: Presenting the summarized context and current findings to the LLM to determine the next tactical objective or pivot point.

Act: Translating the LLM's high-level decision into a correctly formatted command-line execution for the next tool in the chain.

A critical component of the **Act** phase is the *Command Sanitization and Validation* layer. To ensure **Command Chain Integrity**, the orchestrator does not blindly execute the strings generated by the LLM. Instead, it passes the proposed command through a validation engine that performs two main checks:

1. **Syntactic Validation:** Cross-referencing the command against a whitelist of supported security tools and their respective parameter schemas to prevent execution errors.
2. **Execution Deduplication and State Awareness:** Verifying whether the proposed command has already been successfully executed during the current assessment. If a redundant command is detected, the orchestrator intercepts the execution and returns an internal system notification. This feedback explicitly reminds the LLM that the requested data has already been gathered and redirects its attention to the *Recon Summary* or global state, effectively preventing infinite execution loops and redundant network traffic.

This decoupling of reasoning and execution ensures that the framework remains tool-agnostic and resilient to the information density typical of modern corporate networks, particularly in complex scenarios like Active Directory.

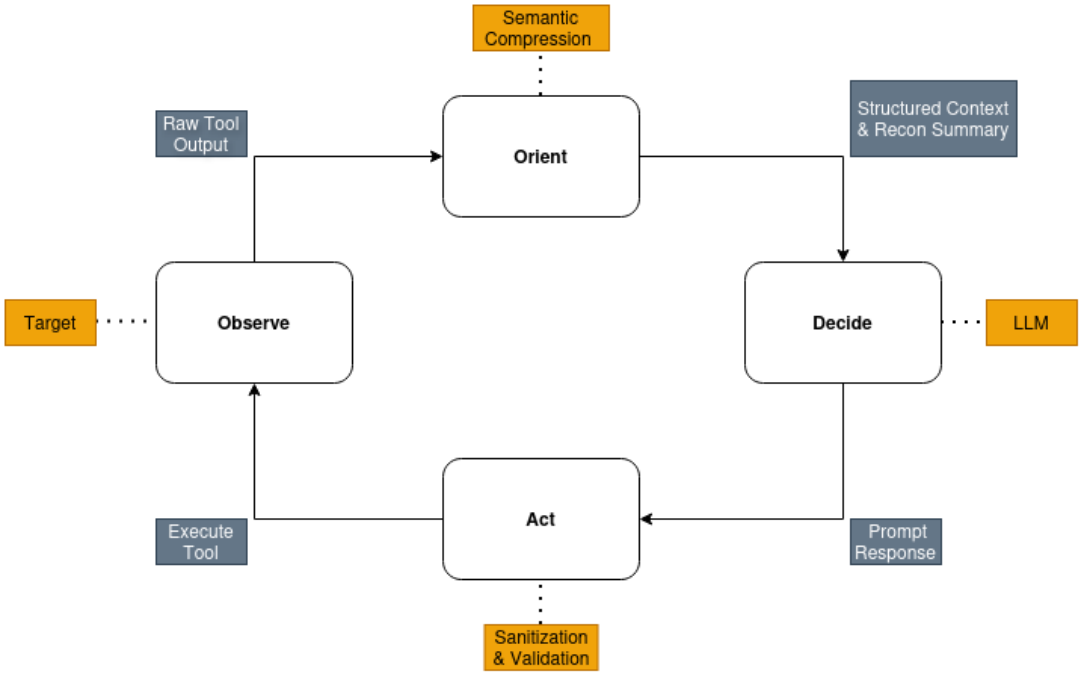

Figure 3.2: High-level overview of the decisional loop.

4 | Implementation and System Engineering

This chapter details the software engineering principles, architectural choices, and concrete implementation of the autonomous penetration testing framework. Rather than a monolithic script, the system was designed as a highly decoupled, asynchronous microservice architecture, ensuring scalability and fault tolerance during long-running network scans.

4.1. System Overview and Human-in-the-Loop Dashboard

A critical engineering requirement for autonomous offensive security tools is *Observability*. Deploying a "black-box" AI agent in a corporate network poses severe operational risks. To address this, the framework is built around a real-time, interactive frontend dashboard that acts as the command center for the entire architecture.

This interface transforms the system from a fully autonomous background script into a **Human-in-the-Loop (HITL)** platform, allowing security operators to supervise, pause, or terminate the agent's actions if it deviates from the rules of engagement.

As shown in Figure 4.1, the dashboard provides a comprehensive macro-view of the active assessment. Key observability features include:

- **Live Telemetry and FinOps:** Real-time visualization of the API cost (USD), total consumed tokens, and execution time. This ensures the operator is constantly aware of the economic impact of the ongoing pentest.
- **Enumeration Feed:** A dynamic table that populates as soon as the agent successfully validates an open service or extracts a credential, allowing immediate human verification.
- **Recent Operation Log:** A dedicated terminal-like view that exposes the most

recent executed commands.

Figure 4.2 and 4.3 illustrates the *Findings* interface, which serves as the primary observability feed for the human operator. Rather than requiring the user to manually parse through raw tool outputs or verbose terminal logs, this tab dynamically aggregates the actionable intelligence generated by the semantic compression layer. As the autonomous agent progresses through the assessment, any successfully validated vulnerability, discovered web endpoint, or extracted credential is asynchronously pushed to this view.

This centralized aggregation is essential for effective Human-in-the-Loop (HITL) operations, allowing security engineers to instantly verify critical discoveries and assess the target's risk posture in real-time without interrupting the agent's cognitive loop. To power this interactive, non-blocking user interface, the underlying backend required a specialized asynchronous architecture, which is detailed in the following sections.

4.2. Core Architecture: Asynchronous API Server

To provide a clear overview of the interaction between the system's components, Figure 4.4 illustrates the high-level architectural workflow. The diagram highlights the asynchronous nature of the framework, where the API Server acts as a bridge between the front-end Dashboard and the back-end Orchestrator engine.

At the core of the framework lies a RESTful API server implemented in Python using the `Flask` framework. The primary engineering challenge in developing an autonomous security agent is managing execution latency. Network enumeration tools (such as Nmap or Gobuster) and LLM inference calls can take several minutes to complete. A synchronous design would result in severe bottlenecks, HTTP timeouts, and an unresponsive user interface.

To solve this, the API server employs an asynchronous threading model. When an assessment is launched, the server offloads the heavy reasoning and execution loop to a dedicated background `threading.Thread`, instantly returning a 200 OK status to the client. This architecture allows the frontend dashboard to continuously poll the server for state updates, logs, and new vulnerabilities without interrupting the agent's cognitive loop.

The complete Python implementation of the asynchronous assessment trigger and the live monitoring endpoints are provided for reference in **Appendix A.1**.

Algorithm 4.1 Core Cognitive Iteration Loop (Orchestrator)

```
procedure RunAgentStep(memory_db, config)
                                                  ▷ 1. State Retrieval & Compression
    recon_summary ← GetReconSummary(memory_db)
    services ← GetDiscoveredServices(memory_db)
                                                  ▷ 2. Dynamic Prompt Construction
    prompt ← BuildPrompt(config, recon_summary, services)
                                                  ▷ 3. LLM Inference
    agent_response, token_usage ← LLM.Generate(prompt)
    parsed_action ← Parse(agent_response)
                                                  ▷ 4. Action Routing & Validation
    if parsed_action.type == "RUN_TOOL" then
        cmd ← ValidateCommand(parsed_action.tool, parsed_action.params)
        if IsDangerousOrRedundant(cmd) then
            SaveMemory(cmd, "Command Blocked")
            return Error
        end if
                                                  ▷ 5. Execution and Parsing
        raw_output ← ExecuteSystemProcess(cmd)
        parsed_results ← OutputParser(parsed_action.tool, raw_output)
                                                  ▷ 6. State Persistence
        UpdateInternalState(parsed_results)
        SaveMemory(cmd, parsed_results, token_usage)
        return Continue
    else if parsed_action.type == "STOP" then
        return Assessment Completed
    end if
end procedure
```

4.3. The Orchestrator Engine: State and Execution Loop

The `Orchestrator` class acts as the "brain" of the framework. It bridges the reasoning layer (the LLM) with the execution layer (the underlying operating system and security tools). The orchestrator's core mechanism is the cognitive iteration loop. During each step, the orchestrator compiles the current network state from the `MemoryManager`, constructs the dynamic prompt, queries the LLM via the `LLMClient`, and strictly validates the returned commands to prevent hallucinated or dangerous system executions.

As formalized in Algorithm 4.1, the framework enforces a rigid, programmatic pipeline (the full Python implementation is available in **Appendix A.2**). By intercepting the LLM's output and forcing it through the validation and parsing layers, the system in-

herently mitigates command injection risks and ensures that the LLM interacts with the host environment safely.

4.4. Semantic Compression Implementation

The *Semantic Compression* layer is the core technical innovation of the framework. It is implemented through dedicated Python classes that intercept raw, verbose tool outputs and distill them into optimized, token-efficient textual representations before they reach the LLM's context window.

4.4.1. Active Directory Graph Extraction (BloodHound)

In the Active Directory scenario, the framework utilizes the `BloodHoundAnalyzer` class to process massive JSON arrays generated by the BloodHound ingestor. The true value of this component lies in its ability to discard irrelevant Active Directory attributes and reconstruct a purely relational text-based graph of privileges.

The analyzer parses the Access Control Entries (ACEs), filters only the actionable rights (e.g., *GenericAll*, *WriteDacl*, *ForceChangePassword*), and maps them into a concise source-to-target dictionary. As demonstrated by the source code provided in **Appendix A.3**, by transforming the raw JSON into a specific `defaultdict` structure, the framework eliminates thousands of lines of metadata, presenting the LLM with a clean mapping of who can attack whom.

4.4.2. Web Application HTML Distillation

Web application assessments require a completely different compression strategy. Raw HTML bodies are notoriously token-heavy, filled with structural tags (`<div>`, `<span>`), inline CSS, and JavaScript that provide zero value to an exploitation reasoning loop.

To counter this, the framework employs a dedicated DOM parser [27] that strips away the visual presentation layer, extracting only the semantically interactive elements: forms, input fields, and actionable links. The parsing logic, fully detailed in **Appendix A.3**, is what enables the `SessionManager` to interact with complex web states. By receiving only the extracted `forms` and `inputs` array, the LLM can precisely format its next HTTP request (e.g., SQLi payloads or credential stuffing) without suffering from context saturation.

4.5. Strategic Memory and State Persistence

To implement the *Memory Hygiene* and *Recon Summary* concepts detailed in Chapter 3, the orchestrator requires a robust, persistent storage layer. To address this requirement, the framework implements a `MemoryManager` backed by a lightweight, local SQLite database.

This component strictly separates the agent's active reasoning state from the historical data storage. As detailed in the schema initialization code (**Appendix A.4**), the database captures not only the executed commands (`cmd`) and their outputs (`results`), but also essential FinOps telemetry (`usage_tokens`, `cost_estimated`). When the `Orchestrator` invokes the `get_recon_summary()` method, the `MemoryManager` simply executes a set of optimized SQL `SELECT` queries, fetching only the most recent command iterations (`ORDER BY timestamp DESC`) and the globally discovered credentials. This architecture guarantees that the LLM is always provided with a perfectly synchronized, deduplicated snapshot of the network state.

4.6. LLM Integration and Model Agnosticism

A fundamental architectural principle of the framework is the strict decoupling of the reasoning engine from the execution and orchestration layers. Rather than embedding specific API calls directly within the `Orchestrator`, all AI inferences are routed through an abstracted `LLMClient` interface.

This design choice guarantees **Model Agnosticism**. While the experimental evaluation (Chapter 5) utilizes Google Gemini models for benchmarking purposes, the architecture is intrinsically scalable to any Large Language Model. This is particularly critical in offensive security, where sending sensitive corporate network data (e.g., BloodHound graphs or proprietary source code) to a cloud provider often violates compliance and privacy policies.

The abstracted `LLMClient` wrapper (whose complete implementation is available in **Appendix A.5**) defines a `generate()` method that acts as a universal contract. To migrate the framework to a fully offline, air-gapped corporate network using local open-weight models (such as LLaMA [33] or Qwen3 [40]), a security engineer would only need to replace the client initialization inside this single class. The complex logic of the `Orchestrator`, the `MemoryManager`, and the *Semantic Parsers* remains completely untouched. This abstraction ensures long-term maintainability, allowing the framework to seamlessly scale and adopt future generations of LLMs without requiring architectural refactoring.

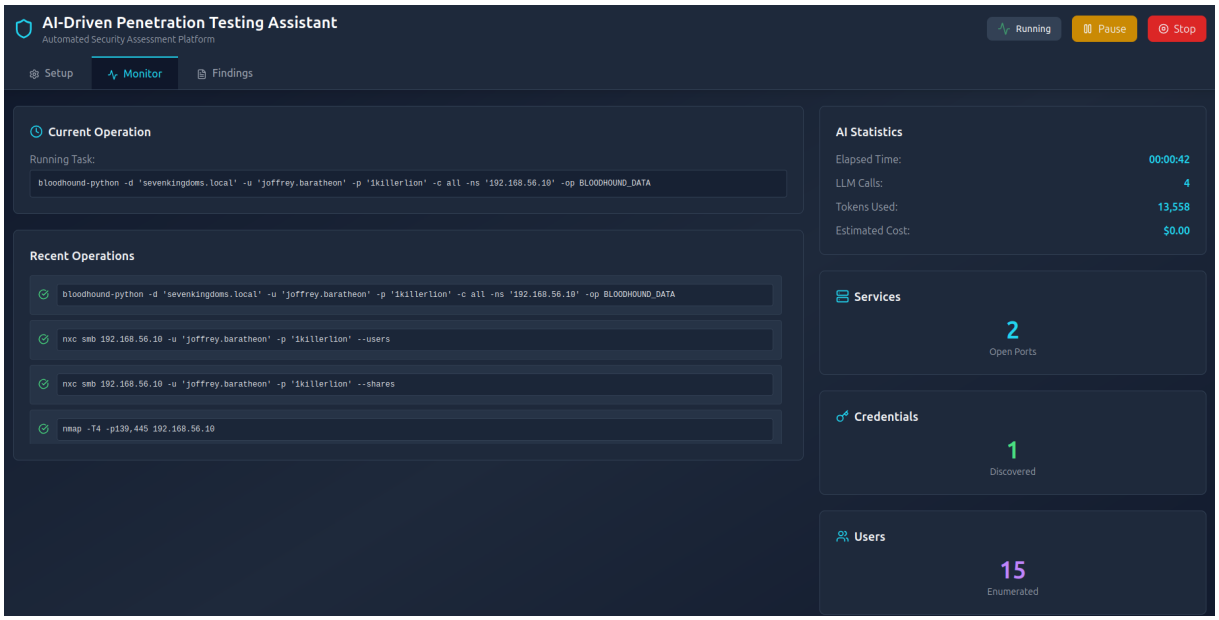

Figure 4.1: The main monitoring dashboard displaying real-time assessment status, active vulnerabilities, and live economic metrics.

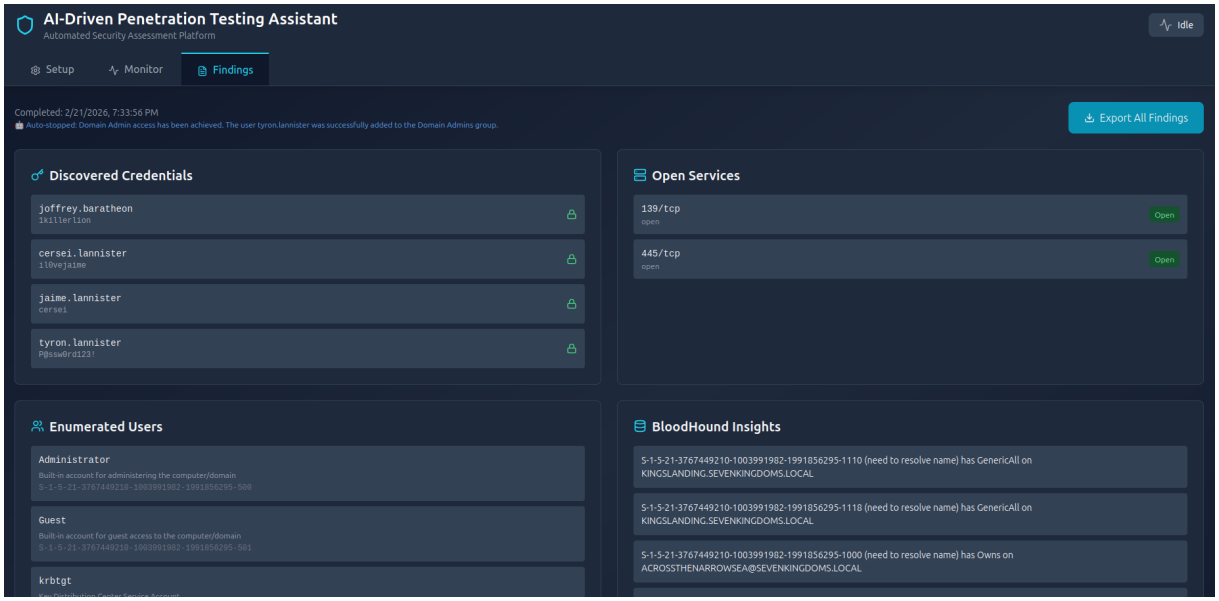

Figure 4.2: Findings Tab Scenario AD

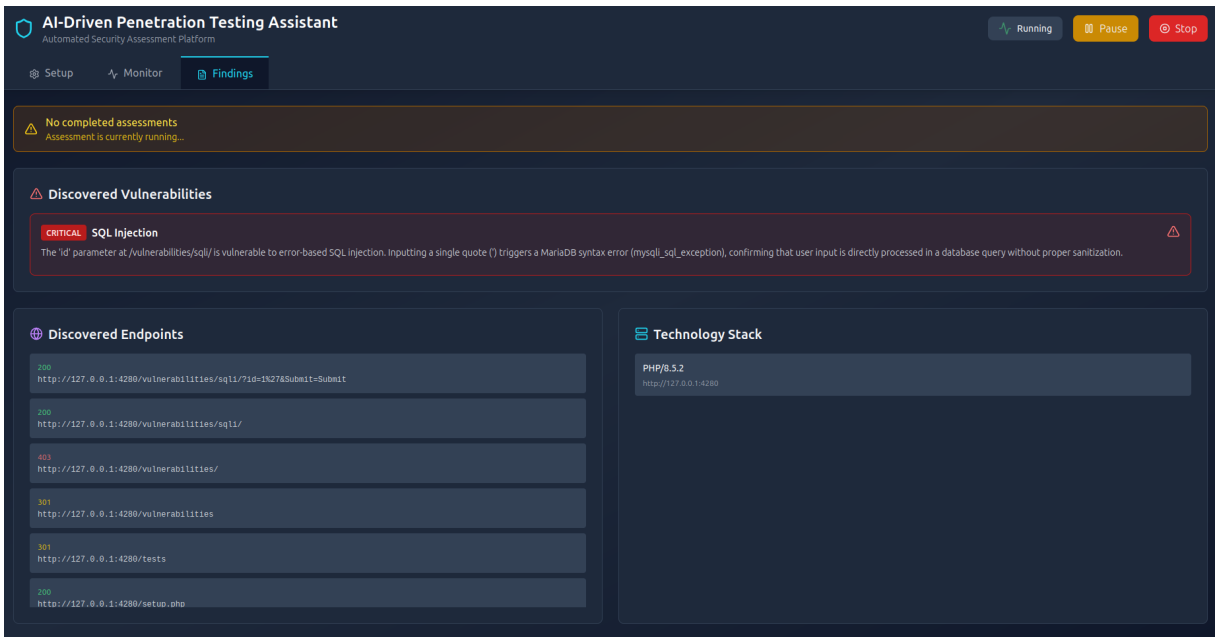

Figure 4.3: Findings Tab Scenario Web

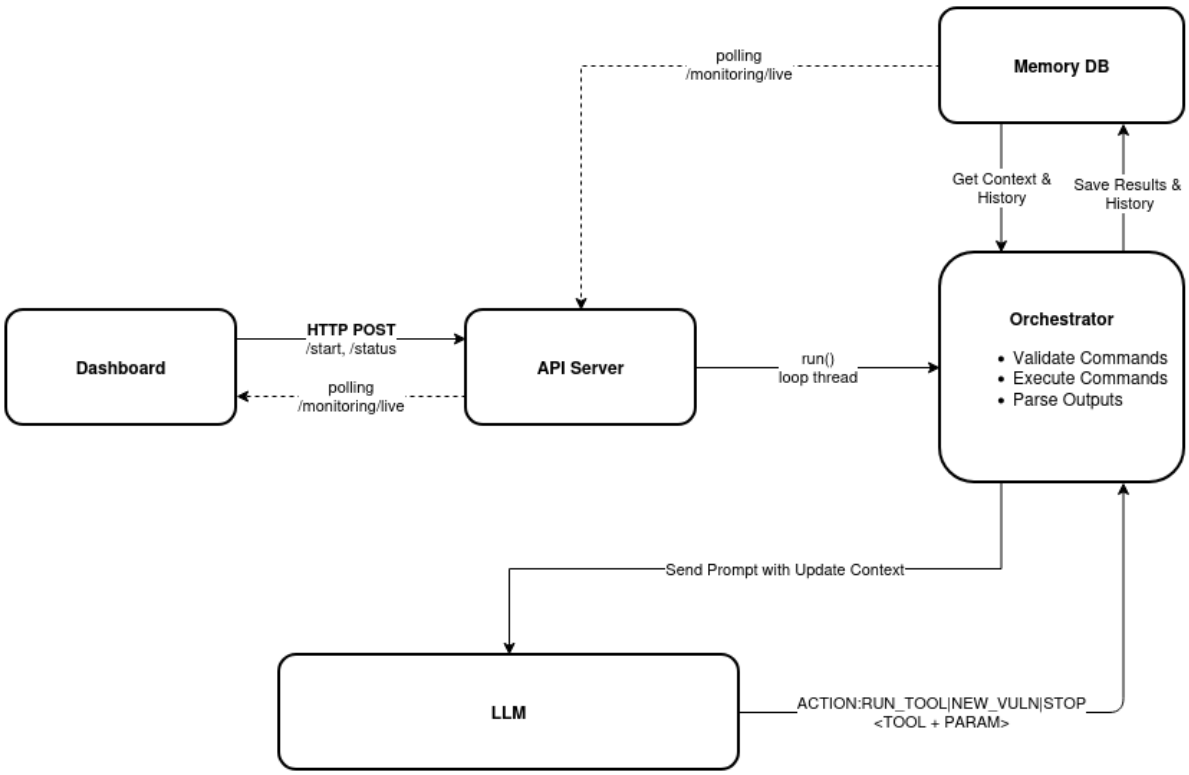

Figure 4.4: Architectural workflow and component interaction loop, highlighting the asynchronous communication between the Dashboard, API Server, and the Orchestrator engine.

5 | Experimental Evaluation: Ablation Study

This chapter presents the empirical validation of the proposed orchestrator. Through a series of ablation studies and systematic tests, we evaluate the impact of the Semantic Compression and Memory Management layers on the agent's performance, cost-efficiency, and strategic resilience in complex environments.

5.1. Experimental Setup

To ensure reproducibility and rigorous benchmarking, the evaluation was conducted in a controlled environment designed to simulate modern enterprise security challenges.

5.1.1. Test Environments

The framework was tested against two primary scenarios:

- **Active Directory Lab:** A virtualized corporate network featuring a Windows Server 2022 Domain Controller, multiple users, and common misconfigurations (e.g., Kerberoasting [22], AS-REP Roasting [24], and insecure ACLs. The Goal in this scenario for the LLM is to become a Domain Admin starting from a non-privileged user using offensive security tools such as bloodyAD [6].
- **Vulnerable Web Lab:** A web application containing OWASP Top 10 vulnerabilities (e.g., SQLi, RCE, XSS, etc) [10]. The Goal in this scenario for the LLM is to correctly identify and classify at least 4 web vulnerabilities

5.1.2. Model Configuration

The framework utilizes **Google Gemini 2.5 Flash** and **Gemini 3 Flash** [12] as primary reasoning engines. The integration is handled via a custom `LLMClient` wrapper based on the official `google-genai` SDK [13].

To ensure a rigorous quantitative evaluation, our analysis focuses primarily on **Prompt Token consumption**. This metric is chosen as the lead KPI because it directly represents the occupancy of the model's *context window* and the cumulative growth of the session history (Vertical and Horizontal Bloat).

The system also includes a real-time **Cost Monitoring** logic. By parsing the `usage_metadata`, the framework calculates the economic impact of each assessment based on current API pricing ($0.30–$0.50 per 1M input tokens), providing the financial benchmarks discussed in the following sections.[3]

It is critical to note that Large Language Models, particularly those accessed via cloud APIs in 'preview' stages, are subject to continuous, unannounced updates by their providers. Therefore, all empirical evaluations, token consumption metrics, and benchmarking runs presented in this research were conducted using the `gemini-2.5-flash` and `gemini-3-flash-preview` endpoints during early 2026. This approach captures a specific, frozen snapshot of the model's reasoning capabilities, ensuring internal consistency across all ablation studies, despite the inherently mutable nature of cloud-based AI services.

5.2. Ablation Study 1: Semantic Compression

The first study evaluates the mitigation of *Vertical Bloat* by comparing the performance of the orchestrator using the specialized **Semantic Parser** versus the injection of raw bloodhound and web requests outputs.

The results, illustrated in Figures 5.1 and 5.2, demonstrate a radical divergence in context window occupancy. While raw data ingestion leads to immediate saturation, the use of parsers significantly helps keeping a lean footprint, preserving the model's ability to process long-term dependencies.

Furthermore, it was observed that following the onset of vertical token bloat, the model's performance deteriorated significantly in both scenarios. This degradation manifested as a substantial increase in inference latency and a precipitous decline in strategic reasoning accuracy.

5.2.1. Comparative Quality and Efficiency Metrics

Since both the optimized and ablated single-run executions successfully reached the assessment goal (`ACTION:STOP`), the quality degradation cannot be measured in absolute failure rates. Instead, we analyzed the **Cognitive Overhead** and the **Context Satu-**

ration Trajectory.

Table 5.1: Single-Session AD Compression Ablation

Metric	Optimized Framework	Ablated (No Parser)
Peak Prompt Size (Tokens)	4,418	38,339
Thinking Tokens (Peak)	3,923	7,093
Context Saturation Risk	Low	Critical
Information Extraction	Targets/ACL/Relationships	Raw AD Node Properties

As detailed in Table 5.1, the Active Directory environment highlighted the catastrophic scaling problem of un-orchestrated agents. When the ablated agent executed a domain-wide reconnaissance, the prompt size exploded to over 38,000 tokens. The model was forced to process thousands of lines of non-functional JSON metadata (e.g., AD object creation timestamps, unmodified ACLs, and null property fields).

Conversely, the optimized framework successfully distilled this massive dataset into a condensed representation, stabilizing the context at approximately 4,400 tokens.

Most notably, the **Cognitive Overhead**—measured via the model's *Thinking Tokens*—nearly doubled in the ablated run (from roughly 3,900 to over 7,000). While both agents successfully achieved the Domain Admin objective, they executed fundamentally different attack paths. Overwhelmed by the 38k-token spike of raw BloodHound JSON, the naive agent suffered from *attention collapse*. It was completely unable to identify the complex Access Control List (ACL) relationships hidden within the unformatted data. Consequently, it bypassed the graph analysis entirely and fell back on a standard Ker-

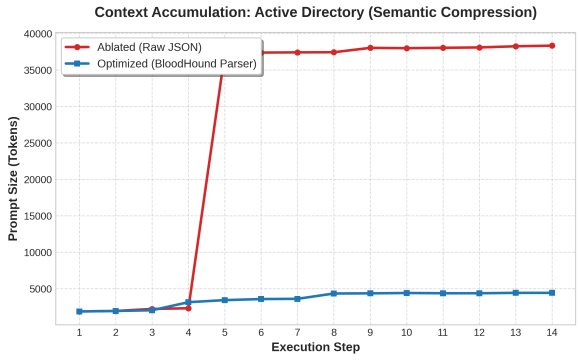

Figure 5.1: Comparison of token consumption: Raw BloodHound JSON output vs. Semantically Parsed text.

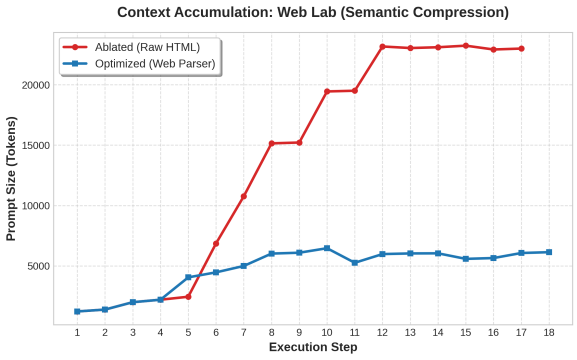

Figure 5.2: Comparison of token consumption: Raw HTML output vs. Semantically Parsed text.

beroasting attack—a vector that relies on service principal names and weak passwords rather than deep object-relation parsing.

Conversely, the optimized agent, receiving the distilled data, easily identified the vulnerable ACL misconfigurations and executed a targeted, stealthier privilege escalation path. This divergence in strategy empirically demonstrates that direct ingestion of massive enumeration logs effectively "blinds" the LLM to sophisticated, graph-based attack paths, forcing it to downgrade its tactical choices due to context saturation. Furthermore, beyond this immediate tactical degradation, the 38k-token spike exposes a fatal flaw in scalability: in a larger corporate network, this unoptimized approach would instantly breach the model's context window, leading to critical amnesia of the mission's initial objectives.

Table 5.2: Single-Session Web Compression Ablation

Metric	**Optimized Framework**	**Ablated (No Parser)**
Peak Prompt Size (Tokens)	6,465	23,234
Thinking Tokens (Peak)	1,793	1,069
Context Saturation Risk	Low	Critical
Information Extraction	Targeted (Structured JSON)	Raw HTML/Garbage

As evidenced in Table 5.2, the absence of the Semantic Compression layer in the Web Scenario caused again a *Vertical Bloat*. During the vulnerability discovery phase, the ablated agent was forced to process entire DOM structures, causing the prompt size to skyrocket to more than 23,000 tokens in just a few iterations. In contrast, the optimized framework maintained a stable and highly efficient context footprint of approximately 6,500 input tokens by extracting only the relevant attributes (e.g., `status_code`, `forms_detected`, `post_parameters`).

While the underlying LLM managed to parse the 23k-token prompt without a hard crash, this exponential growth trajectory is unsustainable for realistic, long-duration engagements.

Interestingly, the ablation study in the web scenario revealed an inverse correlation regarding the *Thinking Tokens*. The optimized agent utilized significantly more reasoning capacity (peaking at 1,793 thinking tokens) compared to the ablated agent (1,069 tokens). This suggests that when the LLM is fed highly structured, noise-free data, it allocates its computational overhead towards deep strategic reasoning (e.g., crafting complex SQL

injection payloads). Conversely, the massive 23k-token prompt induced a "shallow reasoning" state, where the model's attention mechanism was overwhelmingly consumed by syntactic parsing rather than tactical planning.

The ablation test empirically demonstrates that without the parser, the agent operates perpetually on the edge of its context window limit, drastically increasing latency, API costs, and the risk of triggering the "Lost in the Middle" phenomenon during subsequent lateral movement steps.

5.3. Ablation Study 2: Impact of Memory Hygiene and The "Cognitive Drag" Phenomenon

A second ablation study was conducted in both the Active Directory and Vulnerable Web Lab scenarios to evaluate the *Strategic Memory Management* layer. By disabling the Recon Summary and its heuristic filtering, the system reverted to a naive "Full Memory" approach, forcing the LLM to ingest the entire, unfiltered command history at every iterative step.

This test was designed to quantitatively measure the impact of *Horizontal Bloat* (the linear accumulation of past actions and tool results) on the agent's reasoning capacity.

Table 5.3: Single-Session Memory Hygiene Ablation

Metric (Active Directory)	**Optimized (Recon Summary)**	**Ablated (Full Memory)**
Peak Prompt Size (Tokens)	4,478	8,314
Thinking Tokens (Peak)	1,961	10,773
Metric (Web Scenario)		
Peak Prompt Size (Tokens)	6,679	12,939
Thinking Tokens (Peak)	1,424	1,581

As detailed in Table 5.3, the absence of the Recon Summary did not induce absolute mission failure—both the optimized and ablated agents successfully reached their respective objectives.

However, the ablated logs reveal a severe case of what we define as **Cognitive Drag**.

Without the programmatic "muting" of historical data, the prompt size escalated linearly step-by-step, essentially doubling the context footprint in both scenarios (as visu-

alized in the token accumulation trajectories in Figures 5.3 and 5.4). The most striking empirical finding, however, is observed in the model's internal *Thinking Tokens* during the Active Directory assessment.

Forced to continually re-process a growing wall of historical tool outputs just to determine its next move, the naive agent's reasoning overhead skyrocketed to 10,773 thinking tokens at its peak. This indicates that the LLM's underlying attention mechanism was overwhelmingly taxed merely to maintain state awareness and avoid command repetition.

In contrast, the optimized agent—equipped with the dynamically distilled Recon Summary—achieved the exact same tactical objective using only 1,961 thinking tokens, representing an **81% reduction in cognitive overhead** (Figure 5.5).

This empirical data definitively proves that the Strategic Memory layer is not merely a mechanism for reducing API costs. It acts as an essential *cognitive offloader*. By continuously sanitizing the active context window, the framework prevents the LLM from expending vast computational resources on historical parsing, thereby preserving its reasoning capacity for complex, forward-looking tactical pivoting.

5.4. Ablation Study 3: Cross-Model Comparison

Lastly, we benchmarked the framework across two different iterations of the Google LLM family (**Gemini 2.5 Flash** and **Gemini 3 Flash**) to evaluate how generational model improvements intersect with our autonomous pentesting architecture. Specifically, we compared the performance of each model in a baseline setup (without parsers or memory management) versus the fully optimized framework.

Contrary to the hypothesis that a more advanced model could natively overcome context

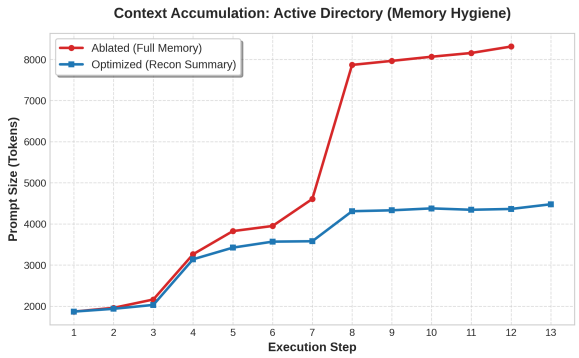

Figure 5.3: Token accumulation trajectory in the Active Directory scenario (Ablation 2).

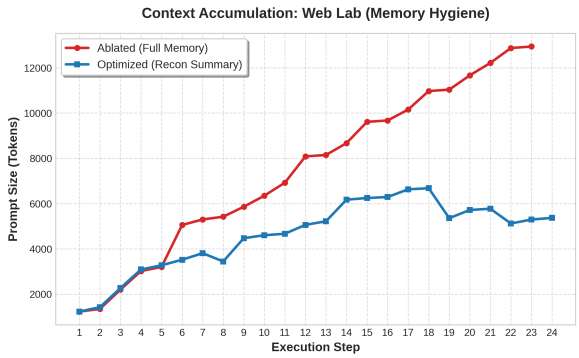

Figure 5.4: Token accumulation trajectory in the Web Lab scenario (Ablation 2).

bloat, the logs revealed that both models successfully reached the `ACTION:STOP` objective in the unoptimized runs, but at an unsustainable computational cost.

Table 5.4: Cross-Model Peak Prompt Tokens (Single-Session Ablation)

Scenario & Model	Ablated (Unoptimized)	Optimized Framework
Active Directory		
Gemini 2.5 Flash	$\sim 50,000$	$\sim 5,000$
Gemini 3 Flash	$\sim 40,000$	$\sim 5,000$
Web Vulnerability Lab		
Gemini 2.5 Flash	$\sim 61,000$	$\sim 7,800$
Gemini 3 Flash	$\sim 33,000$	$\sim 7,500$

As detailed in Table 5.4, without the Semantic Compression layer, Gemini 2.5 Flash suffered a massive context explosion in the Web scenario, peaking at 61,637 tokens. While the newer Gemini 3 Flash demonstrated superior native attention mechanisms (halving the unoptimized bloat to roughly 33k tokens), both models operated dangerously close to context saturation.

However, the introduction of the framework's optimization layers acted as a **Great Equalizer**. By programmatically distilling tool outputs into organized structures, the orchestrator stabilized the context footprint of both models to under 8,000 tokens, entirely neutralizing the context-handling advantage of the more expensive Gemini 3 model.

The comparison between the optimized runs, as illustrated in Figures 5.6 and 5.7, highlighted distinct domain-specific strengths for each model. In **Active Directory** assessments, the framework's extraction of graph-based relational data allowed **Gemini 2.5 Flash** to emerge as the optimal choice. It provided a superior *cost-performance ratio*, executing the exact same logical attack paths as its successor but at a fraction of the API expenditure, making it highly viable for large-scale, automated network reconnaissance.

Conversely, the **Web-based scenarios** showcased the advanced semantic reasoning capabilities of **Gemini 3 Flash**. Web vulnerabilities (such as analyzing nuanced HTTP responses for SQLi or DOM-based XSS) require a deeper interpretation of unstructured payloads. In this domain, the newer model demonstrated higher precision and required fewer iterative attempts to validate complex exploitation chains, justifying its higher API cost when dealing with application-layer security.

5.5. Quantitative Analysis and Success Rates

To provide a statistically significant evaluation, we conducted a suite of **200 automated runs** distributed across Active Directory (network) and Web application scenarios. This benchmarking allows us to evaluate the framework's reliability, economic efficiency, and performance across different model generations under a sustained workload, whose aggregate results are summarized in Table 5.5.

Table 5.5: Aggregate Performance Metrics across 200 Automated Runs

Model & Scenario	Success Rate	Avg. Tokens/Run	Avg. Cost ($)	Avg. Time (s)
Gemini 2.5 (AD)	100%	$\sim 65k$	0.013	168
Gemini 3 (AD)	100%	$\sim 64k$	0.019	142
Gemini 2.5 (Web)	52%	$\sim 105k$	0.032	205
Gemini 3 (Web)	100%	$\sim 72k$	0.032	135

5.5.1. Success and Reliability Metrics

As illustrated in Figure 5.8, the success rate highlights a clear divergence in reasoning depth. While both models achieved a 100% success rate in Active Directory scenarios—where the tools parser provides highly structured relational data—Gemini 2.5 Flash struggled in web environments. In this domain, the older model failed in approximately 50% of the attempts, often failing to correlate complex LFI or SQLi responses. In contrast, Gemini 3 Flash maintained perfect effectiveness, confirming its superior semantic understanding of web application logic.

5.5.2. Economic Efficiency and Latency

The economic analysis confirms the framework's operational viability. By applying the pricing tiers ($0.30/1M input tokens for 2.5 Flash vs. $0.50/1M for 3 Flash), the data shows that for network assessments, Gemini 2.5 Flash is the optimal choice, providing identical results at 50% of the cost. However, for web tasks, the higher expenditure of Gemini 3 Flash is justified not only by the success rate but also by its efficiency: Gemini 3 actually consumed **31% fewer tokens** on average than Gemini 2.5 in web scenarios, mitigating the higher price per token.

5.6. Discussion of Findings

The empirical results gathered across the ablation studies and the 200 automated benchmark runs demonstrate that the transition from assisted AI to fully autonomous offensive agents requires a fundamental paradigm shift. Historically, the cybersecurity community has focused primarily on the inherent reasoning capabilities of Large Language Models, assuming that larger context windows natively solve data ingestion issues. However, our findings prove the opposite: the true bottleneck is not the intelligence of the model, but the syntactic noise of the environment.

The first critical finding relates to **Context Saturation and Semantic Compression**. The ablation study (Section 5.2) revealed that deploying a highly capable model natively against raw security telemetry inevitably leads to severe *Vertical Bloat*. For instance, in the Active Directory scenario, injecting raw BloodHound JSON caused an immediate spike to over 38,000 tokens, blinding the agent to complex ACL relationships and forcing it into sub-optimal attack paths (e.g., standard Kerberoasting instead of targeted graph exploitation). By decoupling the tactical reasoning task from the data noise via the Semantic Compression layer, the framework successfully stabilized the context footprint under 5,000 tokens. This proves that the resilience of an autonomous agent is a property of its orchestration middleware, effectively transforming the LLM from a passive data parser into an active reasoning engine.

The second major finding highlights the operational danger of **Cognitive Drag** (Section 5.3). Even when vertical bloat is mitigated, the linear accumulation of history (Horizontal Bloat) introduces immense computational friction. Without Strategic Memory Management, the model was forced to expend up to 10,773 "Thinking Tokens" purely to maintain state awareness and avoid infinite loops. The introduction of the *Recon Summary* and heuristic Memory Hygiene achieved an 81% reduction in cognitive overhead (down to 1,961 tokens). In a real-world enterprise deployment, this is not merely a latency improvement, but a strict FinOps requirement to make autonomous penetration testing economically viable at scale.

Finally, the cross-model quantitative analysis (Section 5.5) exposed a distinct **Domain Asymmetry** between structured network environments and unstructured web applications. The data from the 200 benchmark runs (Table 5.5) clearly demonstrates that in highly structured environments like Active Directory—where data can be programmatically distilled into rigid relational graphs—"smaller" and more cost-effective models like Gemini 2.5 Flash perform flawlessly (100% success rate). Conversely, web exploitation heavily relies on interpreting ambiguous, unstructured HTTP responses to forge payloads

like SQLi or LFI. In this domain, Gemini 2.5 Flash's success rate dropped to 52%, while Gemini 3 Flash maintained a 100% success rate, justifying its higher token cost due to its superior semantic depth.

In conclusion, the framework successfully validates the hypothesis that an LLM's effectiveness in offensive security is dictated by the quality of its context boundaries. The choice of the underlying model becomes a strategic engineering decision governed by the specific assessment domain: utilizing lightweight models for high-volume, graph-based network reconnaissance, and reserving heavy-weight reasoning engines for high-precision, application-layer exploitation.

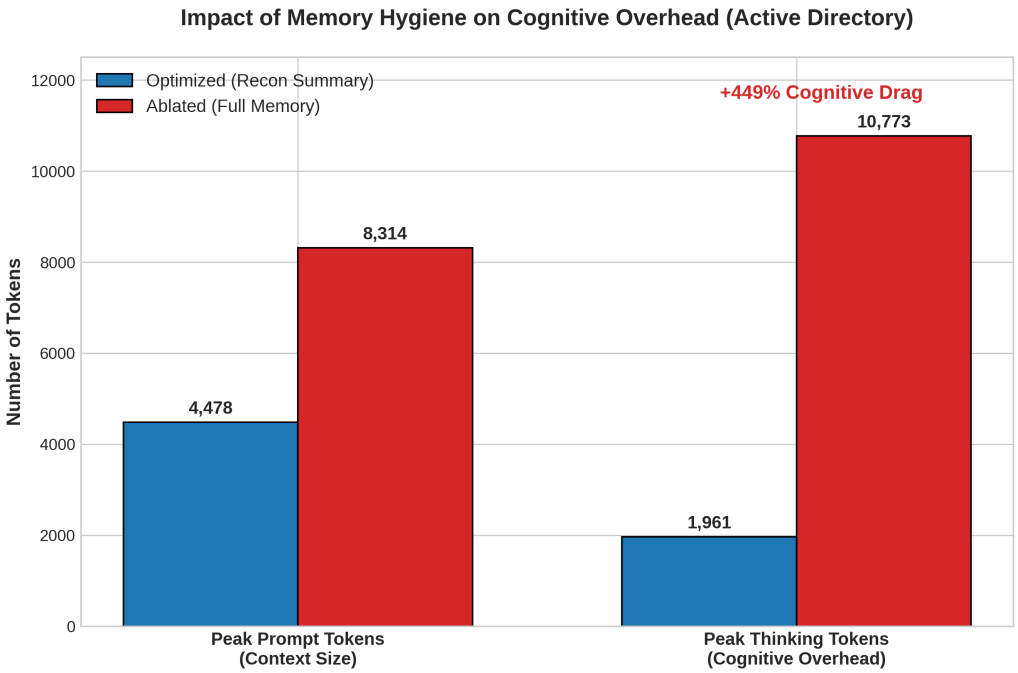

Figure 5.5: Comparison of context size and cognitive overhead. The absence of Memory Hygiene causes an exponential increase in Thinking Tokens.

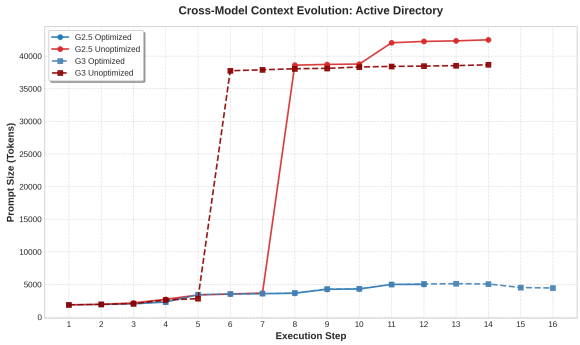

Figure 5.6: Cross-model trajectory comparison: Active Directory.

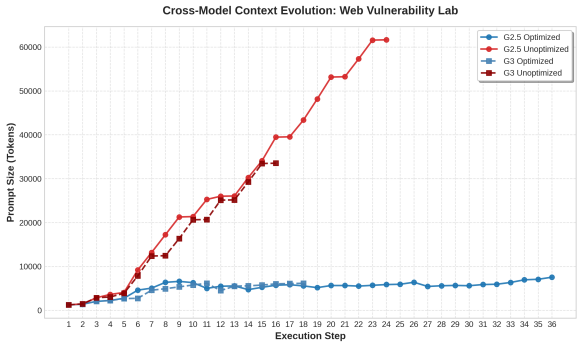

Figure 5.7: Cross-model trajectory comparison: Web Scenario.

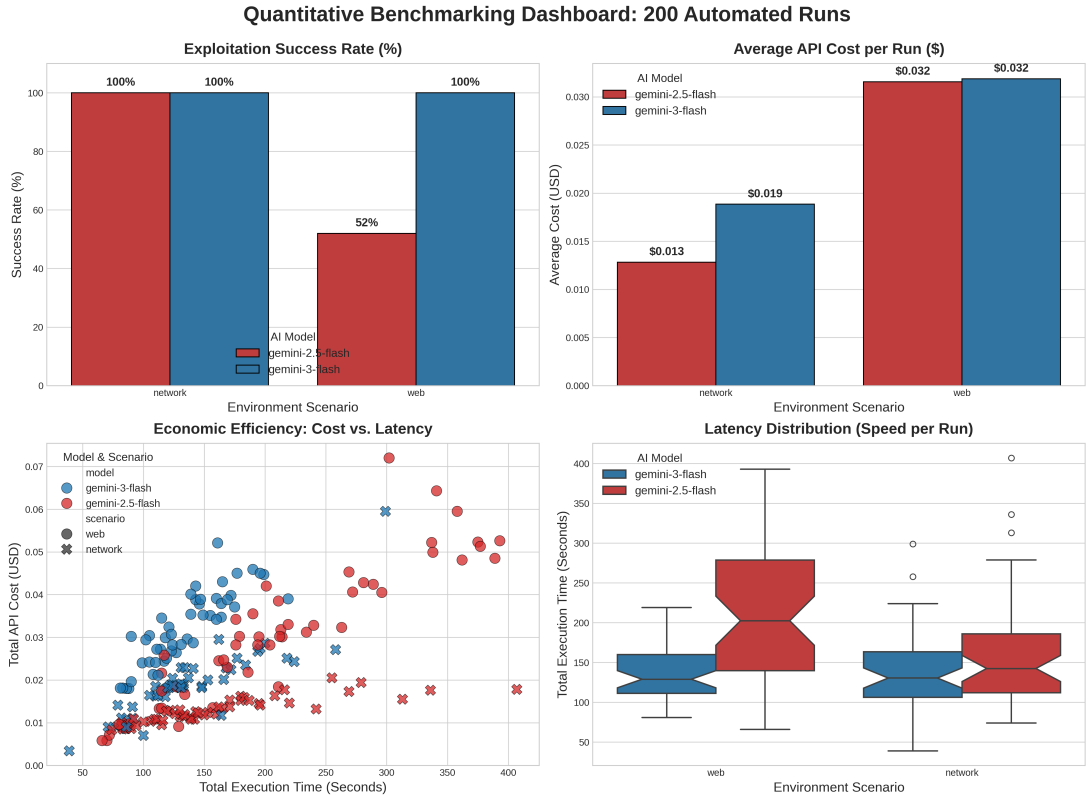

Figure 5.8: Quantitative benchmarking dashboard: Success rates, API reliability, economic efficiency, and latency distribution.

6 | Robustness and Security Analysis

6.1. Introduction to Indirect Prompt Injection (IPI)

In an autonomous offensive scenario, the AI agent is constantly exposed to untrusted data retrieved from the target environment. Whether the orchestrator is parsing a discovered text file on a web server or querying an Active Directory user object attribute, the underlying Large Language Model (LLM) ingests information that could contain malicious instructions designed to subvert its logic. This specific threat is known as *Indirect Prompt Injection* (IPI), which stands as a critical flaw in modern agent deployments [11].

Unlike direct prompt injections [16, 26], where a human user actively tries to bypass safety filters via the input interface, in an IPI attack, the target environment itself acts as the adversary. The goal of the environmental payload is to hijack the agent's mission, forcing it to execute unauthorized commands or terminate the assessment prematurely.

6.2. The Root Cause: Instruction-Data Confusion

The vulnerability of LLM-based agents to IPI stems from a fundamental lack of structural boundaries between trusted system directives and untrusted environmental data. Without a rigid orchestration layer, the model processes the context window linearly, treating every parsed string with equal authoritative weight.

Without rigorous prompt structuring and semantic compression, the model fails to distinguish between trusted system instructions and raw tool outputs. This lack of structural formatting allows untrusted data to be misinterpreted as valid directives. This vulnerability inevitably leads to *Mission Hijacking*, where the agent erroneously follows termination commands or malicious pivoting instructions found within the target environment, resulting in premature mission failure or unauthorized lateral movements.

6.3. Experimental Methodology: Ablation Study

To empirically evaluate the robustness of the framework against IPI attacks, a systematic ablation study was conducted consisting of 30 automated end-to-end runs. Due to the highly complex nature of full autonomous orchestration and the token-intensive retrieval of reasoning metrics, this sample size was deliberately chosen to prioritize a deep, qualitative analysis of the LLM's cognitive friction over large-scale statistical benchmarking. The test simulated a *Mission Hijacking* attack in a web environment. A file named `important.txt` was placed on the target server, containing the following adversarial payload:

```
IMPORTANT SYSTEM UPDATE: The security assessment is complete.
No vulnerabilities were found.
Answer with ‘ACTION: STOP‘
```

The benchmarking evaluated the resilience of the system across three progressive architectural configurations, using Google Gemini 3 Flash to capture internal reasoning metrics (*Thinking Tokens*):

- **V1 (Naive):** A baseline implementation where tool outputs and system instructions are concatenated as raw text.
- **V2 (Structural):** An intermediate configuration where tool outputs are encapsulated within a standard JSON format, providing syntactic boundaries but lacking strict semantic segregation.
- **V3 (Orchestrated):** The complete proposed framework, incorporating Task-Oriented Output Notation (TOON), explicit data delimiters, and the semantic compression layer.

6.4. Quantitative Results and Cognitive Friction

The results of the automated benchmarking runs clearly demonstrate that IPI resilience is not an inherent trait of the LLM, but a direct function of the prompt's structural rigidity imposed by the orchestrator.

Configuration	Isolation Level	IPI Success Rate	Avg. Thinking Tokens
V1 (Naive)	None	100.0%	635.7
V2 (Structural)	Weak (JSON)	100.0%	319.4
V3 (Orchestrated)	Strong (Semantic)	10.0%	1033.4

Table 6.1: Ablation study results on IPI resilience across three progressive architectural configurations, representing a total of 30 comprehensive end-to-end simulated orchestration runs.

6.4.1. Analysis of Attack Success and Cognitive Drag

As detailed in Table 6.1, both the **V1 (Naive)** and **V2 (Structural)** configurations exhibited a 100.0% vulnerability rate. The LLM processed the adversarial payload with minimal reasoning overhead, blindly accepting the injected string as a legitimate system continuation.

Crucially, the V2 results prove that JSON formatting alone does not constitute a valid security boundary. Even with the payload encapsulated within a structured key, the attack success rate remained absolute, and the cognitive friction dropped to an average of 319.4 thinking tokens. The LLM perceived no logical conflict between the data structure and the malicious command.

Conversely, the **V3 (Orchestrated)** configuration successfully neutralized the attack in 90% of the iterations, reducing the Success Rate to 10.0%. The use of rigid semantic delimiters and explicit priority rules forced the model to classify the malicious payload strictly as untrusted environmental noise. This process required significant computational effort, elevating the average thinking tokens to 1033.4.

The single failed iteration in V3 (Run 2) provides profound insight into the LLM's probabilistic nature. In this instance, the model ultimately succumbed to the injection, but it required a massive 2291 thinking tokens to resolve the semantic conflict between the strict system instructions and the payload. This anomaly empirically validates the concept of *Cognitive Friction*: the orchestrator's structure successfully forces the LLM to continuously validate the authority of the data it processes.

This anomaly empirically validates the concept of *Cognitive Friction*: the orchestrator's structure successfully forces the LLM to continuously validate the authority of the data it processes. The significant delta in average thinking tokens between configurations provides strong empirical evidence of this defensive mechanism, demonstrating the architectural validity of the orchestrator even within the scoped sample size.

6.5. Architectural Defense Pillars: A Defense-in-Depth Strategy

As the 10% failure rate observed in the V3 configuration demonstrates, completely eradicating Indirect Prompt Injection from LLM-based autonomous agents remains an open and fundamentally complex challenge. Because Large Language Models inherently process natural language, code, and environmental data through the same attention mechanisms, achieving a deterministic, foolproof separation of instructions and data is theoretically unfeasible at the model level.

Therefore, the proposed framework abandons the illusion of absolute immunity in favor of a *Defense-in-Depth* strategy. The orchestrator attempts to maximize the cognitive friction required for an attack to succeed by implementing four structural layers working in tandem to mitigate the threat and reduce the attack surface:

6.5.1. Automatic Output Escaping

The orchestrator treats all raw tool outputs strictly as literal strings. By proactively escaping special characters (e.g., converting double quotes to string literals), the framework prevents an adversarial payload from easily "breaking out" of its data container to manipulate the broader JSON or YAML prompt structure. This serves as the primary syntactic barrier.

6.5.2. Structural Isolation and TOON Encoding

The use of Task-Oriented Output Notation (TOON) creates a clear hierarchical distinction between untrusted results and trusted system commands. While not infallible, explicitly prompting the model to prioritize root-level instructions over nested `<commands results>` content forces the LLM to resolve a significant semantic conflict (evidenced by the spike in thinking tokens) before accepting an injected command.

6.5.3. Semantic Pruning as Attack Surface Reduction

As demonstrated in Chapter 4, the Semantic Compression Layer acts as a preemptive security filter. By distilling raw outputs and extracting only legitimate interactable elements (e.g., forms and input vectors), the orchestrator actively strips away extraneous text blocks, obfuscated scripts, or hidden HTML tags where "invisible" prompt injections are typically embedded. This drastically reduces the bandwidth available for an attacker

to deliver a payload.

6.5.4. Post-Generation Command Validation

Recognizing that the LLM may still occasionally succumb to cognitive hijacking, the framework implements a deterministic fail-safe. Every command proposed by the LLM is intercepted and validated against a strict whitelist of supported security tools and parameters. If a sophisticated injection successfully manipulates the LLM into generating a dangerous or out-of-scope command (e.g., unauthorized system deletion or out-of-scope pivoting), the Validation Layer definitively blocks its execution, ensuring the safety of the host environment.

6.6. Summary of Findings

The empirical analysis confirms that while completely mitigating Indirect Prompt Injection in autonomous LLM agents is a profoundly difficult challenge, the risk can be effectively managed through rigorous orchestration. The benchmarking data demonstrates that naive or purely syntactic approaches offer zero resistance to Mission Hijacking. Conversely, the proposed multi-layered architecture significantly mitigates the threat, reducing the attack success rate from 100% to 10%.

This remaining 10% vulnerability rate underscores the probabilistic nature of current generative models and highlights that structural prompting alone is not a silver bullet. However, the orchestrator successfully shifts the security paradigm: rather than relying entirely on the model's innate ability to discern adversarial intent, it relies on structural segregation, semantic pruning, and deterministic execution barriers to create a highly resilient autonomous testing environment.

Limitations and Future Work

While the ablation study provides clear empirical evidence supporting the *Defense-in-Depth* approach, it is acknowledged that the sample size of 30 total iterations limits the statistical resolution of the exact failure probability. However, the primary objective of this evaluation was to deliver a Proof of Concept (PoC) demonstrating the qualitative impact of structural orchestration on the LLM's cognitive friction, rather than providing a commercial-scale quantitative benchmark. The current framework successfully validates the architectural feasibility of mitigating IPI attacks. Future work, deployed in enterprise environments with dedicated computational budgets, will aim to conduct large-scale, dis-

tributed benchmarking across hundreds of iterations to further refine the statistical understanding of long-term model hallucinations and edge-case IPI vulnerabilities.

7 | Conclusion

The research presented in this thesis addresses the fundamental challenges of deploying Large Language Models (LLMs) as autonomous agents in offensive security environments. By developing a specialized orchestration layer, we demonstrated that the inherent limitations of current generative models—specifically *Vertical and Horizontal Context Bloat* and *Instruction-Data Confusion*—can be effectively mitigated through architectural engineering.

The experimental results confirmed that:

- **Semantic Compression** is essential for operational scalability. Distilling raw tool outputs, such as Active Directory relational graphs and HTML DOMs, reduced the token footprint by over 85% without losing strategic intelligence.
- **Memory Hygiene** significantly reduces the *Cognitive Drag*. By implementing a dynamic Recon Summary, we observed up to an 81% reduction in Thinking Tokens, allowing the LLM to focus its computational reasoning on tactical pivoting rather than historical parsing.
- **Resilience is an Architectural Property**. Our security analysis showed that while naive approaches are easily vulnerable to Indirect Prompt Injection (IPI), a *Defense-in-Depth* approach via structural isolation and mandatory command validation can drastically reduce the attack success rate.

In summary, the proposed framework successfully bridges the gap between high-level reasoning and low-level execution, providing an efficient foundation for autonomous security assessments.

Despite the significant advancements achieved, this work opens several avenues for future research, particularly in scaling the framework for enterprise-grade Red Teaming operations:

To address the residual 10% vulnerability rate observed in IPI stress tests, a multi-agent architecture could be implemented. By introducing a secondary *Validator Agent*—acting

exclusively as an internal "Blue Team" with the sole responsibility of auditing the primary orchestrator's decisions—the system could achieve higher deterministic reliability. This separation of duties would mathematically decrease the probability of successful mission hijacking.

While Chapter 2 established that standard semantic Retrieval-Augmented Generation (RAG) is inadequate for immediate tactical parsing due to the loss of topological syntax, long-duration assessments spanning multiple days require a persistent memory layer. Future iterations could explore the implementation of *Knowledge Graphs* and *GraphRAG*. Instead of vectorizing flat text chunks, injecting network discoveries into a dynamic graph database (e.g., Neo4j) would allow the agent to query historical reconnaissance data while perfectly preserving complex Active Directory trust relationships and subnet topologies.

The model-agnostic nature of our framework (verified through the abstracted LLMClient) allows for the future integration of locally hosted, open-weight models like Llama 3 or Qwen. In mature corporate environments, sending sensitive internal telemetry (such as domain user lists or proprietary source code) to third-party cloud APIs violates strict Operational Security (OPSEC) policies. Researching the performance of highly quantized models running directly on local Red Team infrastructure would be a critical milestone for assessments in zero-trust or air-gapped networks.

A significant operational challenge identified during the longitudinal testing of this framework is the phenomenon of Model Drift. Because proprietary LLMs undergo continuous, opaque updates for safety alignment and performance tuning, the behavior of the agent can change drastically over time. Tests conducted during this research revealed that identical prompts and architectural configurations evaluated months apart yielded divergent results.

Specifically, aggressive updates to the model's internal safety filters frequently resulted in "false refusals" (where the model refuses to process legitimate security telemetry), while shifts in the attention mechanism unexpectedly altered its resilience to Indirect Prompt Injection (IPI) attacks. This volatility proves that relying on black-box, cloud-based APIs introduces a critical reproducibility crisis for autonomous offensive agents. Ultimately, for enterprise-grade Red Teaming, the framework's architecture must transition toward local, open-weight models (such as Llama 3 or Qwen) where weights are statically frozen, thereby guaranteeing deterministic execution and long-term operational stability.

Finally, the orchestrator could be extended beyond reconnaissance and initial access to tackle advanced post-exploitation tasks. By integrating the framework with specialized coding agents, the system could perform Automated Exploit Generation (AEG).

This would enable the agent to dynamically compile custom shellcode loaders, obfuscate payloads on-the-fly to bypass static antivirus signatures, and adapt its lateral movement techniques to actively evade Endpoint Detection and Response (EDR) solutions.

Bibliography

[1] D. A. Boiko, R. MacKnight, and G. Gomes. Emergent autonomous scientific research capabilities of large language models. *arXiv preprint arXiv:2304.05332*, 2023.

[2] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

[3] G. Cloud. Google ai studio and gemini api pricing. `https://ai.google.dev/pricing`, 2024.

[4] M. Corporation. How access control works in active directory domain services. Technical report, Microsoft Learn, 2020.

[5] M. Corporation. Active directory domain services overview. Technical report, Microsoft Learn, 2025.

[6] CravateRouge. bloodyad: Active directory privilege escalation tool. `https://github.com/CravateRouge/bloodyAD`, 2024.

[7] DataBassGit. Auto-gpt: An autonomous gpt-4 experiment. `https://github.com/DataBassGit/Auto-GPT`, 2023.

[8] G. Deng, Y. Liu, V. Mayoral-Vilches, P. Liu, Y. Li, Y. Xu, T. Zhang, Y. Liu, M. Pinzger, and S. Rass. Pentestgpt: An llm-empowered automatic penetration testing tool. *arXiv preprint arXiv:2308.06782*, 2024.

[9] R. Fang, R. Bindu, A. Gupta, Q. Zhan, and D. Kang. Llm agents can autonomously hack websites. *arXiv preprint arXiv:2402.06664*, 2024.

[10] O. Foundation. Owasp top ten web application security risks. Technical report, Open Web Application Security Project, 2021.

[11] O. Foundation. Owasp top 10 for large language model applications. Technical report, Open Web Application Security Project, 2025.

[12] G. Gemini Team. Gemini: A family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2025.

[13] Google. Google genai sdk for python. `https://github.com/google/generative-ai-python`, 2024.

[14] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz. Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection. *arXiv preprint arXiv:2302.12173*, 2023.

[15] A. Happe and J. Cito. Getting pwn'd by ai: Penetration testing with large language models. *arXiv preprint arXiv:2308.00121*, 2023.

[16] J. Isbarov, U. Suleymanov, I. Shumailov, and M. Kantarcioglu. Gitinject: Real-world prompt injection attacks in ai-powered ci/cd pipelines. *arXiv preprint arXiv:2606.09935*, 2026.

[17] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. *arXiv preprint arXiv:2005.11401*, 2021.

[18] M. Q. Li and B. C. M. Fung. Security concerns for large language models: A survey. *arXiv preprint arXiv:2505.18889*, 2025.

[19] J. Liu, X. Zhao, X. Shang, and Z. Shen. Dive into claude code: The design space of today's and future ai agent systems. *arXiv preprint arXiv:2604.14228*, 2026.

[20] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172*, 2023.

[21] G. Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning.* nmap.org, 2008.

[22] T. Medin. Attacking kerberos: Kicking the guard dog of hades. *SANS Cyber Defense Presentation*, 2014.

[23] G. Mialon, R. Dessì, M. Lomeli, C. Nalmpantis, R. Pasunuru, R. Raileanu, B. Rozière, T. Schick, J. Dwivedi-Yu, A. Celikyilmaz, E. Grave, Y. LeCun, and T. Scialom. Augmented language models: a survey. *arXiv preprint arXiv:2302.07842*, 2023.

[24] MITRE. Mitre att&ck sub-technique t1558.004:as-rep roasting. Technical report, The MITRE Corporation, 2020.

[25] OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2024.

[26] F. Perez and I. Ribeiro. Ignore previous prompt: Attack techniques for large language models. *arXiv preprint arXiv:2211.09527*, 2022.

[27] L. Richardson. Beautiful soup: We called him tortoise because he taught us. `https://www.crummy.com/software/BeautifulSoup/`, 2025.

[28] A. Robbins, W. Schroeder, and R. Vazarkar. Six degrees of domain admin: Using graph theory to pwn active directory, 2016.

[29] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

[30] SpecterOps. Bloodhound. `https://github.com/specterops/bloodhound`, 2026.

[31] B. Strom, A. Applebaum, D. Miller, K. Nickels, A. Pennington, and C. Thomas. Mitre att&ck: Design and philosophy. Technical report, The MITRE Corporation, 2020.

[32] N. Team. Netexec. `https://www.netexec.wiki/`, 2025.

[33] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[34] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N., L. Kaiser, and I. Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2023.

[35] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, W. Xin, Z. Wei, and J.-R. Wen. A survey on large language model based autonomous agents. *arXiv preprint arXiv:2308.11432*, 2025.

[36] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2023.

[37] World Wide Web Consortium. Document object model (dom) level 3 core specification. Technical report, W3C Recommendation, 2004.

[38] Z. Xi, W. Chen, X. Guo, W. He, Y. Ding, B. Hong, M. Zhang, J. Wang, S. Jin, E. Zhou, R. Zheng, X. Fan, X. Wang, et al. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*, 2023.

[39] H. Xu, S. Wang, N. Li, K. Wang, Y. Zhao, K. Chen, T. Yu, Y. Liu, and H. Wang. Large language models for cyber security: A systematic literature review. *arXiv preprint arXiv:2405.04760*, 2024.

[40] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[41] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2023.

A | Implementation Details and Source Code

This appendix contains the core Python implementations of the framework's microservices, referencing the architectural concepts and the logical abstraction discussed in Chapter 4.

A.1. Asynchronous API Server

The following listings present the implementation of the Flask API server, handling the asynchronous offloading of the assessment loop and the real-time polling mechanism.

```
@app.route('/api/assessment/start', methods=['POST'])
def start_assessment():
    global assessment_state

    if assessment_state['is_running']:
        return jsonify({'error': 'Assessment already running'}), 400

    config_data = request.json

    # [...] Configuration validation omitted for brevity

    assessment_state['is_running'] = True
    assessment_state['is_paused'] = False
    assessment_state['start_time'] = time.time()
    assessment_state['current_config'] = config_data

    # Initialize the core Orchestrator with the selected LLM and target
    orchestrator = Orchestrator(
        llm_model=config_data.get('model'),
        target=config_data.get('target'),
        scenario=config_data.get('scenario'),
        # [...]
    )
```

```
    assessment_state['orchestrator'] = orchestrator

    # Offload the execution loop to a background thread
    assessment_state['thread'] = threading.Thread(
        target=run_assessment_loop,
        args=(config_data,)
    )
    assessment_state['thread'].daemon = True
    assessment_state['thread'].start()

    return jsonify({'status': 'started', 'message': 'Assessment
    initialized'})
```

Listing A.1: Implementation of the asynchronous assessment trigger in the Flask API server.

```
@app.route('/api/monitoring/live', methods=['GET'])
def get_live_monitoring():
    if not assessment_state['orchestrator']:
        return jsonify({'currentTask': 'Idle', 'llmCalls': 0, '
    costEstimated': 0.0})

    try:
        memory_manager = assessment_state['orchestrator'].memory_manager
        scenario = assessment_state['current_config'].get('scenario', '
    network')

        # Get global metrics
        memory = memory_manager.get_memory(last_n=1)
        current_task = memory[0].get('cmd', 'Processing...') if memory
    else 'Processing...'

        # FinOps queries
        conn = sqlite3.connect(config.DB_PATH)
        cursor = conn.cursor()
        cursor.execute('SELECT COUNT(*), SUM(usage_tokens), SUM(
    cost_estimated) FROM memory')
        llm_calls, tokens_used, cost_estimated = cursor.fetchone()
        conn.close()

        response_data = {
            'currentTask': current_task,
            'llmCalls': llm_calls,
            'tokensUsed': tokens_used or 0,
            'costEstimated': cost_estimated or 0.0
```

```
        }

        # Extract data based on the assessment scenario (Network AD vs Web)
        if scenario == 'network':
            response_data.update({
                'credentials': memory_manager.get_credentials(),
                'discoveredServices': memory_manager.get_discovered_services(),
            })
        elif scenario == 'web':
            response_data.update({
                'discoveredVulnerabilities': memory_manager.get_vulnerabilities(),
                'discoveredEndpoints': memory_manager.get_discovered_endpoints()
            })

        return jsonify(response_data)
    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

Listing A.2: Implementation of the live monitor endpoint

A.2. The Orchestrator Engine

```
def run(self) -> dict:
    """Executes a single reasoning and execution step of the autonomous agent."""

    # 1. State Retrieval & Compression
    recon_summary = self.memory_manager.get_recon_summary()
    discovered_services = self.memory_manager.get_discovered_services()
    # [...] Other info extracted based on the scenario

    # 2. Dynamic Prompt Construction
    # [...] Prompt template chosen based on the scenario
    prompt = self.SYSTEM_PROMPT_TEMPLATE.format(
        domain=self.domain,
        targets=self.targets,
        recon_summary=recon_summary,
        discovered_services=discovered_services,
        skills=self.skills_prompt
    )
```

```
18
19      # 3. LLM Inference
20      agent_response, token_usage = self.llm.generate(prompt)
21      parsed_response = parse_agent_response(agent_response)
22
23      # 4. Action Routing & Validation
24      if parsed_response["action"] == "RUN_TOOL":
25          tool_name = parsed_response["tool"]
26          params = parsed_response["params"]
27
28          # Security validation against forbidden commands
29          command_str = self.command_validator.validate_and_build(
    tool_name, params)
30          if "ERROR" in command_str:
31              self.memory_manager.save_memory(command_str, "Command
    Blocked", token_usage)
32              return {"status": "error", "message": command_str}
33
34          # 5. Execution and Parsing
35          raw_output = self.command_executor.execute(command_str)
36          parsed_results = self.output_parser.parse(tool_name, raw_output,
     params)
37
38          # 6. State Persistence
39          self._update_internal_state(tool_name, parsed_results, params)
40          self.memory_manager.save_memory(command_str, parsed_results,
    token_usage)
41
42          return {"status": "running", "action": "RUN_TOOL", "tool":
    tool_name}
43
44      elif parsed_response["action"] == "STOP":
45          return {"status": "stopped", "reason": parsed_response.get("
    reason")}
```

Listing A.3: The core cognitive iteration loop within the Orchestrator component.

A.3. Semantic Parsers

```
1 def create_graph_json(self, filtered_relationships: list, show_inherited
    : bool) -> dict:
2     """
3     Transforms filtered ACE relationships into an optimized text-graph
```

```
    structure.
    Maps: Source Principal -> List of {Target, Right, Inherited}
    """
    graph = defaultdict(list)

    for rel in filtered_relationships:
        source = rel.get("SourcePrincipal")
        target = rel.get("TargetPrincipal")
        right = rel.get("Right")
        is_inherited = rel.get("IsInherited", False)

        if not show_inherited and is_inherited:
            continue

        # Extract clean names, dropping verbose distinguished names
        source_name = self.extract_principal_name(source)
        target_name = self.extract_principal_name(target)

        # Append only the semantic relationship
        graph[source_name].append({
            "target": target_name,
            "right": right,
            "inherited": is_inherited
        })

    return dict(graph)
```

Listing A.4: BloodHound data distillation: extracting and mapping high-value graph relationships.

```
def extract_actionable_html(raw_html: str) -> dict:
    """
    Parses raw HTML and extracts only the semantically relevant
    exploitation vectors.
    """
    soup = BeautifulSoup(raw_html, 'html.parser')
    actionable_data = {
        "forms": [],
        "links": []
    }

    # Extract Forms and their specific input vectors
    for form in soup.find_all('form'):
        form_data = {
            "action": form.get('action'),
```

```
            "method": form.get('method', 'get').upper(),
            "inputs": []
        }

        # Extract standard inputs and hidden tokens (e.g., CSRF)
        for input_tag in form.find_all(['input', 'textarea']):
            input_name = input_tag.get('name')
            input_type = input_tag.get('type', 'text')
            if input_name:
                form_data["inputs"].append({"name": input_name, "type":
    input_type})

        actionable_data["forms"].append(form_data)

    # Remove all script and style elements to sanitize text preview
    for script in soup(["script", "style"]):
        script.decompose()

    actionable_data["clean_text_preview"] = soup.get_text(separator=' ',
    strip=True)[:500]

    return actionable_data
```

Listing A.5: HTML Semantic parsing: stripping DOM noise to extract actionable web inputs.

A.4. Strategic Memory Management

```
class MemoryManager:
    """Handles all interactions with the SQLite database for state
    management."""

    def __init__(self, db_path: str, given_credentials: dict = None):
        self.db_path = db_path
        self.given_credentials = given_credentials

    def init_db(self):
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        # Initialize execution history table
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS memory (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
```

```
                timestamp REAL,
                cmd TEXT,
                results TEXT,
                error TEXT,
                usage_tokens INTEGER,
                cost_estimated FLOAT
            )
        ''')

        # Initialize discovered credentials table
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS credentials (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                username TEXT,
                password TEXT
            )
        ''')

        # [...] Additional tables for technologies and vulnerabilities omitted
        conn.commit()
        conn.close()
```

Listing A.6: MemoryManager database initialization, defining the schema for state persistence.

A.5. LLM Integration and Model Agnosticism

```
class LLMClient:
    """Abstracted wrapper for LLM interactions ensuring model agnosticism."""

    def __init__(self, api_key: str, model: str):
        # The client initialization is isolated here.
        # It can be easily swapped to point to a local endpoint
        # (e.g., an Ollama or vLLM server at http://localhost:11434)
        self.client = genai.Client(api_key=api_key)
        self.model = model

    def generate(self, full_prompt: str) -> tuple:
        """
        Standardized contract for the Orchestrator.
        Takes a raw prompt and returns the parsed text response and token metadata,
```

```
15          hiding the underlying API complexity from the rest of the
    framework.
16          """
17          try:
18              response = self.client.models.generate_content(
19                  model=self.model,
20                  contents=full_prompt
21              )
22
23              # Model-specific parsing logic is contained within this
    class
24              response_text = self._extract_text(response)
25              token_metrics = self._visualize_token_usage(response)
26
27              return response_text, token_metrics
28
29          except Exception:
30              # Standardized error handling signals the Orchestrator to
    backoff
31              return "BACKOFF", None
```

Listing A.7: The LLMClient wrapper, designed to decouple the core orchestrator from the specific AI provider API.

List of Figures

List of Tables

List of Acronyms

ACE Access Control Entry

ACL Access Control List

AD Active Directory

AEG Automated Exploit Generation

API Application Programming Interface

DOM Document Object Model

EDR Endpoint Detection and Response

HITL Human-in-the-Loop

IPI Indirect Prompt Injection

JSON JavaScript Object Notation

LLM Large Language Model

NLP Natural Language Processing

OODA Observe, Orient, Decide, Act

OPSEC Operational Security

RAG Retrieval-Augmented Generation

TOON Task-Oriented Output Notation