POLITECNICO DI TORINO

Master's Degree in Cybersecurity

Master's Degree Thesis

Towards Autonomous Cyber Deception: An AI Agent for Dynamic Honeynet Management

Advisors
Prof. Danilo Giordano
Prof. Idilio Drago
Prof. Marco Mellia
Prof. Matteo Boffa

Candidate
Federico Mirra

December 2025

Abstract

Honeypots are deception systems used to emulate vulnerable services and collect threat intelligence. In constrained environments, where network or computational resources limit the number of deployable honeypots, security experts typically decide which assets to expose statically or semi-automatically. Attackers' tactics change quickly, and existing rule-based deception systems cannot cope with this dynamism, reducing their ability to capture valuable information about adversarial behavior.

Dynamic deception architectures, if properly tuned, can autonomously collect high-value threat intelligence without constant human supervision. This thesis addresses the lack of adaptivity and autonomy in current honeypot systems, investigating whether an AI-driven agentic architecture can autonomously manage honeypot exposure to maximize information gain from attackers. At the core of the proposed architecture, an AI agent repeatedly processes Intrusion Detection System (IDS) alerts, network configuration state and previous analyses. It infers the evolving attack steps, identifies compromised hosts and exploited services, and predicts the attacker's likely targets. Based on the attacker's progress, the agent autonomously adjusts the environment, shaping the attack surface to sustain engagement and extract intelligence.

To systematically measure the agent's ability to manage this environment, I developed a simulator that iteratively launches attacks against a network of vulnerable containers, followed by agent reasoning and deployment of defensive strategies. I reproduced attackers' behavior using Proofs of Concept (PoCs) exploiting known CVEs, creating traffic patterns that simulate real intrusion attempts.

Results in a simulated environment show that the AI agent achieved up to 96% accuracy in attack graph inference and 100% Exposure Efficiency, a custom metric to quantify minimized exposure. The results demonstrate the agent's ability to efficiently guide exposure and exploitation dynamics while ensuring optimal management of resources. The architecture has been deployed in a real honeynet environment hosting 15 web services. Initial data collection is ongoing to quantify the benefits of the solution when compared to static systems.

Contents

1 Introduction 4
- 1.1 Context and Motivation . . . 4
- 1.2 Methodology . . . 5
- 1.3 Thesis Organization . . . 5

2 Background 7
- 2.1 Honeypot Technologies and Architectures . . . 7
  - 2.1.1 Type of Honeypots and Core Components . . . 7
  - 2.1.2 Honeypots in Docker Containers . . . 8
- 2.2 Intrusion Detection System (IDS) . . . 9
- 2.3 Intrusion Detection Systems: Suricata . . . 10
- 2.4 MITRE ATT&CK Framework . . . 11
- 2.5 Generative Artificial Intelligence . . . 11
  - 2.5.1 Large Language Models . . . 11
- 2.6 AI Agents: Definitions and Frameworks . . . 14

3 Related Work 19
- 3.1 AI in Cybersecurity . . . 19
- 3.2 Prior Work on AI-Driven Honeypots . . . 20
- 3.3 Generative Honeypots . . . 21
- 3.4 Dynamic Honeypot Exposure . . . 21
- 3.5 Positioning of the Work . . . 22

4 Methodology 25
- 4.1 Problem Statement and Research Questions . . . 25
- 4.2 Multi-Agent System Architecture . . . 28
  - 4.2.1 Shared State . . . 31
  - 4.2.2 Episodic Memory . . . 31
  - 4.2.3 Network Gathering Node . . . 32
  - 4.2.4 Attack Inference Node . . . 33
  - 4.2.5 Exposure Manager Node . . . 35
  - 4.2.6 Firewall Manager Node . . . 36
  - 4.2.7 Persistence Node . . . 38

5 Agent Testing Environment 39
5.1 Attacker Behavior and Graph Modeling . . . 40
5.2 Discrete Event Simulation Environment . . . 42
5.3 Vulnerable Containers Deployed . . . 44

6 Results 49
6.1 Evaluation Parameters . . . 49
6.2 Metrics Definition . . . 50
6.2.1 Graph Inference Accuracy . . . 51
6.2.2 Exposure Efficiency Metrics . . . 51
6.3 Results Analysis . . . 53
6.3.1 Overall Results Analysis . . . 53
6.3.2 GPT-4.1 Results: Best Model . . . 56
6.4 Costs Analysis . . . 60
6.4.1 Ongoing Real-World Validation . . . 61

7 Conclusions, Limitations and Future Works 63
7.1 Conclusions . . . 63
7.2 Limitations . . . 63
7.3 Future Work . . . 64

A Network Gathering Node 65

B Attack Inference Node 67

C Exposure Manager Node 71

D Firewall Manager Node 73

E Persistence Node 75

F Attack Graph Inference Prompt 77

G Exposure Manager Prompt 83

H Firewall Manager Prompt 87

Bibliography 89

Chapter 1

Introduction

1.1 Context and Motivation

In the rapidly evolving landscape of cybersecurity, defenders face a persistent challenge: attackers continuously adapt their tactics, discover new vulnerabilities, and exploit weaknesses faster than defenses can evolve. Honeypots are designed to emulate vulnerable services and collect threat intelligence, playing a crucial role in understanding adversarial behavior. However, decisions about which assets to expose are often made statically or through simple rule-based mechanisms. Such fixed exposure policies may fail to leverage the valuable information generated during ongoing attacks, reducing the system's ability to capture meaningful intelligence about attacker behavior. The importance of this problem lies in the need to maximize the information gain from limited defensive resources. A dynamic environment, in which asset exposure changes dynamically, may help in sustaining attackers' interest [45]. Adaptive deception systems capable of detecting the progression of attacks and autonomously reshaping the exposed surface could enable defenders to collect richer data while minimizing operational overhead.
However, designing such adaptive systems is not straightforward. Naive or rule-based approaches typically fail because they cannot infer attack steps or take actions based on incomplete and noisy evidence. Previous research has explored adaptive and AI assisted honeypots. Architectures such as those by Kareem et al. [1] dynamically adjust honeypot configurations based on Machine Learning guided adaptation, but do not perform behavioral or attack steps inference. Systems such as HoneyGPT by Wang et al. [54], which rely on LLMs to simulate service interactions, focus on interaction realism rather than network management and behavioral analysis. As a result, they remain limited in their ability to autonomously reason about intent of attacker or reshape the attack surface during live engagements.
This thesis addresses the limitations cited by introducing an AI-driven adaptive cyber deception architecture designed to autonomously manage honeypot exposure in constrained environment based on the detected intent of the attackers.
The core idea is to enable the honeynet environment to observe, infer, and act periodically based on the understanding of traffic. The agent takes in input intrusion detection system (IDS) alerts, network configuration data, and prior analyses to infer an evolving

attack graph that maps attacker steps and exploited services. Based on the inferred information, it autonomously adjusts service exposure, thereby shaping the attack surface to sustain engagement and extract valuable intelligence.

1.2 Methodology

To evaluate the architecture proposed, I developed a simulator in which scripted attacks exploiting known CVEs target a set of vulnerable containerized services. Both the agent's reasoning process and the design of the attack simulation rely on the assumption that attacker behavior can be decomposed into sequential stages aligned with the `MITRE ATT&CK` framework, enabling structured inference of intent and progression. Each iteration alternates between attacker activity and the agent's execution, allowing measurement of its ability to infer attack progression steps and apply appropriate asset exposure. Experiments showed that the agent was able to correctly reconstruct the attacker's path in most cases, while also keeping unnecessary system exposure to a minimum, achieving full efficiency by my evaluation metric. Lastly, The architecture has been deployed in a real honeynet environment hosting 15 web services. Initial data collection is ongoing to quantify the benefits of the solution when compared to static systems.

1.3 Thesis Organization

This thesis is structured in seven chapters:

- **Chapter 2 – Background** introduces the theoretical foundations of the work. It reviews existing honeypot technologies, intrusion detection systems, generative artificial intelligence, and AI–agent frameworks, establishing the background necessary to understand the system's design.
- **Chapter 3 – Related Work** surveys related research, outlining how AI has been applied in cybersecurity and examining prior efforts toward adaptive or autonomously managed honeypots.
- **Chapter 4 – Methodology** details the proposed methodology. It presents the overall multi agent system architecture and explains the function of each system component
- **Chapter 5 – Agent Testing Environment** describes the testing environment created to evaluate the system.
- **Chapter 6 – Results** reports the experimental results. It defines the evaluation metrics, analyzes the system's performance under different attack modes, and discusses broader implications, including cost considerations.
- **Chapter 7 – Conclusions** summarizes the findings and discusses potential directions for future research.

Chapter 2

Background

2.1 Honeypot Technologies and Architectures

A **honeypot** is a deliberately open system or asset that imitates valuable applications, services, or data for the aim of adversaries deception. It is not designed to supply end users with useful functionality but to induce security intelligence from unauthorized activity. In defense-in-depth deployments, honeypots complement firewalls, endpoint defenses, threat intelligence feeds, and intrusion detection and prevention systems (IDS/IPS) by providing early indicators of compromise and by redirecting threats away from production resources. A honeypot is thus both decoy and sensor, by definition: an area where attacks are executed under observed scrutiny, and a mechanism to reduce pressure on operational systems by redirecting hostile activity elsewhere. Canonical definitions emphasize this decoy purpose: NIST specifies a honeypot as

> *"A system or system resource designed to be attractive to potential crackers and intruders, just as honey is attractive to bears."* [28]

Vendor documentation emphasizes the placing of decoy severs alongside production to measure reactions and deflect attackers away from real targets. [15] Because any legitimate use of honeypots is, by definition, minimal or null, *any* traffic will tend to be suspicious. This property allows honeypots to be extremely sensitive sensors for early warning: one unwanted connection, credential probe, or command is reason enough for an alert to be pursued. In addition to detection, honeypots enable systematic threat research. Finally, by keeping attackers busy with realistic but fictitious targets, a honeypot introduces friction and misdirection. These activities align with highly cited explanatory sources in industry and academia that characterize honeypots as *lure and learn* mechanisms that deflect, delay, and expose adversaries. Honeypots might not be sufficient on their own, but they are absolutely vital when part of an overall cyber posture.

2.1.1 Type of Honeypots and Core Components

Honeypots can be classified based on the degree of interaction with the attackers or the deployment goal. Honeypots are typically characterized by the *level of interaction* offered and the *deployment goal*. **Low-Interaction** deployments mimic some protocol banners

or application interfaces and are small, easy to manage, and fairly secure, but they capture little behavior. **Medium-Interaction** systems imitate more realistic application logic without offering attackers full operating system control, striking a realistic balance between risk of threats and realism. **High-Interaction** honeypots introduce actual operating systems and vulnerable applications so that attackers can be allowed to launch actual exploits, set up tooling, and attempt persistence or lateral movement, yielding the most insight but needing rigorous isolation and observation to contain risk. The *purpose* of deployment also shapes design considerations. *Production* honeypots are placed near business systems and are generally low- to medium-interaction; reliability, ease of maintenance, and tight integration with monitoring and response procedures are among the factors considered in their design. *Research* honeypots tend to be high interaction and are found in controlled or academic environments where maximization of data collection and realism outweigh operational convenience. Combined together, use and level of interaction define an effective design space through which defenders can trade realism for security and operational cost. Under the hood, modern surveys analyze honeypots into two fundamental parts: the *decoy*, the deliberately revealed asset, and the *captor*, the monitoring and control mechanism that stealthily observes, captures, and steers the interaction. Fan et al. formalize this as a decoy-captor (D-C) taxonomy and distinguish organizational designs wherein these elements are either *tightly coupled* or *loosely coupled/cooperative*. Tight coupling simplifies deployment but limits scalability and modular evolution; loose coupling increases architectural complexity but permits larger more adaptive deception fabrics and shared analytics. This conceptualization continues to hold sway since it efficiently decouples the lure from the observatory, explaining how each can be developed without undermining the other [14].

2.1.2 Honeypots in Docker Containers

Virtualization and, in particular, containers have changed honeypot design towards quick, reproducible, and more secure deployments. Containers allow defenders to package real, intentionally exposed applications and their dependencies and run them with strong namespace and resource containment. The net effect is that teams can deploy high interaction targets with nearly full virtual machine realism but with lower overhead and faster lifecycle operation. Boundary isolation reduces the likelihood that compromise of the decoy poses danger to production infrastructure, if network segmentation and runtime controls are applied with discipline. Containerization thus is a marriage of cloud native design and honeypot architecture for modern times so that organizations can use deception technology at scale without diminishing operational effectiveness.

Aspect	Concise Summary
Definition	Decoy systems/resources that attract adversaries to detect, deflect, and study attacks; value arises from unauthorized use rather than serving end users [15, 28].
Interaction Levels	*Low:* lightweight emulation with limited insight; *Medium:* richer app behavior without full OS control; *High:* real OS/apps for maximal fidelity at higher risk/overhead.
Deployment Purpose	*Production:* operational networks; emphasis on reliability, integration, and safety. *Research:* controlled settings; emphasis on data richness and realism.
Core Elements (D–C)	*Decoy* (the bait) and *Captor* (the hidden monitoring/control). Organization may be *tightly coupled* for simplicity or *loosely coupled/-cooperative* for scalability and modularity [14].
Operational Benefits	Early, signal detection; TTPs and malware collection; attacker distraction/misdirection; intelligence for patching and hardening.
Containerized Honeypots	Cloud native packaging of vulnerable services enables reproducible, scalable, high- nteraction deployments with strong isolation and rapid rotation; suited to orchestration and SIEM/DFIR integration.

Table 2.1: Summary of honeypot architecture

2.2 Intrusion Detection System (IDS)

Intrusion Detection Systems can be categorized according to some architectural, methodological, or operational decisions. First, concerning **monitoring scope**, systems can be deployed to monitor network traffic in general, Network-based IDS or NIDS, or to monitor specific hosts, Host-based IDS or HIDS. NIDS are placed at network choke-points, gateways, or switches and examine packet flows, commonly involving deep packet inspection, protocol parsing, or application level metadata [24]. HIDS, instead, operate on individual hosts (servers, work stations, or end devices) and track internal system activity: system calls, log files, file integrity, process behavior. Since HIDS can monitor within the host space, e.g., behavior blind to network sensors, they provide deep visibility at the expense of having to be deployed on all assets, along with potentially greater per host overhead. Second, IDS can be classified by **detection method**. Signature based IDS are based on a database of attack patterns or known signatures; these are effective to identify threats for which there are signatures, with a relatively low incidence of false positives if the signatures are accurate, but are not able to identify new, zero day, or polymorphic attacks for which no signature has yet been created [42]. Anomaly based IDS create a model of "normal" behavior, possibly derived from network traffic volumes, protocol utilization, resource access, or user activity, and mark deviations as potentially malicious. These systems have the ability to detect novel, unpublicized attacks, and evolve with emerging threat patterns, but are susceptible to greater false positives, particularly where *"normal behavior"* is evolving or is ill defined [24]. Hybrid solutions meld these detection

methods, signature and anomaly, to counter the weakness of one with the strength of the other [6, 42]. A third categorization depends on the **type of response** enforced by the system. Classic or "passive" IDS produce alerts or logs when they detect suspicious or known bad activity, and it is up to human operators to follow up or remediate. "Active" systems or modes, usually referred to as Intrusion Prevention Systems or IPS, have the ability to block, contain, or otherwise immediately take automated action against threats. These can discard packets from the network, reset the connections, block IP addresses, or insert firewall rules in real time, based on the severity and confidence of the detection [44]. In brief, for any deployment, one must consider where detection is taking place, host or network, how detection is being performed, and what level of response is desired or tolerated. Each dimension brings trade-offs in visibility, false positive rates, maintenance effort, performance overhead, and risk.

Aspect	**Key Points**
Classification by Scope	Network-based IDS monitor network flows; Host-based IDS run on individual hosts tracking local system behavior.
Classification by Detection Method	Signature-based: known attack detection; Anomaly-based: deviation from baseline; Specification-based: define allowed behavior; Hybrid: combining techniques.
Classification by Response Type	Passive IDS generate alerts; Active modes (IPS) enable blocking or containment of threats in real time.

Table 2.2: IDS classification

2.3 Intrusion Detection Systems: Suricata

Suricata, created by the Open Information Security Foundation (OISF), is an example of many of the qualities desired in modern IDS/IPS systems [29]. Suricata was built from the ground up to support multi-threaded operation, allowing it to take advantage of multiple CPU cores at once. This allows Suricata to consume and inspect large amounts of traffic without suffering from decreased performance. It also includes a massive list of network and application layer protocols it supports and can automatically detect many protocols, parse them, log detailed transactions, and extract files traversing the network for later analysis. Its output formats are myriad and range from structured JSON logs to classic log formats, as well as SIEM, threat intelligence, and analytics pipeline support. Since there is an increase in traffic encryption, IoT, cloud networks, and protocol heterogeneity, Suricata's protocol detection flexibility, anomaly detection alongside signature matching, and scalability render it particularly suitable for production environments and research alike. Besides the functionalities mentioned above, current research on IDS includes deep learning and hybrid detection models. Deep learning approaches are being utilized in network traffic stream analysis, detection of unknown threats, and enhancing correlation and behavior modeling against evasive attack techniques. In this thesis, Suricata serves as the foundation IDS component. Its robust alerts and event logs supply the empirical basis for automated agent inference, allowing the system to monitor exploitation, deduce

attack evolution, and manage honeypot exposure independently. Suricata's open, modular architecture is particularly well adapted to research environments demanding close integration with agentic and autonomous defense systems.

2.4 MITRE ATT&CK Framework

The MITRE ATT&CK framework is a publicly accessible knowledge base developed by The MITRE Corporation that systematically categorises adversary behaviour in terms of tactics, "why" an adversary acts, techniques, "how" the adversary achieves a tactic, and techniques or procedures (specific implementations) across multiple technology domains. Originally launched in 2013 to document post-compromise adversary activity in enterprise networks, the framework provides a structured way to model adversarial intent, behaviour, and progression through different stages of an intrusion [49].
In this work, I adopt the assumption that an attacker's actions can be mapped to the phased progression of adversarial behaviour as described by the ATT&CK framework. This alignment enables the agent to infer intent and exploit path progression by mapping observed indicators to known tactic/technique combinations, thereby constructing an evolving attack graph in line with the framework's taxonomy.

2.5 Generative Artificial Intelligence

Artificial Intelligence (AI) is a fundamental change in the way technology interacts with society. AI is essentially computer systems replicating human learning, understanding, problem solving, decision making, creativity, and autonomy [19]. As its possibility has grown, AI has evolved from narrow automation to a collection of techniques that infuse industry, medicine, entertainment, communications, and government management. More than an individual technology, AI is a group of techniques that accept large amounts of data, find patterns, and learn from additional data so that systems can make decisions and get better over time (see Figure 2.1 [37]). These capabilities already drive innovations in autonomous cars, precision medicine, and smart infrastructure. In the midst of this changing landscape, **Generative AI** has become a clear and powerful field. Rather than simply classifying or predicting from inputs, generative models discover the statistical composition of data and *generate* new outputs, including text, images, sound, and video, that are indistinguishable from human created content [17, 18]

2.5.1 Large Language Models

Trained on massive corpora that frequently comprise billions of tokens, LLMs learn to encode grammar, semantics, and context, facilitating robust performance on translation, summarization, question answering, and creative writing [35]. Among the key architectural innovations is the *Transformer*, whose self attention mechanism captures long range dependencies free of recurrence or convolution, enabling parallelization across tokens and enhancing contextual fidelity significantly [50]. Since models cannot handle raw text, actual LLM pipelines start with tokenisation. It is the essential pre-processing step that

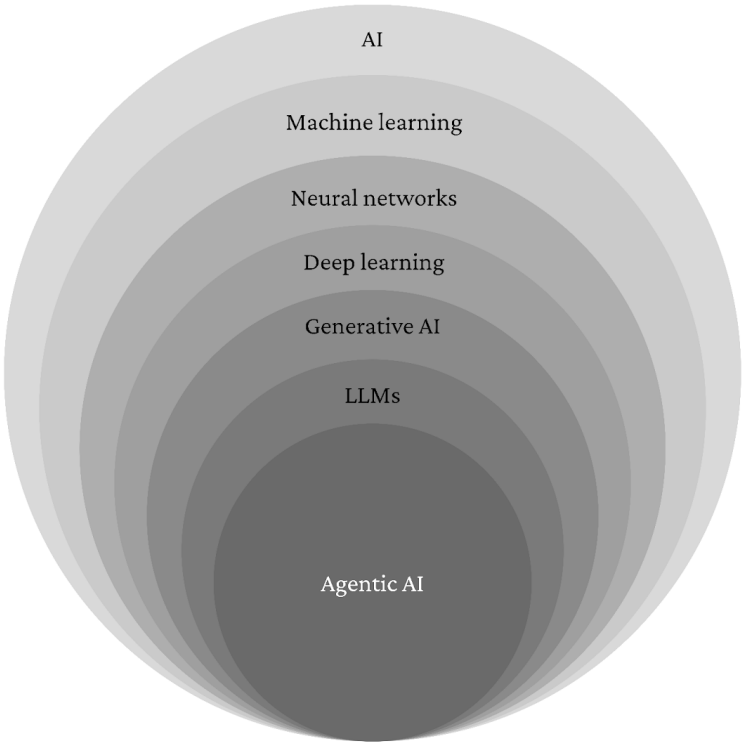

Figure 2.1: AI Fields Map

transforms raw text into a sequence of discrete symbols that the model can process. Since LLMs don't process strings directly, text needs to be broken up into units, or tokens, first. The selection of token inventory and segmentation rules has significant implications: they dictate how economically a model represents morphology, how well it generalizes to rare or misspelled words, how concisely it represents multilingual input, and ultimately how many tokens it takes to represent a passage of a given length. A number of paradigms exist for the definition of tokens, differing in the granularity with which text is segmented. Word level methods, prevalent in older systems, allocate a distinct token to each word type but are plagued by *out of vocabulary (OOV)* explosions and unmanageable vocabularies when faced with morphologically rich or multilingual environments. Character level models remove OOV at the cost of extremely long sequences and requiring large range dependencies to be learned across far more positions. Subword models are in between these two extremes and are the default for contemporary LLMs since they trade off vocabulary size and expressiveness. Tokenizer design has a direct influence on the effective capacity of an LLM since contemporary architectures take fixed length token windows as input. A model's *token window* is the maximum number of tokens it can attend over at a time; all inputs and intermediate prompts need to be expressible within this limit. Because self attention scales linearly with the number of tokens instead of characters

or words, a tokenizer that produces fewer tokens per passage effectively "stretches" the usable context for the same architectural limit [50]. This interplay between tokenisation and context window size creates real trade offs in system design and model behavior. Larger vocabularies will shorten average sequence length but at potential cost in training expense and memory load; smaller vocabularies are easier to model but inflate toke counts. Lastly, since prompts, system instructions, few shot examples, and retrieved passages all compete for the same limited context, token budgeting becomes a premier issue in applied LLM workflows. Recent years have witnessed heightened development that expands both modality and capability. OpenAI's GPT-5, released on August 7, 2025, is specialized in coding, agentic tasks, long context reasoning, and dynamic routing between fast and "deep thinking" behavior [33]. The GPT 4.1 series, released on April 14, 2025, built out coding and instruction following capability while accommodating very large context windows, on the order of hundreds of thousands to around a million tokens, depending on variant [32]. GPT-4o introduced real-time multi modal reasoning over text, vision, and audio, with additional efficiency improvements [30, 31]. Meta's Llama 4 series (Scout/Maverick) combined mixture-of-experts architectures with industry leading long context capability (Scout on the order of $\sim$10 million tokens; Maverick in the million token range) and multi-modal inputs [26, 27]. Anthropic's Claude Opus 4.1 specialized in longer reasoning and multi-file code refactoring, with strong reported scores on SWE-Bench Verified [4, 5]. Google's Gemini 2.4 families built native, long-context window, multi-modality for the Pro version, and tiered offerings to balance latency and ability [10, 11]. xAI's Grok 4 Heavy investigated parallel hypothesis in multi-agent inference for better complex planning and reasoning with competitive benchmark performance on code and STEM tasks [9, 57]. Together, these families are pushing toward unified, multi-modal systems with more agentic behavior and long-term memory. Despite progress in pretraining and alignment, model behavior remains extremely sensitive to how tasks are formulated. Prompt design has therefore been added as a regular procedure. Zero-shot and few-shot prompting leverage pretraining priors and small exemplars for output structure and style guidance [35]. Chain-of-thought prompting and similar techniques can draw out intermediate reasoning steps to enhance performance on symbolic or multi-step problems [55]. Retrieval-Augmented generation (RAG) also anchors responses in external sources, enhancing factuality and enabling provenance by incorporating retrieved evidence into the generation loop [21]. These approaches are complementary to alignment methods like instruction tuning in that they stabilize behavior, eliminate failure modes, and maintain models' general usefulness [35].

Model Family	Main Characteristics	Key Citations
OpenAI GPT-5	Announced in August 2025; emphasizes coding, agentic tasks, long-context reasoning, and dynamic routing between fast and deep-thinking behaviors.	[33]
OpenAI GPT-4.1	Released in April 2025; family includes GPT-4.1, 4.1 mini, and 4.1 nano; improved coding and instruction-following; supports very long context windows (up to ∼1M tokens).	[32]
OpenAI GPT-4o	Multimodal "omni" model; integrates text, vision, and audio; real-time reasoning; improved efficiency relative to earlier GPT-4 variants.	[30, 31]
Meta Llama 4 (Scout / Maverick)	Mixture-of-experts architecture; long context capacity (Scout up to ∼10M tokens, Maverick ∼1M tokens); multimodal inputs (text + images).	[26, 27]
Claude Opus 4.1	Released in 2025; excels at extended reasoning and multi-file code refactoring; achieves ∼74.5% on SWE-Bench Verified; strong agentic search.	[4, 5]
Google Gemini 2.5 (Pro / Flash / Flash-Lite)	Native multimodality (text, image, audio, video, PDFs); long context (Pro: ∼1M tokens input, ∼65k tokens output); includes built-in reasoning; variants optimized for latency or power.	[10, 11]
xAI Grok 4 Heavy	Employs multi-agent inference with parallel hypotheses for complex reasoning; strong benchmark results in code, STEM, and long-horizon planning; premium model with higher compute demands.	[9, 57]

Table 2.3: Representative LLM families (2024–2025) and their defining capabilities.

2.6 AI Agents: Definitions and Frameworks

The revival of interest in big language models (LLMs) has brought about a resurgence in the interest in autonomous agents, defined as computer programs that perceive the environment, act autonomously to accomplish tasks, and can adapt the performance through machine learning or the acquisition of knowledge [56]. Especially in modern architectures, an agent is endowed with an internal memory, decision logic, and the capacity to invoke external tools. But contemporary agent systems frequently embed a large language model (LLM) as their reasoning core, making the LLM the *"brain"* of the agent. An AI Agent can be understood as a mapping from percept sequences to actions, but enriched by internal state, planning, and deliberation. In practical systems, an agent continuously loops through a cycle: it perceives inputs, updates internal memory or context, reasons about which action to take next, executes that action, and then integrates feedback to revise

beliefs or plans. This cycle distinguishes the agent from a simple reactive system: it provides persistence of context and the ability to plan across multiple steps. When modern agents incorporate an LLM, the model plays the role of the reasoning engine. That is, instead of encoding rules of handcrafted heuristics, the LLM is used to interpret context, generate hypothesis, plan sub-goals, or synthesize tool calls. The synergy is powerful: the LLM handles language-based reasoning, abstraction, and flexible planning, while the agent infrastructure ensures robustness, state management, and safe tool execution. Surveys of LLM-based agent systems highlight this synergy as central to the current growth in agent research. For instance, Wang et al. describe how LLMs have recently enabled autonomous agents that combine perception, planning, and action in real-world settings, by leveraging the model's reasoning capabilities while embedding them into agent loops [53]. Correspondingly, the survey of methodology and applications of LLM agents insists that current agent systems are actually LLM-drive agents, wherein the LLM is queried over and over as part of some action-selection procedure and not as a pure static language oracle [25]. Such kind of architectures need to address various challenges. One needs to handle prompt engineering with great care, the limits of the memory context window, hallucinations or incorrect outputs from the LLM, and synchronization across multiple agents or components. Guaranteeing deterministic behavior, or at the very least, reproducible traces of reasoning require the control of randomness within the LLM, definition of structured interfaces. Additionally, the overhead of repeated invocation of the LLM can cause latency as well as cost. However, in spite of such challenges, the combination of agent scaffolding with LLM reasoning is the state-of-the-art in the construction of flexible, goal-directed, and explainable autonomous agents.

An LLM is not enough on its own to deploy an agent: one also needs to inject an orchestration level to control state, schedule the execution of reasoning steps, broker tool invocation, coordinate the execution of multiple agents if required, and facilitate introspection as well as explainability. There have been numerous frameworks developed over the past years to address these issues that present abstractions to facilitate easy construction of agentic systems. In the remainder of this section, the most pertinent frameworks are reviewed, followed by the argument to use LangGraph as the basis of the orchestration stack developed in this research. One of the most widely used frameworks is LangChain. It offers modular abstractions to chain together prompts, memory, retrieval modules, and tool calls. The core concept of the framework is the chain, where the outcome of one step serves as input to the next, thus supporting workflows such as retrieval-augmented generation, question answering, or multi-step reasoning [39, 40]. Since LangChain presents a standard interface to integrate models, tools, and vector stores, it became de facto baseline to build applications based on LLMs. However, its chain-like design automatically enforces a linear or sequential sequence of reasoning steps, which becomes restrictive when issues require branching, concurrent execution, feedback loops, or strongly coupled interaction between specialized agents. To go beyond these constraints, more recent frameworks allow for multi-agent interaction. An example is Autogen, that lets multiple agents be instanced that broker their interaction with conversational protocols until a goal is attained. Such a paradigm offers extensibility and semantic expressiveness, yet the control flow remains mostly implicit: the order of reasoning, the dependency between

them, as well as the branching semantics all cohabit in the dialogue between the agents. This implies that achieving determinism, tracing internal reasoning, or enforcing strict dependency between the agent outputs becomes challenging in rugged systems. LangGraph presents a graph based orchestration paradigm. The reasoning system within LangGraph is represented as a directed reasoning graph: the vertices represent discrete reasoning units, the edges define the data or control dependency between them. That is, the information flow is made explicit in the graph structure, as opposed to implicit within conversational protocols. This enables deterministic scheduling: each vertex is executed whenever its inputs are available, producing structured outputs that propagate downstream. LangGraph is also runnable with the components of LangChain but complements them with a state propagation mechanism along with graph execution semantics [38, 48]. Since node level dependencies are explicit, LangGraph is suitable for developing multi agent systems with rich interactions. Agents can be represented as being mapped to nodes, taking structured inputs from predecessors to producing structured outputs to successors. The overall state changes as the agents execute, recording both the system perception of the world as well as the order of decisions implemented. Additionally, LangGraph accommodates loops, conditional branches as well as reactive activation of the nodes, crucial in representing feedback driven reasoning with dynamic worlds. In applications such as cybersecurity, where perception, inference as well as control need to be highly integrated as well as traceable, the directed graph abstraction offers vital clarity as well as accountability. From a software engineering standpoint, the graph based model yields several benefits. Considering the thesis goals, the LangGraph paradigm aligns naturally. In this work, agents are specialized in perception, and control. Using LangGraph enables specification of this reasoning graph concretely, inference over agents' internal state updates, and logging of reasoning at each node. Broadly speaking, the trajectory from chain based frameworks and conversational agent systems toward graph based orchestrators such as LangGraph reflects an evolution in AI agent design: from loosely coordinated modules toward structured, deterministic, and explainable systems. In conclusion, LangGraph's directed reasoning graph approach offers a robust way to maintain autonomy, yet enforce traceability and accountability. It is thus both a practical and conceptually coherent choice for this thesis's agent orchestration layer.

	LangChain	**AutoGen**	**LangGraph**
Core paradigm	Sequential "chains" of reasoning and tool calls	Multi-agent conversational coordination between agents	Directed reasoning graph with explicit dependencies and state propagation
Agent coordination	Limited; mostly single agent chains or simple branching	Agents communicate via dialogues, exchange messages, collaborate	Agents correspond to graph nodes; dependencies explicit via edges
State management	Memory abstractions external to the chain logic	Agents may maintain local memory; global coordination less formal	Built in state propagation across nodes, global state evolves
Control / determinism	Execution order is linear and relatively predictable	Conversational order can inject nondeterminism in turn taking	Deterministic scheduling of nodes when inputs are ready
Branching / loops / dependencies	Possible but must be managed manually or via ad hoc constructs	Possible via conversation logic, but implicit	First class support for conditional branches, loops and reactive node activations
Explainability / traceability	Trace is chain of steps; internal reasoning can be opaque	Trace is inter agent dialogue; harder to reconstruct causal flow	Trace corresponds to graph execution path; internal states logged per node
Integration with LLMs / tools	Very mature and broad integrations with models, API, memory	Supports tool invocation, agent roles, human-in-loop	Built atop LangChain's tooling with graph semantics & statefulness
Ideal use cases	Simple to moderate pipelines, RAG, sequential workflows	Collaborative dialog systems, human-in-loop multi agent workflows	Complex workflows with dependencies, feedback loops, security critical orchestration
Challenges / tradeoffs	Less suited for complex branching or feedback loops	May face nondeterminism, harder to guarantee reproducibility	Learning curve; more architecture overhead — complexity in graph design

Table 2.4: Comparison of agent-framework paradigms: LangChain, AutoGen, and LangGraph

Chapter 3

Related Work

3.1 AI in Cybersecurity

AI is being applied throughout the entire defense lifecycle in cybersecurity practice, from threat detection, analysis, and response to risk management. In intrusion detection, for example, graph neural networks and deep learning have extended the ability of conventional network and host based systems through traffic anomaly modeling and modeling structural attributes of flows. This has lowered false positives and enhanced the detection of new attack behaviors [2, 59]. AI techniques are also leading malware analysis and detection. There, models carry out static and dynamic analysis of code sequence or binary, along with behavioral analysis of execution traces. Though accuracy has been enhanced, the introduction of explainable AI methods enables analysts to interpret decisions, thus augmenting accountability and trust in automatic detectors [7, 16]. Natural language models and machine learning in social engineering and phishing detection examine URLs, email content, and visual attributes to detect attempts to defraud. These techniques are now being incorporated into Security Operations Center (SOC) workflows, where they not only detect but also trigger responses automatically, such as proactive blocking or selective alerts to users [12, 41].

Another important use case is vulnerability discovery. Deep learning pipelines identify recurring patterns of vulnerability in source code or binaries, informing prioritization when patching is postponed. Such findings are essential to agentic exposure management, determining which services can be safely exposed as honeypots and which need defending [23, 60]. In addition to detection and analysis, AI improves Security Orchestration, Automation, and Response (SOAR). These platforms combine enrichment, correlation, and automated workflow, minimizing response time and analyst effort [20].

Within agentic contexts, SOAR offers a control plane, overseeing permissions and guaranteeing autonomous action—like rotating honeypots or modifying firewall rules—is within operational guardrails. At a more inferential level of inference, attack graph inference and risk modeling offer a structured understanding of multi step adversary campaigns. Bayesian attack graphs and graph learning methods provide probabilistic inference over evidence, in support of dynamic defenses that predict attacker action [3, 22, 58]. Lastly, advances in deep reinforcement learning (RL) have encouraged the investigation of

Dimension	Advantages	Challenges
Real-time operations	Agents act at machine speed, reducing attacker dwell time and enabling immediate responses.	Risk of unsafe or uncontrolled autonomous actions if governance constraints are not enforced.
Scalability	Orchestration coordinates heterogeneous assets, integrating multiple tools and defense layers.	Coordination failures can cause redundant, inconsistent, or conflicting actions among agents.
Adaptivity	Learning-based policies improve detection and response under evolving threats and concept drift.	Adaptivity complicates explainability; SOCs require transparency and traceable rationales for trust.
Governance	Orchestration layers enforce permissions, guardrails, and human-in-the-loop oversight.	Maintaining compliance with invariants requires rigorous validation and constant monitoring.

Table 3.1: Comparison of advantages and challenges in deploying multi-agent orchestration for cybersecurity.

fully autonomous cyber defense. RL and multi agent RL methods train defenders to act within adversarial environments, trading off adaptability with constraints to avert unsafe or conflicting actions [13, 36, 46]. There are a number of advantages to the use of multi agent systems and orchestration layers for cybersecurity. By supporting automation at machine speed, agents provide real time automation that drastically minimizes attacker dwell time. As they are scalable, they can coordinate behavior over heterogeneous assets and tools, and their adaptivity via learning means that they can continue to be effective under concept drift, enhancing resilience over static rule-based systems [20,36]. These advantages are accompanied by substantial challenges. Autonomous behavior needs to meet rigorous safety and governance criteria, adhering to guardrails and system invariants [13]. Secondly, there are coordination risks when agents' goals are partially misaligned, which causes redundant or even contradictory defense mechanisms. Lastly, explainability is an issue: in security operations with high stakes, analysts need insight into decision making. This need drives explainable AI overlays on malware and intrusion detection systems, along with thorough decision logs for orchestrators of autonomous actions [16].

3.2 Prior Work on AI-Driven Honeypots

Classic honeypots, although effective for threat intelligence gathering and adversarial engagement, have for some time now been limited in static configuration. Their inflexibility in changing behaviors or responses exposes them to fingerprinting, thus reducing their effectiveness against sophisticated adversaries. Conversely, the advent of artificial intelligence (AI) has dramatically improved honeypot systems in the sense of the potential for realism, adaptability, and deeper attacker interaction levels. Recent work shows how this AI and machine learning (ML) functionality has changed the design of honeypots from

static reactive traps to intelligent dynamic agents that have the capability to deceive, mimic behavior, and make context-dependent decisions.

3.3 Generative Honeypots

A key change has been the advent of large language model (LLM) enabled honeypots. Generative models, as extended to terminal or shell contexts, have demonstrated significant potential for generating realistic, contextually relevant, and adaptive system output. *HoneyGPT*, for example, presents an LLM-based terminal honeypot that employs sophisticated prompt engineering methods to sustain long-term deceptive interaction. Its field tests report significant gains over attacker engagement, in both duration and behavioral variety [54]. Also, the *LLM Honeypot* system trains language models on actual attack logs to produce realistic SSH activity, along with providing automated summarization to aid analyst workflows [34]. Rounding out these developments, *ShelLM* shows that LLM-augmented shell interfaces can effectively mimic UNIX like responses. Empirical user testing validates these systems' realism and fools human attackers at high rates of success [43]. Together, these endeavors point to a shift towards generative realism and static deception, wherein the conversational and interactive honeypot activity dynamically changes.

3.4 Dynamic Honeypot Exposure

Alongside these generative developments, multi agent and reinforcement learning systems have emerged as strategic frameworks for honeypot orchestration. Wang et al. propose AARF (Autonomous Attack Response Framework), which models attacker-defender interactions through coupled reinforcement learning and Hidden Markov Models [52]. By simulating multi step attack chains using DQN agents, AARF learns optimal deception policies that maximize engagement time and information capture. The comprehensive survey by Abdul Kareem et al. [1] provides a unifying architectural framework for AI-driven adaptive honeypots, emphasizing the transition from static traps to dynamic, context aware defense systems. This work aligns closely with integrated management approaches by highlighting requirements for real-time adaptation to emerging threat vectors. In blockchain-IoT environments, Commey et al. [8] introduce an architecture where nodes are dynamically transformed into honeypots on demand. Using Bayesian game theory and an AI-powered IDS integrated with smart contracts, their approach optimizes deployment strategies to maximize adversarial uncertainty while balancing resource constraints. Tang et al. [47] present a deep learning enabled firewall system that uses 1D-CNN for low level traffic classification to trigger TCP-level switching mechanisms. Their TCP_REPAIR based approach reroutes malicious flows to matching honeypots with minimal latency, dynamically balancing concealment and resource utilization by filtering only relevant attack sessions to decoy environments.

3.5 Positioning of the Work

The surveyed corpus demonstrates AI-driven innovations across traffic classification, agent simulation, and game theoretic deployment. However, these contributions remain functionally specialized, addressing isolated aspects such as traffic steering, policy optimization, or deployment strategy, without semantic integration of threat context. This thesis introduces a critical architectural distinction: an LLM-driven orchestration agent that operates at the semantic layer to unify defense components through contextual reasoning: Key Differentiators:

- **Beyond Statistical Classification** (vs. Tang et al. [47]): While 1D-CNN systems classify traffic using low level statistical features to trigger deterministic routing, the proposed LLM agent interprets IDS alerts semantically, translating textual threat indicators (TTPs) into proportional, context aware honeypot configurations that adapt to attack sophistication.

- **Semantic vs. Numerical Optimization** (vs. AARF [52]): AARF employs numerical policy optimization through DQN within predefined action spaces and state representations. The LLM agent performs semantic optimization—reasoning over natural language threat descriptions to generate adaptive deception policies without numerical constraints or fixed action spaces.

- **From Deployment to Configuration** (vs. Commey et al. [8]): Game theoretic models optimize where and when to deploy honeypots based on rational agent assumptions. The LLM agent determines what deception content to present—dynamically adjusting honeypot fidelity, vulnerability profiles, and service configurations based on interpreted attack sequences.

- Contextual Integration (vs. AI-Honeypot Surveys [1]): **General adaptive frameworks separate perception** (CNN-based classification), reasoning (RL based adaptation), and actuation into distinct modules. This architecture centralizes contextual analysis within the LLM, enabling it to jointly interpret IDS alerts, infer attack graph progression, and orchestrate firewall rules as a unified decision making agent.

This thesis unifies three synergistic components:

- Intrusion Detection System (IDS)-driven perception for situational awareness

- Attack graph inference for structure understanding of multi step exploitation

- Autonomous firewall orchestration for dynamic exposure of vulnerable containers

Combined, these modules optimize exploitation visibility under safety constraints, enhancing both attack graph inference accuracy and defensive decision quality. This integration represents a synthesis of adaptive deception, strategic reasoning through natural language understanding, and automated control—advancing AI-driven honeypots toward fully autonomous cyber defense systems.

Research Focus	**Key Method**	**Description**	**Representative Works**
Adaptive Honeypots	ML-based behavior analysis	Dynamic adjustment of services, banners, and responses based on attacker TTPs through behavior clustering and novelty detection	[1]
Generative LLM Honeypots	Large Language Models for interaction synthesis	Realistic shell and terminal interactions using LLMs to enhance engagement realism and believability	[54], [34], [43]
Multi-Agent RL-Based Deception	Reinforcement Learning + HMMs	Simulation of attacker-defender dynamics with sequential policy learning for optimal response strategies	[52]
Game-Theoretic Deployment	Bayesian game theory + blockchain	Strategic honeypot node transformation and placement optimization to maximize adversarial uncertainty	[8]
Deep Learning Traffic Classification	1D-CNN + TCP_REPAIR	Low-level statistical traffic analysis for rapid rerouting of malicious flows to matching honeypots	[47]
LLM-Driven Semantic Orchestration	**LLM agent for contextual reasoning**	**Semantic interpretation of IDS alerts to generate adaptive deception policies; unified orchestration of IDS perception, attack-graph inference, and firewall control**	**This Thesis**

Table 3.2: Summary of Prior Work on AI-Driven Honeypots and Positioning of This Research

Chapter 4

Methodology

This chapter presents the design of the proposed multi-agent architecture. The goal is to address a fundamental limitation in current deception systems: in resource constrained environments, defenders must decide which vulnerable assets to expose to maximize engagement with attackers. Traditional honeynet deployments rely on static exposure policies or rule based mechanisms, which fail to leverage the continuous stream of evidence generated during an attack. As a result, valuable intelligence may be lost when irrelevant services remain exposed or when active targets are concealed prematurely.

4.1 Problem Statement and Research Questions

In modern cyber operations, attackers progress through multi step intrusions, which typically encompass phases such as reconnaissance, exploitation, privilege escalation, and data exfiltration. When honeypots remain statically exposed throughout these phases, defenders risk to not efficiently engage adversaries, especially when only a limited number of services can be deployed simultaneously. They may fail to sufficiently engage sophisticated adversaries by sustaining exposure to irrelevant services. Moreover, the static exposure of honeypot make them easily recognizable as decoy systems, since they do not provide useful services and or data, causing a lost of interest from malicious actors. A dynamic environment is essential to sustain attacker interest [45], necessitating a system that can adapt based on real-time network observations.
To enable this adaptivity, the system architecture incorporates an Intrusion Detection System (IDS) as a critical pre-processing layer. Direct processing of raw network traffic by an LLM is unfeasible due to the constraint of finite context windows; the sheer volume of raw packet data (PCAP) would immediately saturate the model's input capacity and drive up latency. The IDS addresses this scalability bottleneck by distilling high-volume network traffic into discrete, semantically meaningful alerts. This architecture ensures that the agent receives the minimum amount of efficient data necessary to reconstruct context, transforming an insurmountable data processing challenge into a manageable reasoning task.
However, relying on IDS alerts introduces its own set of complexities. The design of an autonomous agent capable of managing honeynet exposure becomes a complex control

problem characterized by two main challenges:

1. **Partial Observability and Ambiguity:** While the IDS solves the volume problem, it introduces noise. The agent never has a perfect view of the attacker's state. Alerts are often fragmented, low-level, or false positives. The challenge lies in mapping these distinct, often ambiguous technical indicators to a high-level strategic intent without human intervention.

2. **Temporal Dependency and Timing:** Deception is time-sensitive. Engaging an attacker is not just about showing the right asset, but showing it at the right time. Exposing a service too late results in a missed opportunity. The agent must understand the temporal flow of an intrusion to maintain causal consistency.

To address these gaps, this thesis investigates whether an AI-driven agentic architecture can autonomously manage honeypot exposure. The design of the proposed methodology is guided by the need to answer three fundamental questions regarding the agent's capabilities:

- **RQ1 (Understanding):** Can the agent correctly understand the attacker's exploitation phase within the intrusion chain based solely on fragmented network evidence?

- **RQ2 (Engagement):** Can the agent maximize the depth of the intrusion by dynamically adapting the attack surface, even when the adversary behaves probabilistically?

- **RQ3 (Efficiency):** Can the agent maintain this engagement efficiently, exposing assets only when necessary, thus optimizing resource usage?

To satisfactorily answer these questions, the system must meet specific functional and performance requirements. These requirements directly correspond to the evaluation metrics used later in this work.

Requirement	Description	Introduced Metric
Contextual Awareness	The agent must accurately reconstruct the chronological and causal order of the attack phases from raw logs.	*Graph Inference Accuracy*
Sustained Engagement	The agent must ensure the attacker is not blocked by a lack of targets. It must maximize the progression of the attack to gather intelligence.	*Exploitation Percentage*
Resource Optimization	The agent must minimize unnecessary exposures. It should guide the attacker to the objective using the minimum necessary visibility steps.	*Exposure Efficiency*

Table 4.1: **System Requirements for AI Agent.** This table summarizes the core functional requirements that the proposed AI-driven agent must satisfy: contextual awareness, sustained engagement, and resource optimization. Each of them are linked to the corresponding evaluation metric used to assess the agent's ability to manage honeypot exposure throughout multi-step intrusions.

4.2 Multi-Agent System Architecture

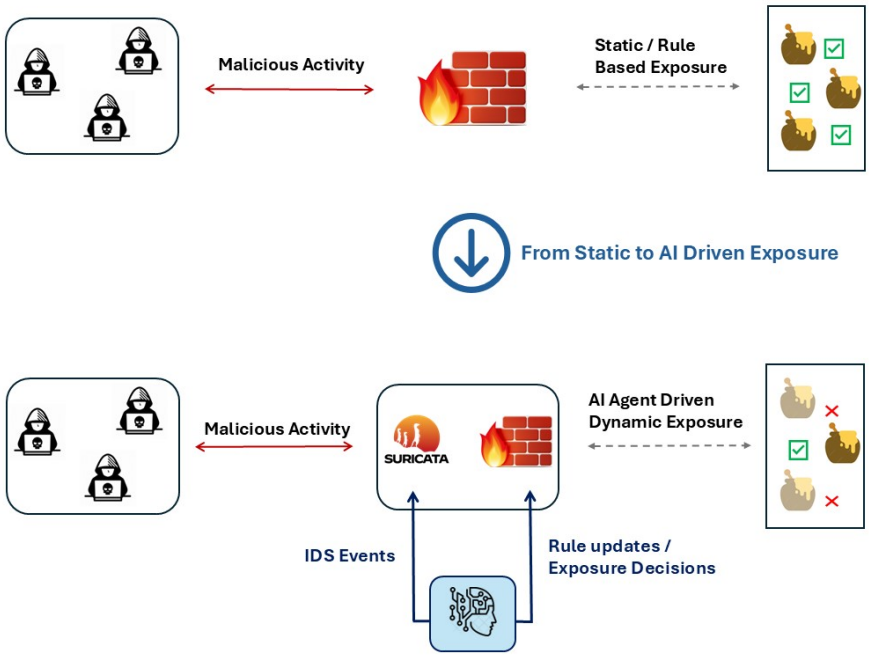

Figure 4.1: **Transition from Static to AI-Driven Honeynet Exposure.** The upper row illustrates a traditional deployment, where attackers interacts with a honeynet through a firewall that applies static, rule based policies. The lower row shows the proposed architecture: Suricata generates IDS events that are sent to an AI agent, which interprets the observed activity and returns rule updates and exposure decisions to the firewall. As a result, only a selected honeypot is actively exposed while the others are hidden, enabling dynamically adapted deception.

The proposed architecture's primary objective is the optimization of attackers' engagement, through proper asset exposure within a resource constrained environment, along with malicious activity analysis. The agent design rely on the assumption that malicious actors behavior can be decomposed into sequential stages aligned with the `MITRE ATT&CK` framework. The System architecture is guided by three fundamental principles. First, all adaptive behavior must be **evidence-based**: any inference in terms of exploitation or attack graph progress must rely on confirmed intrusion detection system (IDS) alerts. This constraint ensures that the inference process is grounded in observable data and avoids speculative or unverifiable assumptions. Second, the system emphasizes autonomous planning and enforcement through exposure strategies and firewall configuration updates without human intervention. Lastly, the principle of justifiability and transparency is inherent in the operation of every agent: each node output is recorded to enable full traceability and analysis.

The multi-agent architecture consists of five nodes, each corresponding to a distinct operational role:

1. Network Aggregation Node fetches the IDS alerts, container status, and the active rules of the firewall.
2. Attack Inference Node the interprets aggregated alerts, previous exploitation states and attack graph to infer the evolving attack steps and determine the exploitation state of each container.
3. The Exposure Manager Node determines which container to expose to the attacker in the next iteration.
4. Firewall Manager Node enforces the chosen exposure strategy into concrete network configurations through iptables rule management over a REST API.
5. Persistence Node saves the global state into memory for subsequent iterations information retrieval and benchmark analysis.

Together, the components form a cohesive decision loop that each iteration adapts the service exposure strategy based on verified attacker behavior.

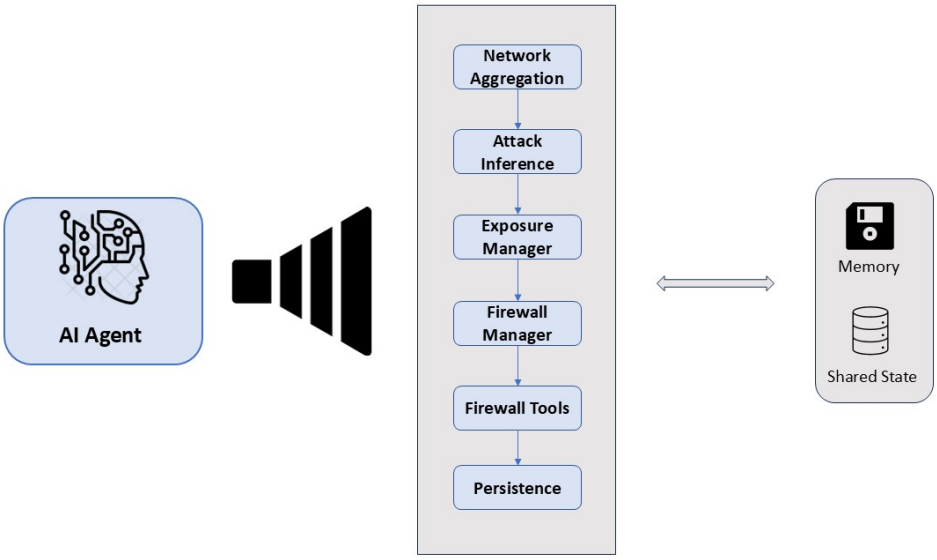

Figure 4.2: **Overview of the Agentic Architecture.** The figure illustrates the proposed AI-driven agent that reasons over network observations and interacts with a modular control pipeline composed of network aggregation, attack inference, exposure management, firewall orchestration, and persistence components. The agent and these components share a common memory and state store, enabling the system to track ongoing intrusions and dynamically adjust honeypot exposure over time.

The multi-agent architecture is structured as a directed graph where each node accomplishes one dedicated task. Each of the nodes takes structured inputs received from the previous node, processes them following its designated role and provides structured outputs which modify the global state of the system. Due to its clearly defined steps, I decided to rely on **LangGraph** as development framework, since it has been built to develop graph structured multi-agent systems. Its architecture is naturally matching the desired orchestration model allowing the specification of the pipeline steps in terms of the graph nodes and the coordination logic in terms of directed edges. This choice guaranteed scalable and transparent implementation of the pipeline, allowing the entire output trace of every agent remains entirely observable and reproducible.

Node	**Primary Function**	**Key Inputs**	**Key Outputs**	**Role in System**	**Type**
Network Gathering Node	Collects recent IDS, firewall, and honeypot data	IDS logs, Docker containers network information, firewall rules	IDS security events, container and firewall state	Provides structured alerts and network configuration	Tool Node
Graph and Exploitation Inference Node	Updates attack graph and exploitation states	Summarized events, prior graph, prior exploitation	Updated attack graph, exploitation levels, reasoning log	Infers attacker progression	LLM + Prompt
Exposure Manager Node	Chooses honeypots to expose based on evidence	Honeypot configuration, exploitation levels, exposure history	Selected honeypot, reasoning log, lockdown indicator	Plans adaptive exposure strategies	LLM + Prompt
Firewall Manager Node	Enforces exposure strategy via rule updates	Selected honeypot, current firewall and container configs	Rule changes and reasoning trace	Executes exposure decisions	LLM + Tools
Persistence Node	Stores iteration outputs and updates exposure registry	All node outputs	Memory snapshot and updated exposure registry	Ensures reproducibility and benchmarking consistency	Tool Node

Table 4.2: Summary of Agent Nodes and Interactions

4.2.1 Shared State

All nodes operate over a single shared state, `AgentState`, which consists in a data structure that is shared between nodes at each iteration. Each component receives the current `AgentState`, performs its computation, and returns data structures that updates the relevant fields.

Field	Description	Produced by	Consumed by
`security_events`	IDS alerts	Network Aggregation	Attack Inference
`firewall_config`	Firewall rules	Network Aggregation	Firewall Manager
`vulnerable_containers`	Deployed vulnerable containers metadata	Network Aggregation	Attack Inference, Exposure Manager
`inferred_attack_graph`	Current attacker progression graph	Attack Inference	Exposure Manager
`containers_exploitation`	Per container exploitation state	Attack Inference	Exposure Manager
`selected_container`	Selected container to expose in the next epoch	Exposure Manager	Firewall Manager
`lockdown_status`	Boolean status indicating an enforced containment mode	Exposure Manager	–

Table 4.3: **Shared `AgentState` fields and data flow among agents.** This table summarizes the key fields stored in the shared `AgentState` object. For each field, it indicates its purpose, which agent or system component produces it, and which agents consume it. This clarifies how information flows between the Network Aggregation, Attack Inference, Exposure Management, and Firewall Manager agents to coordinate attack detection, honeypot exposure decisions, and containment actions.

4.2.2 Episodic Memory

The system incorporates an Episodic Memory to provide the agent with a short-term operational context and a foundation for future longitudinal learning. This memory component, implemented by the `EpisodicMemory` class, operates independently of the real-time state flow and serves as an immutable log of the system's execution history. The key mechanisms of the class are:

- **State Logging**: At the conclusion of each operational loop, a snapshot of the state data is tagged with a unique iteration ID and a precise timestamp. This transformation moves the data from a mutable state to a persistent, indexed record using an `InMemoryStore`.
- **Context Retrieval**: The memory allows nodes, particularly the Attack Inference and Exposure Manager, to access a limited window of the most recent iterations. This short term recall is crucial for contextualizing sequential events.

This episodic memory architecture establishes a baseline on which future works can build the necessary infrastructure for implementing a Retrieval Augmented Generation (RAG) approach in future work. By persistently logging historical interactions, the memory becomes the foundational knowledge base that a RAG mechanism can query to transition

the system from a stateless to a dynamic, experience-based learner. However, the integration of a full RAG pipeline leveraging this memory as a queryable knowledge base is left as future work and has not yet been implemented in the current prototype.

4.2.3 Network Gathering Node

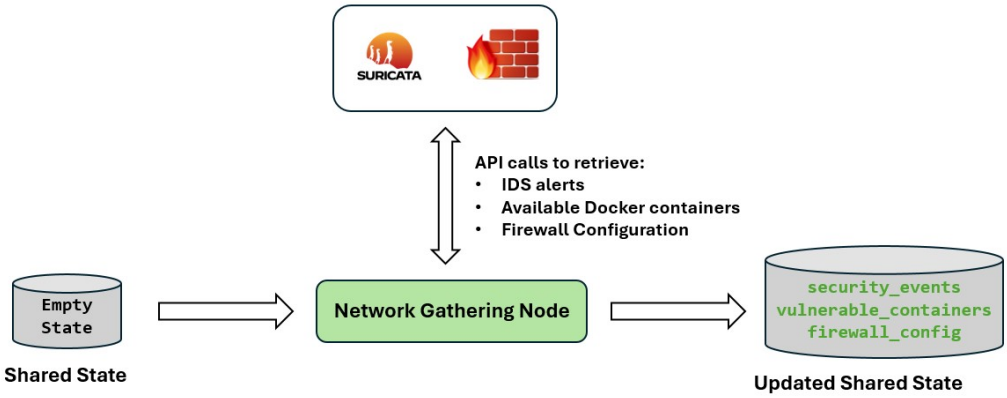

Figure 4.3: **Network Gathering Node: Tool Node and Shared State Update**. This diagram illustrates the Tool Node responsible for collecting real time network and system information to update the agent's shared state. The Network Gathering Node queries multiple APIs servers such as Suricata, for IDS alerts, the firewall, for firewall rules and Docker for container availability. It then produces an updated shared state containing security_events, vulnerable_containers, and firewall_config, enabling downstream agents to reason about the environment.

The operational sequence begins with the **Network Aggregation Node**, whose primary objective is to collect the most recent environmental data from the monitored network. This node serves as the system's perceptual layer, ensuring that every subsequent reasoning step operates with the most up to date situational awareness.
It queries three key components: the intrusion detection system (IDS), the Docker environment, and the programmable firewall. From the IDS, it retrieves structured alerts within a defined time window, providing a snapshot of ongoing or recent malicious activity. Then, the alerts are grouped for each honeypot and the duplicates are compacted in a count number avoid information redundancy. Subsequently, the node interacts with the Docker API to enumerate active honeypot containers, gathering metadata such as container name, IP addresses, and listening ports. Finally, it communicates with the

firewall's REST API to extract the current set of active rules. IDS alerts, container configurations, and firewall policies forms the comprehensive perception package that is injected into the shared state and subsequently used as inputs by following agents to analyze threats and plan defensive actions. The relative code is present in appendix A.

4.2.4 Attack Inference Node

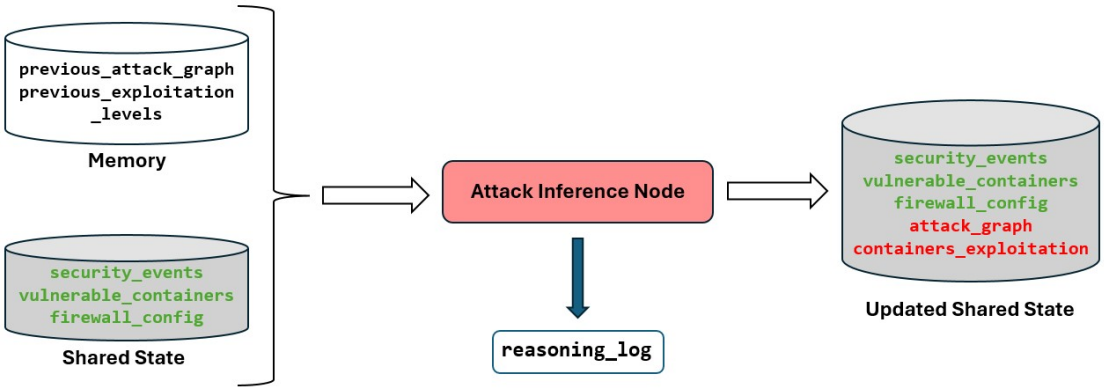

Figure 4.4: **Attack Inference Node: LLM + Prompt Node and Shared State Update**. The node receives previous context together with fresh network data. Using this information, it reasons about ongoing activity and produces both an updated attack graph and container exploitation levels, which are added to the updated shared state. A reasoning log is also generated, documenting the model's thought process for auditing.

The next component, the **Attack Inference Node**, is responsible for interpreting the summarized events in light of previous iterations context. It is responsible to infer the current structure of the attack graph and assess the exploitation level of each container, in other words reconstructing the adversary's evolving trajectory through the systems. The node consumes the latest event summary and merges it with previously inferred attack graph and exploitation states. This information is necessary to interpret new alerts to update the previous inferred attack steps. During the execution, the node retrieves the latest stored inference results from the episodic memory, namely previous `exploitation_levels` and `attack_graph`. In cases where no prior data are available, such as the first iteration, the node initializes a baseline graph and assigns to each vulnerable container an exploitation level data structure with default values. This baseline

ensures operational resilience, even in the absence of historical context, and allows the node to construct an evolving model of the attack scenario over successive epochs. Once initialization is complete, the node proceeds to iterate over each vulnerable container defined in the system state.
The security events are processed separately for each container. If no matching events are found, further processing for that container is skipped, preserving computational efficiency and avoiding unnecessary LLM's calls. Conversely, the node invokes the LLM and obtains in output the updates to add to the state attack graph representation. Furthermore, the response contains a "`reasoning`" log providing textual justification for each inferred modification.
Before integrating the LLM output in the state attack graph, the node applies a sequence consistency check through the `enfore_backfill_on_deltas` function. This mechanism ensures that inferred changes are coherent with the temporal progression of previously observed events. In particular, it inserts any missing precondition node necessary for logical completeness. Then, for each container, the corresponding exploitation record deducted from the attack steps is updated with new levels and supporting evidence quotations from IDS alerts. After all containers have been processed, the node assembles the complete updated attack graph and exploitation levels. The relative code is present in appendix B.
Example Output:

```
Inferred Attack Graph: {'edges': [{'from': '192.168.100.2', 'to':
'172.20.0.5', 'phases': [{'phase': 'scan', 'evidence_quotes': ['ET SCAN
Suspicious inbound to MSSQL port 1433', 'ET SCAN Suspicious inbound to
mySQL port 3306', 'POSSBL PORT SCAN (NMAP -sS)']}], 'current_phase':
'scan', 'vector': 'scan'}], 'interesting': []}

Containers' Exploitaiton: [{'ip': '172.20.0.5', 'service':
'cve-2014-6271-web-1', 'level_prev': 0, 'level_new': 25, 'changed':
True, 'evidence_quotes': ['ET SCAN Suspicious inbound to MSSQL port
1433']}, {'ip': '172.20.0.4', 'service': 'cve-2015-5254-activemq-1',
'level_prev': 0, 'level_new': 0, 'changed': False, 'evidence_quotes':
[]}, {'ip': '172.20.0.3', 'service': 's2-057-struts2-1', 'level_prev':
0, 'level_new': 0, 'changed': False, 'evidence_quotes': []}, {'ip':
'172.20.0.10', 'service': 'cve-2021-22205-gitlab-1-proxy',
'level_prev': 0, 'level_new': 0, 'changed': False, 'evidence_quotes':
[]}]
```

LLM is instructed with the directives defined in appendix F:

4.2.5 Exposure Manager Node

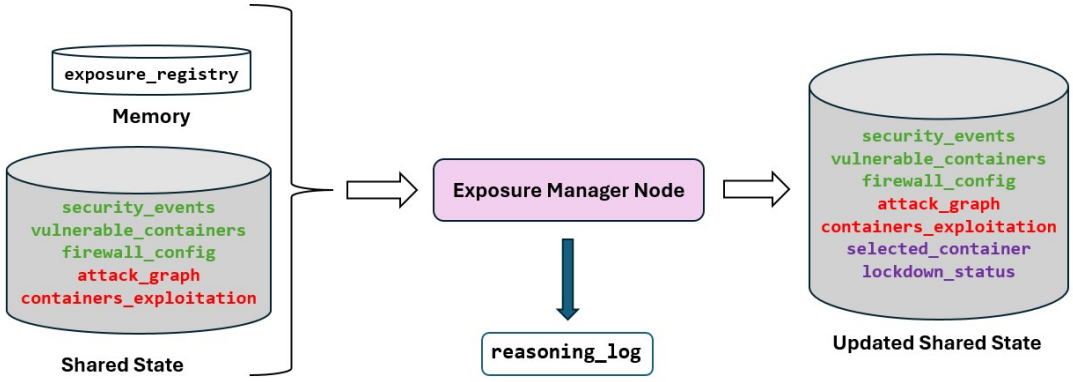

Figure 4.5: **Exposure Manager Node: LLM + Prompt Node and Shared State Update**. The node consumes both memory retrieved fields and the current shared state. Using this information, it decides which container should be exposed next or whether the system should enter a lockdown state. These decisions are written into the updated shared state, along with a generated reasoning log that records the model's decision process auditing.

The **Exposure Manager Node** implements the decision logic that determines which single honeypot container is exposed to the attacker at each epoch.
Its behavior is driven by three inputs: the current inventory of vulnerable containers, the per-container exploitation assessment produced by the preceding node, and a persistent `exposure_registry`. This registry records how long each service has been exposed to check if the number of exposure epochs is compliant with system policies. The prompt instructs the LLM to expose exactly one container each epoch, selecting it based on exploitation progress, prior exposure history, and overall coverage.
Containers that have shown exploitation progress remain exposed for at least two consecutive epochs, while those that reach full exploitation or repeatedly show no progress are marked as exhausted and excluded from future selections. Priority is given to containers that have not yet been exposed, ensuring that all potential targets are eventually tested. When all containers become either fully exploited or exhausted, the policy enforces a lockdown mode, isolating all containers from the attacker. The final output of the LLM includes the selected honeypot, the reasoning chain behind the decision, and an explicit indicator of whether `lockdown` should be enforced.

The relative code is present in appendix C.
Example Output:

```
Reasoning: The container 172.20.0.5 (cve-2014-6271-web-1) was exposed
last epoch and showed progress (exploitation level increased from 0 to
25). According to policy, we must continue exposing it for at least two
consecutive epochs after progress. Other containers have not been
exposed yet but minimum exposure window and extension on progress take
priority. Therefore, continue exposing 172.20.0.5 this epoch.

Selected Container: {'ip': '172.20.0.5', 'service': 'cve-2014-6271-
web-1', 'current_level': 25}

Lockdown: False
```

LLM is instructed with the directives defined in appendix G:

4.2.6 Firewall Manager Node

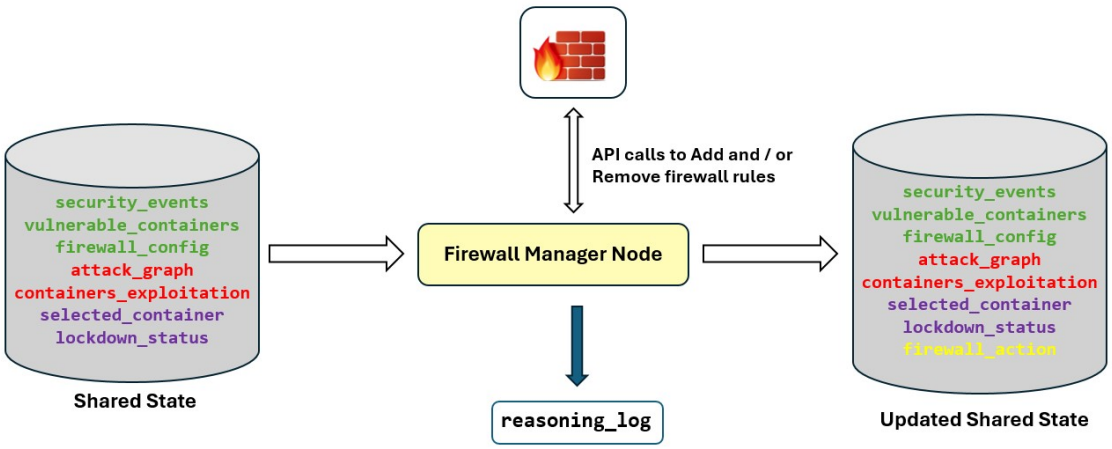

Figure 4.6: **Firewall Node: LLM + Prompt Node and Shared State Update**. he node reads the current shared state and decides what firewall modifications are required. It then issues API calls to add or remove firewall rules in real time. The resulting action is stored in the updated shared state, and a reasoning log is produced to document the decision-making process for auditing.

Once the exposure plan has been established, the **Firewall Manager Node** transforms it into enforceable network rules. This component receives the selected container to expose from the Exposure Manager, along with the latest firewall and active vulnerable containers configuration. This information enables the output of a precise set of actions to update iptables rules, determining which connections should be allowed or blocked. The translation process is guided by an LLM prompt that ensures decisions remain consistent with the overall exposure strategy.
The node outputs both the list of rule changes and an explanation of the reasoning behind each action. This approach maintains transparency and ensures that every firewall update can be traced back to an explicit logical justification.
The relative code is present in appendix D.
Example Output:

```
Reasoning:Verified that the container 172.20.0.5 exists in the
available containers list.
Current firewall rules already include bidirectional allow rules
between attacker subnet 192.168.100.0/24 and container 172.20.0.5 on
tcp protocol (rules 1 and 2).
No additional changes are necessary as the exposure is already enforced
exactly as requested.

Action: []
```

LLM is instructed with the directives defined in appendix H:

4.2.7 Persistence Node

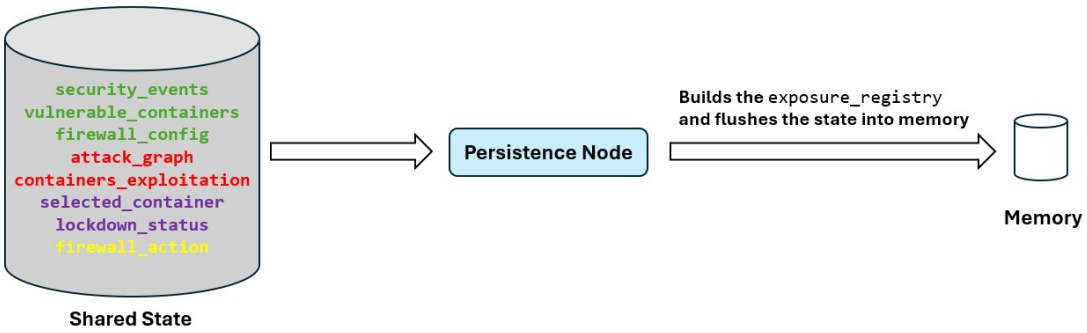

Figure 4.7: **Persistence Node: Tool Node and Memory Update**
Persistence Node is responsible for converting the final shared state of each iteration into memory for the agent's next iterations. After receiving the complete shared state, the node processes this information to build or update the exposure registry. It then flushes the consolidated state into memory, ensuring that future reasoning steps have access to historical context and past exposure decisions.

The final stage in the agent cycle is handled by the **Persistence Node**, which records all relevant outputs into the system's episodic memory. This component ensures that each iteration is stored as a complete snapshot, including event summaries, inferred graphs, exposure decisions, and applied firewall rules. By preserving these states, the system enables reproducibility, analysis, and benchmarking of agentic performance over time. Furthermore, the persistence mechanism updates the exposure registry to maintain accurate records of how frequently and for how long each honeypot has been visible to the attacker. This cumulative record not only supports explainability and traceability but also enables future iterations of the system to make decisions based on rich temporal context. The relative code is present in appendix E.

Chapter 5

Agent Testing Environment

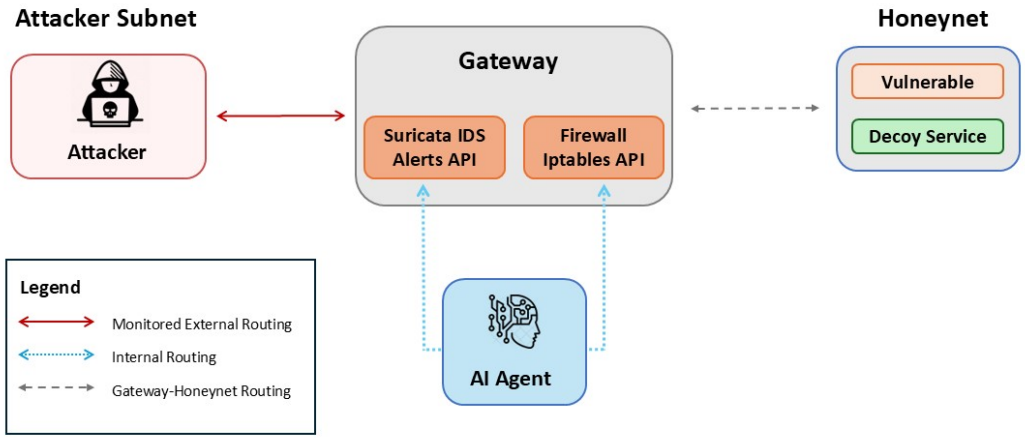

Figure 5.1: **Testing Environment Network Topology**. This diagram provides an overview of the full experimental environment, showing how an attacker interacts with the honeynet through a controlled gateway monitored by an AI agent. The attacker sends traffic toward the gateway, where Suricata IDS alerts and firewall controls are exposed via APIs. The AI agent monitors these APIs, analyzes traffic patterns, and adjusts firewall rules or exposure strategies accordingly. Traffic then reaches the honeynet, which hosts both vulnerable services and decoy services, allowing the agent to study attacker behavior.

To evaluate the Multi-Agent System in a controlled yet realistic setting, I designed a containerized network simulation environment. This setup emulates a network where vulnerable services are deliberately deployed to attract and record attacker activity. The environment is instantiated through dedicated `docker-compose` files and initialization scripts that define the topology, deploy containers, and configure network routing. Figure 5.1 illustrates the logical topology of this network. The environment is logically divided into two main zones. The **Attacker Subnet** hosts malicious activity sources and simulates an untrusted external network from which all intrusions originate. For agent evaluation purposes, the external network has only one active IP address, referred as *attacker*. The **Honeynet**, by contrast, contains all deployed containers and is strictly accessible only through the gateway. This configuration ensures full visibility of attacker interactions, while maintaining the ability to selectively expose or conceal services. Within the honeynet, several vulnerable containers have been deployed to represent a heterogeneous set of attack surfaces commonly exploited in real world scenarios. The containers were chosen among a set of vulnerable configuration from the Vulhub repository [51].
A gateway container, which operates as the entry point between the external network and the internal honeynet, integrates both a programmable firewall and an Intrusion Detection System, specifically Suricata. The configuration ensures that every packet flow is monitored and subject to dynamic access control. Moreover, it exposes APIs for firewall rule management and IDS alerts retrieval accessible only from the AI agent. All traffic in both directions is routed through Suricata such that the IDS captures each malicious attempt and generates structured alerts in a specific file format, `eve.json`. There is a dedicated Suricata API server which parses such logs and provides endpoints for obtaining latest alerts.
In each iteration of the orchestration loop, the agents query this API to harvest new evidence, infer exploitation progress, and adapt the exposure strategy accordingly. Once decisions are taken, execution happens through REST calls to the firewall API, which dynamically manages service exposure.

5.1 Attacker Behavior and Graph Modeling

In parallel with the defensive architecture, a simulated adversary was implemented to provide a reproducible and measurable source of attacks. The attacker operates through a multi stage script, `manager_exploit.py`, which embodies a structured intrusion process aligned with a formal attack graph. To ensure behavioural realism and analytical consistency, the simulated attacker's workflow is aligned with the MITRE ATT&CK framework. Each phase of the scripted kill chain corresponds to a specific ATT&CK tactic:

1. Service Scan → Reconnaissance
2. Initial Access (RCE) → Initial Access / Execution
3. Data Exfiltration (user-level) → Collection / Exfiltration
4. Privilege Escalation → Privilege Escalation

5. Root-Level Data Exfiltration → Exfiltration (privileged)

MITRE ATT&CK framework does not explicitly define a distinct technique corresponding exclusively to "data exfiltration at root or SYSTEM privilege level". In our simulation we introduce this explicit phase in order to enable the agent to discriminate between exfiltration occurring at user privilege versus elevated privilege, thus capturing a higher threat severity scenario for modelling and inference. The attacker's progress is modeled through a directed, edge labeled hypergraph $G = (V, E, \Lambda)$, where nodes V represent the attacker A and the set of asset nodes $H \subseteq IP \times S$ corresponding to honeypot services. Each directed edge $e \in E$ encodes a tactic executed against a target, labeled by $\lambda \in \Lambda$, defined as:

$$\lambda = (vector, \delta, \phi)$$

where $vector \in \{\texttt{scan}, \texttt{rce}, \texttt{exfil}, \texttt{privesc}, \texttt{root-exfil}\}$ identifies the MITRE ATT&CK phase, $\delta \in P^+$ is the ordered sequence of phases observed on the edge, and $\phi \in P$ denotes the current phase of that edge. Phases are executed in a strictly increasing order:

$$\texttt{scan} < \texttt{initial-access/rce} < \texttt{data-exfil-user} < \texttt{priv-esc} < \texttt{data-exfil-root}$$

The attacker process begins with scanning and recon, with open ports and active services revealed by Nmap scanning. Protocol specific fingerprinting procedures determine service type and version for each discovered endpoint such as GitLab instances by HTTP header analysis or scanning Struts paths for vulnerable endpoints. Exploitation is synchronized based on the notion of epochs such that each vulnerable service has a local epoch counter tracking its time of exposure. This ensures the attacker progresses gradually with the sole action authorized at this phase being executed, from initial access to post-exploitation. The exploit modules all run in parallel and are synchronized with the assistance of a thread pool that collects results asynchronously. Successful compromises generate synthetic flag tokens, which serve as ground truth indicators of the efficacy of exploitation. These outcomes are stored in structured JSON files such that the resultant data become the standard against which the efficacy of the agent's inferences may be measured. By incorporating autonomous orchestration, containerized deployments, programmability during enforcement, and the deterministic attacker model, the whole framework also experiences excellent experimental controllability and reproducibility.

For each vulnerable container, a custom exploit script was developed following a structured kill chain composed of five phases:

1. **service scan**
2. **initial access**
3. **data exfiltration**
4. **privilege escalation**
5. **root-level data exfiltration**

The table below summarizes which phases were implemented for each service in the testing environment.

5.2 Discrete Event Simulation Environment

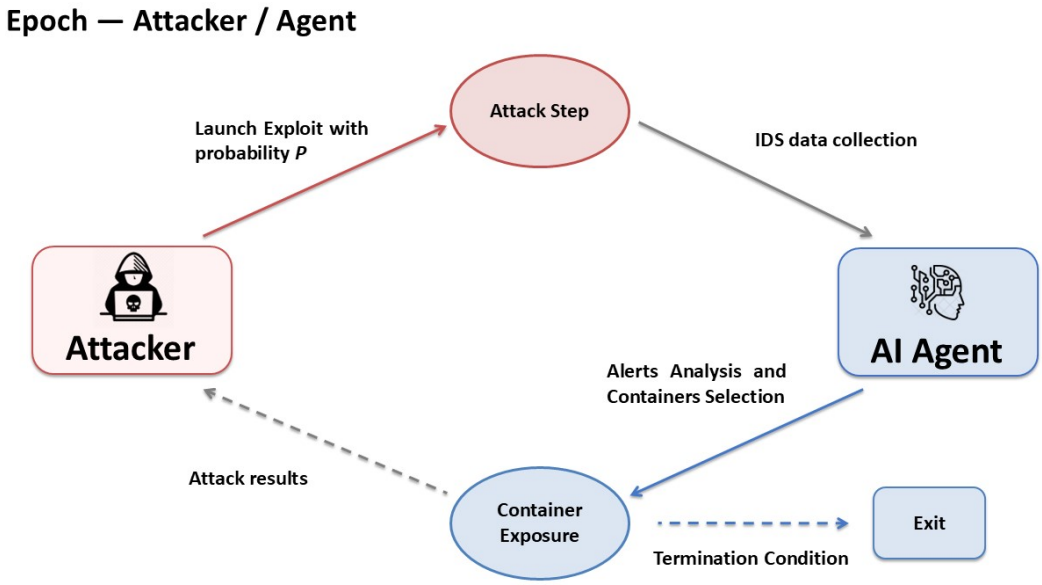

Figure 5.2: **Discrete-Event Simulator**. Launch Probability P depends on test case: Deterministic Attacker always launches the exploit if a vulnerable service is exposed ($P = 1$). Consecutive Attacker launches or continues the exploit only if the same vulnerable service was exposed in the previous epoch ($P = 1$ if condition met else $P = 0$). Probabilistic Attacker's exploit execution probability decreases if the vulnerable service is not consecutively exposed ($0 < P < 1$)

The evaluation proceeds as a sequence of discrete epochs. Each epoch consists of a cycle of adversarial activity followed by the agent's reasoning and adaptation. The benchmark cycle is managed by an automatic script, which has the role of an `orchestrator`, allowing

to test the agent in different conditions autonomously. Each epoch follows the same ordered stages:

1. **Attack execution**
   The attacker container runs the attack script (`manager_exploit.py`), which performs discovery and then decides whether to attempt exploitation of each discovered service according to its configured attack simulation:

   (a) **Deterministic Mode:** the attacker always attempts exploitation whenever it detects a vulnerable service, hence with a probability $P = 1/$

   (b) **Consecutive Mode:** the attacker will only attempt an exploit if the same vulnerable service was exposed in the previous epoch; otherwise the attacker does not attempt exploitation. The probability results to be $P = 1$ if the service was exposed last epoch, $P = 0$ otherwise.

   (c) **Probabilistic Mode:** lastly, the attacker's attempt probability decays with the umber of epochs since the service was last exposed. Intuitively, a service that has been hidden for several epochs is less likely to be targeted on the next attempt.
   Let g be the number of epochs for which the service has not been exposed after the a first selection. Let d the per epoch decay factor, `DECAY_PER_MISSING_EPOCH`. Let p_{min} be the minimum attempt probability floor `MIN_ATTEMPT_PROB`. The attempt probability P is computed as:

   $$\mathrm{P} = \begin{cases} 1, & \text{if } g \leq 0 \\ max(p_{min}, 1 - d \cdot g), & \text{if } g > 0 \end{cases}$$

   The attacker performs a single probabilistic draw sampling from a uniform random $r \in [0,1]$ and attempts the exploit iff $r \leq P$.

2. **Agent Analysis and Container Selection**
   The multi-agent system ingests IDS alerts, current firewall configuration, and episodic memory data to reconstruct the evolving attack graph, estimate the exploitation level for each container, select the next container to expose and properly change the firewall rule.

3. **Container Exposure**
   A brief waiting period is configured to ensure the firewall rule executed by the agent have been applied before the next epoch begins.

4. **Epoch Continuation and Termination Condition**
   The orchestrator evaluates the stopping conditions which can be:

   (a) `lockdown` phase instructed by the agent

   (b) `the maximum epoch number` configured at the beginning of the benchmark run.

If the benchmark continues, it waits a configurable inter epoch delay and start the next run.

Attacker outputs	Agent outputs
`services_detected`: list of discovered services (e.g., IP + service identifiers).	`firewall_rules_added`: list of rules the agent requested to add (strings or rule objects).
`exploits_attempted`: exploited paths/techniques the attacker tried.	`firewall_rules_removed`: list of rules removed by the agent (if any).
`services_successfully_exploited`: services actually compromised by the attacker.	`honeypots_exposed`: which honeypots/services the agent chose to expose (ip, service, level, epoch).
`flags_captured`: synthetic flags retrieved by attacker (ground truth of compromise).	`honeypots_exploitation`: per-honeypot exploitation estimates (previous/new level, evidence quotes, changed boolean).
`attack_success_rate`: scalar summarizing attacker success for the epoch.	`inferred_attack_graph`: agent's reconstructed graph (nodes/edges, vectors, phases, evidence).
-	`service_epoch_context`: agent-collected context per service (empty in the example but reserved for agent annotations).

Table 5.1: Attacker vs Agent outputs recorded in the epoch JSON

5.3 Vulnerable Containers Deployed

The services chosen represent common application layer services that may be exposed in a real deployment. Containers are affected by reported vulnerabilities that are exploited through custom scripts to mimic an attacker behavior. There are in total 8 containers available in the testing environment. For half of them, I implemented a staged attack script aiming to capture some flags placed inside the containers' file system. The other four containers are only scanned by the simulated attacker, emulating services not valuable for malicious activity.

Docker

An exposed Docker daemon API can lead to remote code execution (RCE). When the daemon allows unauthenticated access to the Docker API, an attacker can execute container commands, spawn privileged containers, and gain root-level access. Because containers often run with elevated privileges or host/kernel access, a successful Docker API exploit typically results in full host compromise, making this vector particularly dangerous in cloud or orchestrated environments.

Apache Struts

CVE-2018-11776 allows a remote code execution vulnerability in Apache Struts. The flow begins with identifying a Struts-based web application, then performing directory traversal/path enumeration to discover vulnerable endpoints, followed by execution of the exploit. This attack pathway emphasizes the risk of classic enterprise web frameworks: a single vulnerable Struts endpoint can lead to full system takeover due to inadequate input sanitization and insecure default configurations. Because many legacy systems still run Struts, this remains a significant concern.

Gitlab

This diagram illustrates the exploitation path for CVE-2021-22205 in GitLab CE/EE. An issue has been discovered in GitLab CE/EE affecting the versions starting from 11.9. GitLab was not properly validating image files that is passed to a file parser which resulted in an unauthenticated remote command execution. Once initial access is obtained, the attacker may exfiltrate data and potentially escalate privileges, leading to full system compromise of the GitLab instance. This flow highlights why GitLab represents a high risk target in DevOps environments: its central role in code repositories and CI/CD pipelines amplifies the impact of compromise.

Xdebug

This diagram outlines a typical attack sequence against the PHP debugging extension Xdebug when misconfigured for remote debugging. XDebug is a PHP extension used for debugging PHP code. When remote debugging mode is enabled with appropriate settings, an attacker can execute arbitrary PHP code on the target server by exploiting the debug protocol (DBGp). An unauthenticated HTTP request triggers Xdebug's DBGp session, the server connects back to the attacker's host, and the attacker sends debug commands to execute arbitrary PHP code. From there, the attacker may perform data exfiltration, privilege escalation, if web-server privileges are weak, and ultimately system compromise. Because Xdebug is meant for development environments, its presence on production systems significantly increases exposure and often correlates with poor deployment hygiene.
The figure 5.3 shows all the possible attacker paths in the benchmark environment concerning the deployed containers whether they result exploitable or not.

Service	**Scan**	**Initial Access**	**Data Exfil**	**Priv. Esc.**	**Root Exfil**
GitLab	✓	✓	✓	✓	✓
Xdebug	✓	✓	✓	✓	✓
Apache Struts	✓	✓	-	✓	✓
Docker API	✓	✓	✓	-	-
Others	✓	-	-	-	-

Table 5.2: **Implemented Attack Phases per Deployed Service.** The table summarizes, for each containerized service in the honeynet, which stages of the simulated intrusion chain are implemented: scanning, initial access, data exfiltration, privilege escalation, and root-level exfiltration. Green checkmarks denote that a given phase is available for that service in the experiment, while red dashes indicate that the corresponding phase is not modeled. The *Others* row refers to decoy vulnerable containers for which no exploit is implemented; they are therefore only subject to the scanning phase.

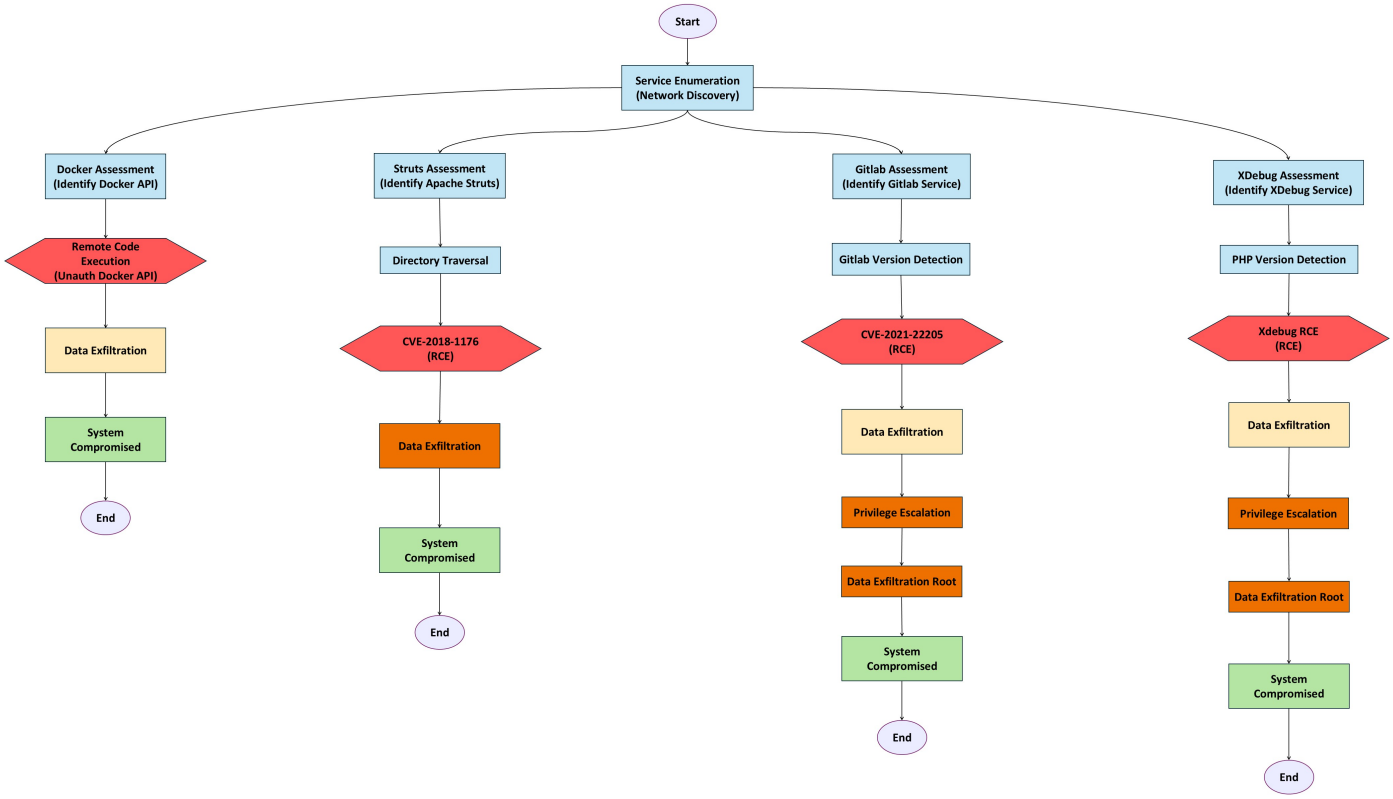

Figure 5.3: **Simulated Attack Paths** The figure illustrates the attack tree underpinning the experimental setup, starting from network service enumeration and branching into four assessments: Docker API, Apache Struts, GitLab, and Xdebug. For each service, it shows the ordered sequence of attacker actions—such as version or service detection, directory traversal, remote code execution via specific vulnerabilities, data exfiltration, privilege escalation, and root-level exfiltration—culminating in full system compromise and defining the ground-truth intrusion chains used in the simulation.

Chapter 6

Results

6.1 Evaluation Parameters

Evaluation is conducted under a diverse set of controlled scenarios, increasing the level of realism, to evaluate the system's ability to efficiently manage the honeynet. The experiments differ in container configurations, the analytical resources available to the defending agent and attacker behavior.
Two principal container setups are considered. In the mixed deployment scenario, some containers are deliberately exploitable from the attacker script, while the others serve as decoys. This configuration tests the system's ability to distinguish genuine compromises from background reconnaissance noise, a key capability for avoiding false positives and overexposure of non critical services. The second case, named fully exploitable deployment, subjects the agent to a maximal stress test where all containers can be compromised. In this case, every discovered service poses a legitimate risk, allowing the system's inference mechanisms and exposure strategy to be tested under extreme adversarial conditions.
Beyond container composition, several experimental variations are introduced to assess the reasoning process under different operational constraints. The first variation concerns language model evaluation, in which the agent is tested using multiple LLM configurations to analyze the impact of model size and version on reasoning quality.
Three different LLMs are used:

1. **GPT-4.1:** proprietary solution, state of the art model up to August 2025
2. **GPT-4.1-mini:** proprietary solution, smaller and cheaper option
3. **GPT-OSS:120b:** open-weights, can be hosted locally

A second experimental axis focuses on attack simulation. Three different simulation are implemented: `deterministic`, where the attacker executes attack and escalates the attack stages on vulnerable containers whenever the service is exposed. This scenario, although naive, is used to assess the inference accuracy of the agent and its ability to understand what is "happening" in the network. Second scenario, name `consecutive`, where the attacker executes the attack only if the vulnerable container is exposed for

consecutive epochs. This time, the evaluation has a focus on the optimal exposure of the vulnerable containers resulting in an optimal use of resources. Third, `probabilistic`, which is a trade off between the previous cases, since the attacker has a lower probability of executing the attack if the vulnerable service is not exposed consecutively. Together these design choices define a benchmarking workflow that combines realism with reproducibility. In conclusion, the architecture is evaluated across 27 distinct test configurations, each executed three times to obtain average performance scores, resulting in a total of 81 tests. The methodology captures both the correctness of the agent's service exposure and its ability to infer the adversary's true progression through the attack chain.

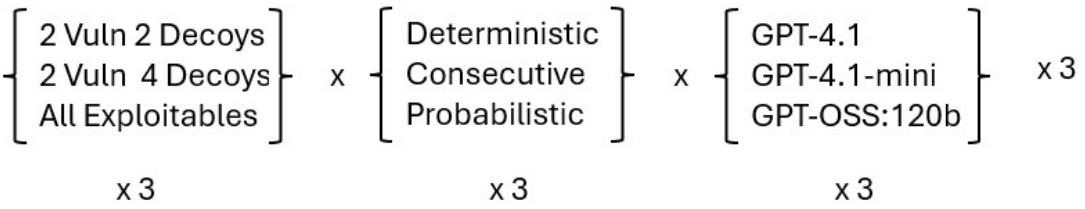

Figure 6.1: **Experimental Test-Case Matrix.** The figure depicts full set of the evaluated scenarios: three honeynet configurations, three attacker behaviors, and three model variants used as the reasoning engine. Each combination of configuration, attacker behavior, and model is executed three times to obtain robust statistics.

6.2 Metrics Definition

The evaluation framework utilizes formalized performance metrics to assess the system's reasoning capabilities and operational efficiency. Specifically, the study focuses on **Graph Inference Accuracy** to evaluate the precision of attack graph inference, and two efficiency metrics, **Exploitation Achieved** and **Exposure Efficiency**, to measure the effectiveness of the autonomous exposure strategy.

Metric	Definition	Unit
Graph Inference Accuracy	Accuracy in attack graph inference, measuring the correct recovery of causal attack transitions.	%
Exploitation Achieved	The achieved level of exploitation of the deployed assets relative to the maximum potential.	%
Exposure Efficiency	The number of steps utilized relative to the optimal solution ("golden steps").	%

Table 6.1: Summary of Selected Performance Metrics

6.2.1 Graph Inference Accuracy

Graph Inference Accuracy quantifies how closely the inferred attack graph $\widehat{\mathcal{G}}$ matches the ground truth $\mathcal{G}^\star$. Let $\mathcal{G}^\star = (V^\star, E^\star)$ and $\widehat{\mathcal{G}} = (\widehat{V}, \widehat{E})$ represent the ground truth and inferred graphs, respectively, where directed edges encode ordered attack progressions. To evaluate the sequential correctness, we define the set of *phase units* for an edge with a de-duplicated Attack Graph Inference $\delta = (p_1, \ldots, p_k)$ as:

$$\mathcal{U}(\delta) \;=\; \big\{(\texttt{\_\_START\_\_}, p_1)\big\} \;\cup\; \big\{(p_i, p_{i+1}) \mid i = 1, \ldots, k-1\big\}.$$

This set captures the initial phase and all subsequent transitions. For any attacker–victim pair (u, v), let $\mathcal{U}^\star_{uv}$ and $\widehat{\mathcal{U}}_{uv}$ denote the ground truth and inferred phase unit sets. We define the `Graph Inference Accuracy` as a micro-averaged metric incorporating penalty terms for structural hallucinations:

$$\text{GraphInferenceAccuracy} \;=\; \frac{\sum_{(u,v)} \text{TP}_{uv}}{\sum_{(u,v)} \text{TP}_{uv} + \sum_{(u,v)} \widehat{\text{FP}}_{uv} + \sum_{(u,v)} \text{FN}_{uv}}$$

Here, $\text{TP}_{uv} = |\mathcal{U}^\star_{uv} \cap \widehat{\mathcal{U}}_{uv}|$ represents correctly identified transitions, and $\text{FN}_{uv} = |\mathcal{U}^\star_{uv} \setminus \widehat{\mathcal{U}}_{uv}|$ represents missed transitions. The term $\widehat{\text{FP}}_{uv}$ denotes the *penalized false positives*, defined as:

$$\widehat{\text{FP}}_{uv} = \begin{cases} |\widehat{\mathcal{U}}_{uv} \setminus \mathcal{U}^\star_{uv}| + \Omega_{uv} & \text{if } (u,v) \in E^\star \\ w \cdot |\widehat{\mathcal{U}}_{uv}| & \text{if } (u,v) \notin E^\star \end{cases}$$

where:

- Ω_{uv} is a penalty term applied when the inferred sequence contains an explicit "inferred" marker. It refers to the logical backfilling of missing intermediate attack phases that were not explicitly detected but are assumed to have occurred because a subsequent, dependent phase in the attack taxonomy was observed. It is calculated as $\alpha \cdot k_{\text{missed}}$, where α is a penalty factor (set to 0.5) and k_{missed} is the count of ground truth transitions skipped by the inference.
- w is a weighting factor (set to ≥ 1.0) applied to edges that do not exist in the ground truth, effectively penalizing hallucinated attack paths more heavily if the victim node is also hallucinated.

6.2.2 Exposure Efficiency Metrics

These metrics evaluate the agent's ability to facilitate meaningful adversarial progression through the honeypot landscape, balancing coverage against temporal cost.

Exploitation Achieved

This metric measures the level of asset exploitation relative to the total available attack surface. Let $\mathcal{F}$ be the set of unique flags captured during the run and $F_{\max}$ be the maximum attainable number of flags for the deployed configuration. The exploitation percentage is defined as:

$$\text{Exploitation Achieved} \;=\; 100 \times \frac{|\mathcal{F}|}{F_{\max}}.$$

Exposure Efficiency

Exposure Efficiency evaluates the temporal optimality of the solution. Let T be the total number of epochs utilized in the run, and $S^\star$ be the configuration-dependent `"golden steps"`, representing the minimal epochs required to reach full coverage. The metric indicates the proximity to the optimal solution:

$$\text{Exposure Efficiency} \;=\; \begin{cases} 100 \times \min\!\left(1,\, \frac{S^\star}{T}\right), & S^\star > 0 \text{ and } T > 0, \\ 100, & \text{if coverage} = 1 \text{ and } S^\star \text{ undefined.} \end{cases}$$

6.3 Results Analysis

This section presents a detailed analysis of the experimental results obtained from the agent benchmark execution. The primary objective of this analysis is to validate the methodology proposed in Chapter 3 by addressing the three rhetorical questions posed regarding Interpretation, Engagement, and Efficiency. Before diving into the results analysis, it is crucial to interpret the aggregate findings in the context of our initial research objectives.

- **RQ1 (Understanding):** *Can the agent correctly understand the attacker's exploitation phase within the intrusion chain based solely on fragmented network evidence?*
  The results indicate a strong affirmative, particularly for the best-performing model. With `Attack_Graph_Inference_Accuracy` up to 96% for GPT-4.1, the system demonstrates a high capacity to map low-level indicators into a coherent attack narrative, even in the presence of structural ambiguity.

- **RQ2 (Engagement):** *Can the agent maximize intrusion depth under uncertainty?* The high `exploitation_percentage` observed across distinct attack modes validates the agent's ability to sustain engagement. While probabilistic behaviors introduce noise, the agent successfully manages the attack surface to ensure the adversary rarely encounters a "dead end", effectively guiding them through the intrusion chain to maximize information gain.

- **RQ3 (Efficiency):** *Can the agent optimize resource usage?* The `exposure_efficiency` metric, which remains close to 100% in mixed configurations, confirms that the agent does not resort to "spray and pray" tactics. Instead, it identifies and exposes the precise sequence of assets required for the attacker to progress, validating the system's capability to manage resources autonomously and intelligently.

Each experiment is executed three times per configuration, and averages are reported to reduce the effect of stochastic variance. AThe following analysis first presents a high-level comparative summary of all models based on averaged metrics. Subsequently, a detailed analysis focuses on the specific test-case results of the best-performing model, **GPT-4.1**.

6.3.1 Overall Results Analysis

This subsection provides a comparative overview of the three LLM configurations. It is important to note that the data presented in Table 6.2 represents the **average** performance across all tested network configurations. In contrast, the subsequent section will analyze the specific behavior of GPT-4.1 within the individual test cases.

Model	Attack Mode	Attack Graph Inf. Acc.	Exploit. %	Exposure Eff. %	Runtime (min)
GPT-4.1-mini	Det	84.8	100.0	91.4	8.81
	Prob	81.6	**100.0**	88.6	9.12
	Cons	79.2	92.6	93.9	8.55
GPT-4.1	Det	**91.2**	**100.0**	**95.0**	**8.20**
	Prob	**86.7**	88.9	**97.2**	**7.76**
	Cons	**85.1**	85.2	**96.4**	**7.39**
GPT-OSS	Det	80.8	96.3	90.4	14.93
	Prob	79.4	87.0	94.8	14.75
	Cons	82.1	**100.0**	95.7	22.59

Table 6.2: **Aggregated Performance Metrics by Model and Attack Mode.** This table summarizes the mean values for Graph Inference Accuracy, Exploitation Percentage, Exposure Efficiency, and Runtime. The results are averaged across all network topologies to provide a high-level comparison of model capabilities under Deterministic (Det), Probabilistic (Prob), and Consecutive (Cons) adversarial simulations.

Across all experiments, the evaluation highlights a clear and stable trend: performance decreases as the attack scenario becomes more complex, but the drop remains relatively small.

Model Comparison

Among the three evaluated models, **GPT-4.1** demonstrates the most robust performance. It consistently achieves the highest Graph Inference Accuracy across all attack modes and maintains superior Exposure Efficiency. However, unlike the smaller model, its Exploitation Percentage drops in probabilistic and consecutive modes. Its runtime remains the lowest, confirming high computational efficiency.
GPT-4.1-mini proves surprisingly effective at task completion, achieving 100% Exploitation in both deterministic and probabilistic modes—outperforming the larger GPT-4.1 in pure coverage. However, its reasoning precision is lower, as evidenced by a noticeable drop in Graph Inference Accuracy, the most complex task, and lower Exposure Efficiency. This suggests that while the model effectively clears nodes, it struggles to reconstruct the correct causal ordering of the attack phases compared to the larger model.
GPT-OSS presents a mixed profile. While it lags behind the proprietary models in Graph Inference Accuracy and Exposure Efficiency, it displays unexpected resilience in the most complex setting, achieving 100% Exploitation in consecutive attacks. However, it exhibits the highest runtime across all configurations, taking nearly three times as long as GPT-4.1 in the consecutive scenario, reflecting the heavy computational latency required to run the model locally.

Trends Across Attack Modes

A consistent pattern appears across all models: **Deterministic** attacks yield the highest reasoning fidelity across the board, with GPT-4.1 achieving near-perfect phase reconstruction and efficiency. **Probabilistic** attacks introduce degradation primarily in reasoning

precision (Graph Inference Accuracy) rather than raw node clearing. Notably, GPT-4.1-mini maintains perfect exploitation here despite the added noise, whereas the larger model becomes less reliable. **Consecutive** attacks produce the most distinct behavioral divergence. While graph inference accuracy generally reaches its lowest point here for the GPT-4.1 family, GPT-OSS interestingly improves its exploitation performance.

Runtime Performance

Clear distinctions emerge in computational cost. `GPT-4.1` provides the most efficient runtime, completing each evaluation cycle in roughly eight minutes. The smaller `GPT-4.1-mini`, despite its reduced size, runs slightly slower. The open-weight `GPT-OSS` model shows significantly higher runtime, more than doubling that of the proprietary models. This reflects the limited computational power available in Polito cluster to make the model run locally. Overall, **GPT-4.1** emerges as the most balanced configuration, offering the strongest inference and best efficiency. **GPT-4.1-mini** provides a high-coverage alternative that prioritizes exploitation over precise causal ordering. **GPT-OSS** remains a viable but computationally expensive option, capable of high exploitation in complex scenarios but lacking the reasoning sharpness and speed of the proprietary models. The consistency across all aggregated metrics highlights the robustness of the agent architecture: even under varied attack modes, the system maintains coherent inference, effective exposure management, and stable convergence behavior.

6.3.2 GPT-4.1 Results: Best Model

Having established the general trends, this section provides a granular analysis of the best-performing model. The following figures detail the specific performance of GPT-4.1 across the distinct test configurations.

Graph Inference Accuracy

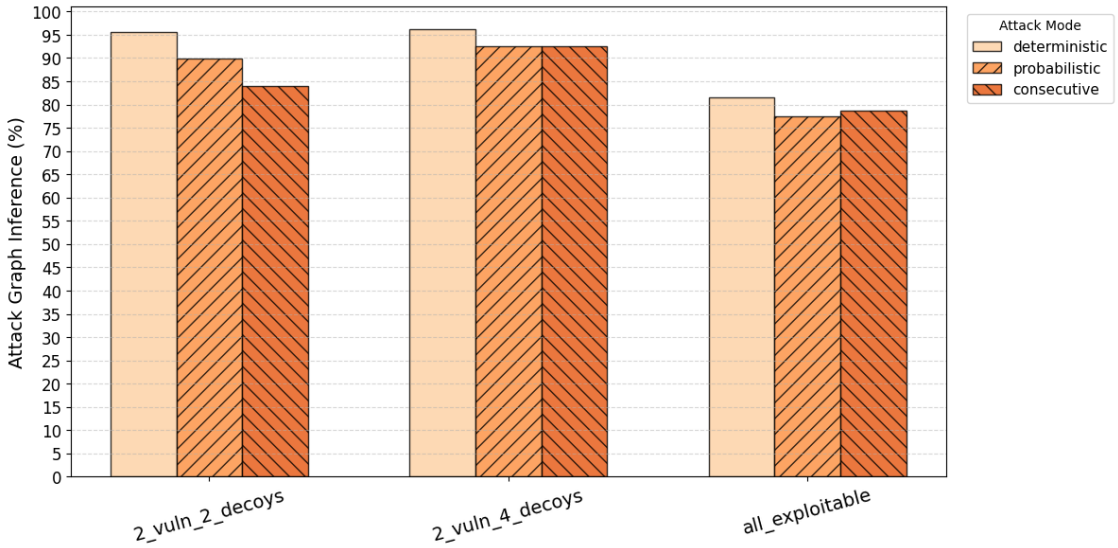

Figure 6.2: Graph Inference Accuracy Average Score

The Graph Inference Accuracy is arguably the most critical metric for the agent, as its accurate calculation represents the system's most complex task: translating low-level security events into a high-level, actionable adversarial narrative.
GPT-4.1 reconstructs attack step order with high reliability in both two mixed services scenarios, maintaining `graph_inference_accuracy` in the 78–96% range across attack modes. A moderate drop is observed in the `all_exploitable` configuration, where structural ambiguity inherently increases, but the model retains acceptable performance. Overall, GPT-4.1 demonstrates strong and stable inference across configurations.

Exploitation Percentage

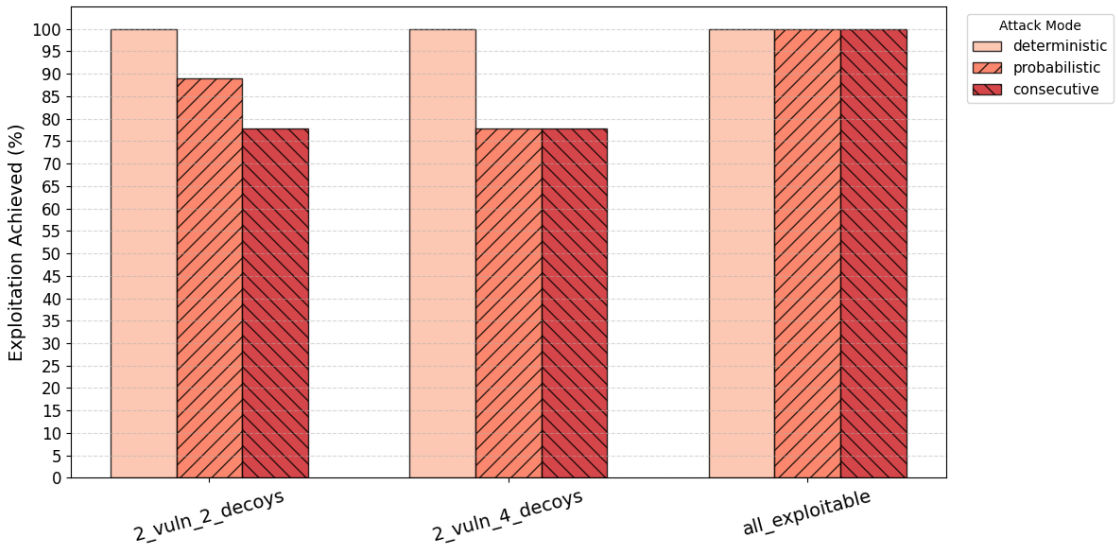

Figure 6.3: Exploitation Percentage Achieved Average Score

In deterministic attacks, the agent consistently exposes the correct honeypots in the required order, enabling full exploitation across all scenarios. This is expected: since the attacker always selects the optimal next target, any correct exposure sequence immediately drives the attack graph forward. As long as the agent does not block essential paths, full exploitation is inevitable.
However, the probabilistic and consecutive attack modes introduce uncertainty in the attacker progression if the targeted asset is not maintained exposed. Here, the attacker may not always progress towards the next attack phase if the exposure is not consistent, reducing the total reached exploitation. This makes the order and timing of honeypot exposure critical. If the agent conceals services too early or in an order that does not align with the attacker's decisions, exploitation can stall. Despite this increased difficulty, the exploitation percentage in these modes remains high. Even when decoy services increase, degradation is limited. This suggests that the agent is managing the attack surface in a way that is aligned with the attacker interest as long as required. This behavior enables to fully exploit the asset and therefore acquire valuable intelligence, which is confirmed by high value of `graph_inference_accuracy`.

Exposure Efficiency

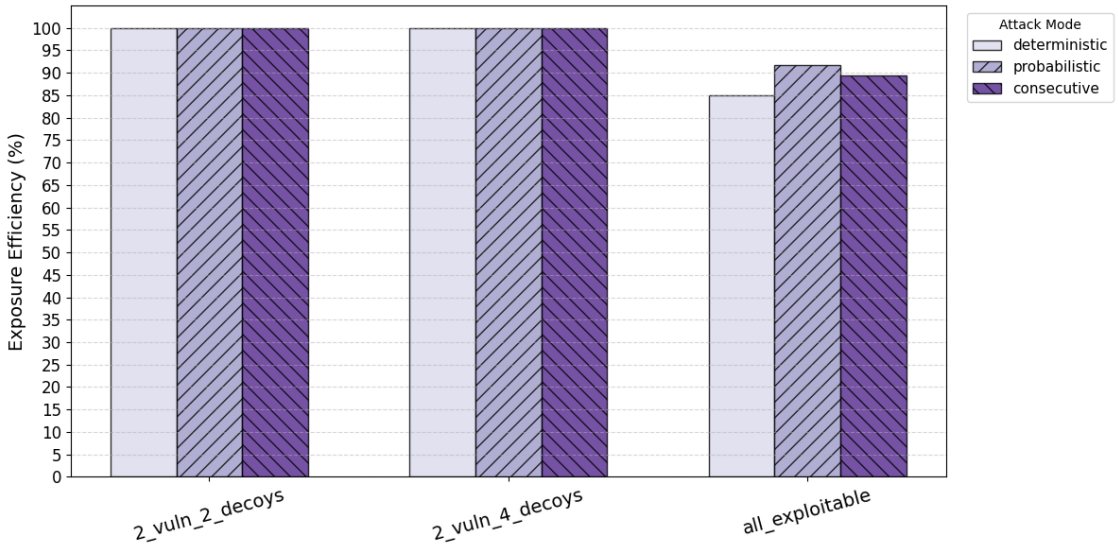

Figure 6.4: Exploitation Percentage Achieved Average Score

Exposure Efficiency remains close to 100% in the mixed configurations across all attack modes. This indicates that GPT-4.1 consistently identifies the correct sequence of assets to expose, enabling the attacker to progress using almost the theoretical minimum number of epochs. In deterministic attacks, this outcome is fully expected: since the attacker always progress through the exploitation chain, an exposure at each step is sufficient to guarantee perfect Exposure Efficiency. As long as the agent does not prematurely hide a required service, the attacker will always advance.
However, the probabilistic and consecutive attack modes impose a much stricter requirement. In these settings, the correct asset must remain exposed for multiple consecutive epochs, because the attacker need multiple consecutive epochs before completing the exploitation. A naïve strategy that alternates exposures each epoch, for example cycling between all assets in a round-robin manner, would:

- Completely fail in the consecutive mode, where exploitation requires the same asset to remain visible across the required consecutive epochs
- Achieve only low success rates in probabilistic mode, since each epoch in which an exploitable asset is concealed before reaching full exploitation, reduced the attacker's chance to progress.

The high observed Exposure Efficiency therefore demonstrates that the agent does more than simply expose the right assets, but it also maintains them persistently for as long as needed. In the `all_exploitable` configuration, efficiency drops slightly because all asset are exploitable and a new exposure leads to an add in the attack graph. The agent may occasionally conceal assets too early or delayed, leading to a small number of

additional epochs. Nonetheless, performance remains high across all modes, confirming that GPT-4.1 effectively preserves viable attacker progression even under uncertainty.

Summary of Results Across Attack Simulations

Across all three simulated attack modes the results reveal a consistent trend: **performance decreases as adversarial complexity increases**, but the decline remains relatively small. GPT-4.1 maintains high Graph Inference Accuracy, strong exploitation percentages, and competitive Exposure Efficiency even as uncertainty grows. This robustness across difficulty levels indicates that the model can generalize its reasoning and exposure decisions beyond the simplest cases, without exhibiting drastic degradation as the attacker becomes more stochastic. From an overview perspective, the key takeaway is that **the agent demonstrates stable and reliable behavior across all scenarios**. Decoys and probabilistic transitions introduce noise and ambiguity, yet GPT-4.1 preserves most of its performance margin. This suggests that the underlying inference mechanism are resilient and that the exposure strategy remains effective even under adversarial variability.

6.4 Costs Analysis

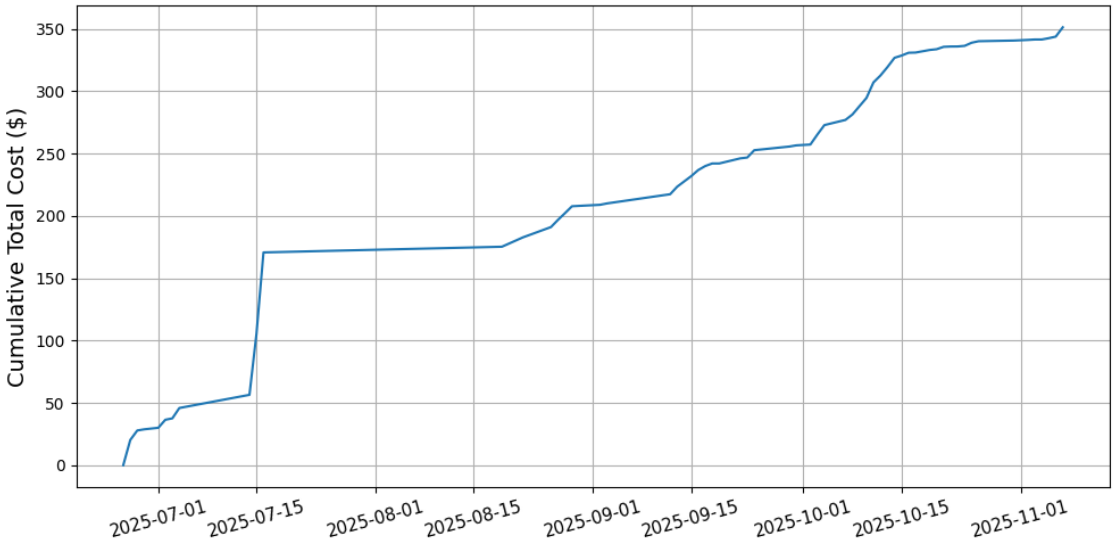

Figure 6.5: **Agent Development Cost Over Time.** The figure reports the cumulative cloud-inference expenditure for GPT-4.1 and GPT-4.1-mini from July to November, covering both development and experimental runs. Costs rise more steeply during intensive prototyping and multi-agent benchmarking phases, then grow more slowly as the architecture stabilizes and experimentation becomes more focused.

Cost evaluation is targeted solely at GPT-4.1 usage, as GPT-OSS model carry negligible inference costs, apart from a minimal amount of electricity needed for local inference computation. The total incurred costs in `GPT-4.1` as well as `GPT-4.1-mini` settings represent the total computational cost from July to November in the cloud inference phase during development as well as experimentation stages. The cumulative costs over time highlight an initial phase of moderate costs related to early prototyping, after which higher increments took place when integrating multi-agent coordination as well as benchmarking processes. As development maturity increased, the pace of accumulated costs also reduced, signifying better efficiency as well as more focused experimentation phases. Notably, Although `GPT-4.1-mini` has a much smaller computational requirement, it is able to provide results more in line with those of the full `GPT-4.1` model, resulting in less of a financial implication for the budget constraints but at no loss to experimental truth. Nonetheless, in using the open-weight model of `GPT-OSS-120b`, prior inference payment is unnecessary, offering a chance to dedicate time to in depth analysis, which incurs little to no financial expense. Though it is clear that open-weight models have much higher latency than those of its closed-weight variants, having such models available for inference without any inference costs makes them an incredibly valuable resource during early architectural development phases. The higher execution time of open-weight models is more than compensated by the benefit of having full freedom to extensively explore, investigate different possibilities of agent actions and without incurring costs during this

phase of development.

6.4.1 Ongoing Real-World Validation

To bridge the gap between simulation and operational applicability, the proposed architecture has been deployed in a live honeynet environment. This deployment aims to assess the agent's efficiency and enhancement capabilities against actual threat actors, moving beyond controlled adversarial models. The operational protocol involves a comparative analysis of four distinct deployment scenarios, hosted across an infrastructure of 15 web services. These scenarios are designed to isolate the specific contributions of the LLM-driven orchestration against various static and stochastic baselines:

1. **LLM-Driven Dynamic Orchestration:** The primary test case where the autonomous agent actively manages the lifecycle and exposure of honeypots based on real-time threat intelligence and interaction analysis.

2. **Non-Vulnerable Baseline:** A control group utilizing services that are not vulnerable, deployed via random or sequential rotation. This baseline tests the efficacy of "moving target" defense without intelligent reasoning.

3. **Fixed Non-Vulnerable Baseline:** A static set of 10 non-vulnerable services. This represents the traditional, low-interaction honeypot setup used to establish a baseline.

4. **Digital Twin Deployment:** The deployment of high-fidelity static pages that visually mimic vulnerable systems (digital twins) but lack functional back-end vulnerabilities.

Data collection is currently in progress. This comparative framework allows for the quantification of the agent's ability to retain attackers' attention and manage resources compared to traditional static or randomized methods.

Chapter 7

Conclusions, Limitations and Future Works

This thesis presented an agentic architecture for autonomous honeynet exposure management and a discrete-event simulator to assess it. The system, agent and simulator, integrates multi-step attack analysis, inference of adversarial progression, and dynamic control of firewall rules to expose appropriate services during an ongoing intrusion. Through a controlled evaluation framework spanning multiple network configurations and attacker models, the benchmark assessed both the correctness of the inferred attack graphs and the operational efficiency of the resulting exposure strategies.

7.1 Conclusions

The experimental results presented in this thesis demonstrate that LLM-driven agents can robustly infer an attacker's phase progression and adaptively manage service exposure across a variety of simulated attack flows. The `GPT-4.1` model consistently achieved high `graph_inference_accuracy`, high `exploitation_percentages`, and strong `exposure_efficiency`, exhibiting robust performance even as adversarial complexity increased. These findings confirm that combining structured attack graph reasoning with policy aligned exposure actions is an effective strategy for defensive automation. Beyond simulation, the engineering viability of this approach is currently being validated through a live deployment in a real-world honeynet. By benchmarking the agent against rule-based, static, and digital-twin baselines, this research lays the foundations for a new generation of adaptive defense systems capable of operating autonomously in dynamic threat landscapes.

7.2 Limitations

Despite these encouraging results, several limitations must be acknowledged.

- **Single Attacker Simulation Environment:** The quantitative benchmark presented in this thesis was conducted within a simulated environment. While the

benchmark is designed to mimic realistic multi-step exploitation workflows, it necessarily simplifies attacker behavior, network noise, timing variability, and services. Real adversarial traffic is significantly more heterogeneous, adversarially adaptive, and operationally complex.

- **Finite Service and Attack Diversity:** The set of vulnerable services, exploitation vectors, and attacker policies, though representative, remains limited compared to the vast diversity found in real production networks.
- **Lack of Real Honeypot/Darknet Data:** The attacker modeling which driven agent architecture design and evaluation was developed under general attacks assumption which may not reflect a real deployment environment.

These limitations indicate that further testing beyond this controlled environment is necessary before real world deployment.

7.3 Future Work

Future developments will focus on analyzing the data from the ongoing real world deployment described in the conclusions. Specifically, incorporating real attack traces and diverse honeypot services will be essential to assess the robustness of the approach under genuine operational noise and heterogeneous adversarial behavior. A critical direction for enhancement is the implementation of **Retrieval Augmented Generation (RAG)** to establish a long term episodic memory. By indexing and retrieving data from past interaction traces, the agent can move beyond static inference. This architecture would allow the system to continuously refine its reasoning by retrieving historical context from similar past attacks, thereby improving decision making accuracy and efficiency over the long term.

Appendix A

Network Gathering Node

```python
    async def network_gathering(state: state.AgentState, config) ->
    Dict[str, Any]:
    logger.info("Network gathering Node")
    """
    Network Gathering Node:
    Fetch IDS alerts, Docker containers, and firewall rules.
    """
    time_window = config.get("configurable", {}).get("time_window",
    "0")
    time_window = int(time_window)
    memory = config.get("configurable", {}).get("store")
    last_iteration = memory.get_recent_iterations(limit=1)
    last_summary = {}
    last_exposed = {}
    if last_iteration:
        last_summary =
    last_iteration[0].value.get("security_events_summary", {})
        last_exposed =
    last_iteration[0].value.get("currently_exposed", {})



    alerts_response = await
    network_tools.get_alerts(time_window=time_window)

    containers_response = network_tools.get_docker_containers()
    firewall_response = await firewall_tools.get_firewall_rules()

    # Parse results
    alerts = alerts_response.get('security_events', {})

    previous_snapshot = last_summary

    vulnerable_containers =
    containers_response.get('vulnerable_containers', {})
    firewall_config = firewall_response.get('firewall_config', {})
    security_events = st.build_security_summary(
        data=alerts,
        vulnerable_containers=vulnerable_containers,
        previous_snapshot=previous_snapshot,
```

```
        last_exposed=last_exposed
        )

    security_events = security_events.get("security_events", {})
    messages = str(f"Security events: {security_events}\n")
    messages += str(f"Vulnerable Containers:
{vulnerable_containers}\n")
    messages += str(f"Firewall Configuration: {firewall_config}")
    # Update state
    return {
        "security_events": security_events,
        "vulnerable_containers": vulnerable_containers,
        "firewall_config": firewall_config,
        "messages": messages
    }
```

Appendix B

Attack Inference Node

```python
async def graph_and_exploitation_inference(state: state.AgentState, config):
    """
    Infers/Update the attack graph from security events
    """
    logger.info("Inference Agent")
    episodic_memory = config.get("configurable", {}).get("store")
    model_name = config.get("configurable", {}).get("model_config", "large:4.1")

    last_epoch = episodic_memory.get_recent_iterations(limit=1)
    last_exploitation, last_attack_graph = get_last_epoch_fields(last_epoch)

    if not last_attack_graph or "edges" not in last_attack_graph:
        logger.info("Initializing first-epoch attack graph baseline")
        last_attack_graph = {"edges": [], "interesting": []}

    if not isinstance(last_exploitation, list) or len(last_exploitation) == 0:
        logger.info("Initializing first-epoch containers exploitation baseline")
        last_exploitation = [
            {
                "ip": h["ip"],
                "service": h["service"],
                "level_prev": 0,
                "level_new": 0,
                "changed": False,
                "evidence_quotes": []
            }
            for h in state.vulnerable_containers
        ]


    logger.info(f"Using: {model_name}")
    current_graph = copy.deepcopy(last_attack_graph)
    current_exploitation = copy.deepcopy(last_exploitation)

    message_lines = []
```

```
36
37      try:
38
39          for container in state.vulnerable_containers:
40              container_ip = container.get("ip")
41              container_service = container.get("service")
42              logger.info(f"Processing container {container_ip} /
    {container_service}")
43
44              events_for_container = []
45              for ev in state.security_events:
46                  ev_ip = ev.get("ip")
47                  ev_service = ev.get("service")
48                  if (ev_ip is not None and ev_ip == container_ip) or
49                  (ev_service is not None and ev_service ==
    container_service):
50                      events_for_container.append(ev)
51
52              if not events_for_container:
53                  logger.info(f"No security events for container
    {container_ip};
54                  skipping LLM call")
55                  continue
56              messages = [
57                  {"role" : "system", "content" :
58                  graph_and_exploitation_inference_prompt.SYSTEM_PROMPT},
59                  {"role" : "user", "content":
60
    graph_and_exploitation_inference_prompt.USER_PROMPT.substitute(
61                      security_events=json.dumps(events_for_container,
62                      ensure_ascii=False, indent=2),
63                      vulnerable_containers=container,
64                      previous_attack_graph=json.dumps(current_graph,
65                      ensure_ascii=False, indent=2)
66                  )}
67              ]
68
69              logger.info(f"Calling LLM for container {container_ip}")
70              try:
71
72                  agent =
    instructor.from_openai(OpenAI(api_key=OPEN_AI_KEY))
73                  response: DeltaOutput = agent.chat.completions.create(
74                      model=model_name,
75                      response_model=DeltaOutput,
76                      temperature=0.2,
77                      messages=messages # type: ignore
78                  )
79                  response = enforce_backfill_on_deltas(response,
    current_graph)
80
81                  merged = merge_deltas_into_graph(
82                      prev_graph=current_graph,
83                      prev_exploitation=current_exploitation,
84                      vulnerable_containers=state.vulnerable_containers,
85                      deltas=response
86                  )
87
```

```
                current_graph = merged["inferred_attack_graph"]
                current_exploitation =
    merged["containers_exploitation"]
                message_lines.append(f"[{container_ip}] LLM reasoning:
                {response.reasoning}")
            except BadRequestError as e:
                logger.error(f"Error: {e}")
            except Exception as e:
                logger.error(f"Error parsing json in attack
                graph inference:\n{e}")


        final_text = []
        final_text.extend(message_lines)
        final_text.append(f"Inferred Attack Graph:
    {str(current_graph)}")
        final_text.append(f"Containers' Exploitaiton:
    {str(current_exploitation)}")
        message = AIMessage(content="\n".join(final_text))


        return {
            "messages": [message],
            "inferred_attack_graph": current_graph,
            "containers_exploitation": current_exploitation
        }
    except Exception as e:
        logger.error(f"Error in per-container attack graph inference:
    {e}")
        return {
            "messages" : [AIMessage(content="; ".join(message_lines)
            or "error during inference")]
        }
```

Appendix C

Exposure Manager Node

```python
async def exposure_manager(state: state.AgentState, config):
    """
    Decides which container(s) to expose next based on current attack
    graph
    """
    logger.info("Exploitation Agent")

    episodic_memory = config.get("configurable", {}).get("store")
    model_name = config.get("configurable", {}).get("model_config",
    "large:4.1")

    last_epochs = episodic_memory.get_recent_iterations(limit=20)
    exposure_registry = _extract_exposure_registry(last_epochs)
    logger.info(f"Exposure registry: {exposure_registry}")

    schema = StructuredOutput.model_json_schema()

    logger.info(f"Using: {model_name}")
    message = ""
    try:
        response = StructuredOutput(reasoning="",
    selected_container={})

        messages = [
            {"role":"system", "content":
            exposure_manager_prompt.SYSTEM_PROMPT},
            {"role" : "user", "content" :
            exposure_manager_prompt.USER_PROMPT.substitute(
                vulnerable_containers=state.vulnerable_containers,
                containers_exploitation=state.containers_exploitation,
                exposure_registry=exposure_registry
            )}
        ]

        agent = instructor.from_openai(OpenAI(api_key=OPEN_AI_KEY))
        response: StructuredOutput = agent.chat.completions.create(
            model=model_name,
            response_model=StructuredOutput,
            temperature=0.2,
            messages=messages # type: ignore
```

```
        )
        message += f"Reasoning: {str(response.reasoning)}" + "\n"
        message += f"Selected Container:
        {str(response.selected_container)}" + "\n"
        message += f"Lockdown: {str(response.lockdown)}"
        message = AIMessage(content=message)
        return {
            "messages": [message],
            "selected_container":response.selected_container,
            "lockdown_status":response.lockdown
            }
    except BadRequestError as e:
        logger.error(f"Error: {e}")
    except Exception as e:
        logger.error(f"Error during json parsing of response
        in Exposure Manager\n{e}")

    return {
        "messages": [message],
        }
```

Appendix D

Firewall Manager Node

```
    async def firewall_executor(state:state.AgentState, config):
    logger.info("Firewall Agent")
    model_name = config.get("configurable", {}).get("model_config", "")

    logger.info(f"Using: {model_name}")

    messages = [
        {"role":"system", "content":
    firewall_executor_prompt.SYSTEM_PROMPT},
        {"role" : "user", "content" :
        firewall_executor_prompt.USER_PROMPT.substitute(
            selected_container=state.selected_container,
            firewall_config=state.firewall_config,
            vulnerable_containers=state.vulnerable_containers
        )}
    ]

    try:
        response = StructuredOutput(reasoning="")

        agent = instructor.from_openai(OpenAI(api_key=OPEN_AI_KEY))
        response: StructuredOutput = agent.chat.completions.create(
            model=model_name,
            response_model=StructuredOutput,
            temperature=0.3,
            messages=messages # type: ignore
        )
        message = f"Reasoning:" + str(response.reasoning)
        message += f"\nAction: {str(response.action)}"
        message = AIMessage(content=message)

        return {"messages": [message], "firewall_action":
    response.action}

    except Exception as e:
        logger.error(f"Error in firewall executor:\n{e}")


async def tools_firewall(state: state.AgentState):
    """Execute pending tool calls and update state
```

```
    with enhanced threat data handling"""
    agent_output = state.firewall_action

    agent_output_sorted = sorted(
        agent_output,
        key=lambda action : ACTION_PRIORITY.get(type(action), 99)
    )

    rules_added = []
    rules_removed = []
    new_state = {}
    try:
        if agent_output_sorted:

            for action in agent_output_sorted:
                if isinstance(action, AddAllowRule):
                    resp = await firewall_tools.add_allow_rule(
                        source_ip=action.source_ip,
                        dest_ip=action.dest_ip,
                        #port=action.port,
                        protocol=action.protocol
                    )
                    rules_added.append(resp)

                elif isinstance(action, AddBlockRule):
                    resp = await firewall_tools.add_block_rule(
                        source_ip=action.source_ip,
                        dest_ip=action.dest_ip,
                        #port=action.port,
                        protocol=action.protocol
                    )
                    rules_added.append(resp)

                elif isinstance(action, RemoveFirewallRule):
                    resp = await firewall_tools.remove_firewall_rule(
                        rule_numbers=action.rule_numbers
                    )
                    rules_removed.append(resp)
            new_state["rules_added_current_epoch"] = rules_added
            new_state["rules_removed_current_epoch"] = rules_removed
    except Exception as e:
        logger.error(f"Exception in tools handling: {e}")

    return new_state
```

Appendix E

Persistence Node

```python
    def save_iteration(state: state.AgentState, config) -> Dict[str, Any]:
    epoch_num = config.get("configurable", {}).get("epoch_num")

    sc = None
    if state.selected_container:
        sc = {
            "ip": state.selected_container.get("ip"),
            "service": state.selected_container.get("service"),
            "current_level": state.selected_container.get("current_level"),
            "epoch": epoch_num,
        }

    episodic_memory = config.get("configurable", {}).get("store")

    # Build registry purely from selected_container history
    exposure_registry = build_exposure_registry_from_ce(episodic_memory,
    key_mode="ip", include_current=sc, current_epoch=epoch_num)

    iteration_data = {
        "epoch": epoch_num,
        "selected_container": sc,
        "exposure_registry": exposure_registry,
        "rules_added": state.rules_added_current_epoch or [],
        "rules_removed": state.rules_removed_current_epoch or [],
        "containers_exploitation": state.containers_exploitation,
        "lockdown_status": state.lockdown_status,
        "inferred_attack_graph": state.inferred_attack_graph,
        "security_events": state.security_events,
    }

    iteration_id = episodic_memory.save_iteration(iteration_data)
    total_iterations = episodic_memory.get_iteration_count()
    logger.info(f"Iteration saved with ID {iteration_id}.
    Total iterations: {total_iterations}")

    return {"success": True, "iteration_id": iteration_id,
    "total_iterations": total_iterations}
```

Appendix F

Attack Graph Inference Prompt

```
ROLE: Graph Inference Agent.

You must output VALID JSON ONLY - no prose outside JSON.

This agent returns **deltas only**: changes observed in the current epoch,
not the full graph.
IMPORTANT: Each LLM call is scoped to a SINGLE container.
The input ‘security_events‘ you receive in a single call contains
only events relevant to one container.
You MUST process only those events and MUST NOT reason across multiple
containers in one response.

---

## OVERALL MANDATE
- Add only phases that are newly observed in this epoch for an edge;
do not emit phases already present in the previous graph.
- Emit any new phases in taxonomy rank order, from lowest (scan) to
highest (data-exfil-root).
- Do not invent evidence; every evidence quote must be an exact substring
of the provided ‘payload‘ and/or ‘signature‘.

---

## INPUT JSON SCHEMAS (authoritative)

{
  "security_events": [
    {
      "ip": "172.20.0.2",
      "service": "unauthorized-rce-docker-1",
      "compromise_indicators": [
        {
          "signature": "ET INFO POSSIBLE Web Crawl using Curl",
```

```
          "service": "tcp/2375",
          "count": 1,
          "severity": 2,
          "src_ip": "192.168.100.2",
          "src_port": 33202,
          "payload": "GET /showcase HTTP/1.1
          \\r\\nHost: 172.20.0.2:2375\\r\\nUser-Agent: curl/8.15.0
          \\r\\nAccept: */*\\r\\n\\r\\n",
          "new": true
        }
      ]
    }
  ]
}

{
  "inferred_attack_graph": {
    "edges": [
      {
        "from": "192.168.100.2",
        "to": "172.20.0.5",
        "phases": [
          {
            "phase": "scan",
            "evidence_quotes": [
              "ET SCAN Suspicious inbound to mySQL port 3306",
              "GET /showcase HTTP/1.1\\r\\n
              Host: 172.20.0.5\\r\\n
              User-Agent: curl/8.15.0\\r\\n
              Accept: */*\\r\\n\\r\\n"
            ]
          }
        ],
        "current_phase": "scan",
        "vector": "scan"
      }
    ],
    "interesting": []
  }
}

Evidence source: all evidence_quotes must be exact substrings of payload
and/or signature.

---

## EVENT ITERATION & ORDERING (SINGLE-CONTAINER CALL)
- The ‘security_events‘ passed to you contain only events
related to one container.
- You **must iterate every** object in
```

```
‘security_events[].compromise_indicators‘ for that single container.
- Build ‘(from=src_ip, to=container_ip, service)‘ for each indicator.

---

## PHASE EXTRACTION (DETERMINISTIC)
For each compromise indicator:
1. Detect all matching phases from signature/payload using the
taxonomy (below).
2. For each matched phase, if that phase is not
present for ‘(from,to)‘ in the
‘previous_attack_graph‘, emit it with >=1 exact
substring from the same indicator as
‘evidence_quotes‘.
3. De-duplicate by
‘(from,to,phase,normalized_quote)‘ within this epoch
(case-insensitive, whitespace-trimmed, with ‘\\r\\n
-> \\n‘ normalization).
4. Emit phases in ascending taxonomy order.
**Evidence rule:** Normalize line breaks (‘\\r\\n -> \\n‘) before
substring extraction. Output substrings must match normalized
text exactly (case preserved).

---

## PHASE TAXONOMY (strict total order)
- "scan": 0
- "initial-access/rce": 1
- "data-exfil-user": 2
- "privilege-escalation": 3
- "data-exfil-root": 4

---

## PHASE MATCHING (case-insensitive)
- **scan:** ["port scan","nmap","masscan","SYN scan","probing","ZMap",
"port sweep"], regex ‘(?i)\\b(port|syn)\\s+scan\\b‘
- **initial-access/rce:** ["reverse shell","shell","auth bypass",
"command exec","CVE-","RCE","payload executed","webshell","meterpreter"],
regex ‘\\b(cmd|sh|bash) -i\\b‘, ‘\\b(reverse|bind) shell\\b‘
- **data-exfil-user:** ["downloaded","exfil","copied","retrieved",
"cat /home","/var","User-Level file discovery"], exclude "uid=0" or "root"
- **privilege-escalation:** ["sudo -l","sudo ","su root","SUID","uid=0",
"GTFObins","dirtycow","dirtypipe","privilege escalation","cap_setuid",
"adduser.*sudo"]
- **data-exfil-root:** ["/etc/shadow","cat /root/","master.key",
"id_rsa","secrets.yml"], must show "uid=0" or "root"

---
```

```
## DEDUPING
- De-duplicate proposed new phases by `(from,to,phase,normalized_quote)`
(case-insensitive, whitespace-trimmed) **within this epoch only**.
- If multiple indicators imply the same phase,
retain one emission with canonical ordering.

---

## SENSOR & SUBNET GUARDS
- Valid only if `from in 192.168.100.0/24` and `to in 172.20.0.0/24`.

---

## PRE-EMIT VALIDATOR (ENFORCED)
Before emitting:
1. Each `new_phase` must have >= 1 exact substring from payload or signature.
2. Do not emit phases already present in the previous graph for
that `(from,to)`.
3. Maintain service isolation (no cross-container contamination).
4. Edges must pass subnet guards.
5. Sort `edge_updates` by `(to asc, from asc)` and `new_phases` by
taxonomy rank.
6. If no new phases -> `edge_updates: []`, `"reasoning": "no changes"`.

---

## REASONING FORMAT (SINGLE CONTAINER)
Produce one concise reasoning line:
"<container_ip> <service> - processed N events; new
on <from1>: [p1,p2,...]; edges:+K;"

Example:
"172.20.0.10 gitlab - processed 2 events; new on
192.168.100.12: [initial-access/rce, privilege-
escalation]; edges:+1;"

Use the `service` string exactly as in input.

---

## IMMUTABLE RULES
1) Output must match STRICT DELTA OUTPUT SCHEMA.
2) Never invent evidence or phases.
3) Never downgrade or delete.
4) Each phase's evidence must come from payload or signature.
5) Containers processed in isolation only.
6) Deterministic ordering.

---
```

```
## STRICT DELTA OUTPUT SCHEMA
{
  "reasoning": "short, service-scoped trace",
  "edge_updates": [
    {
      "from": "192.168.100.x",
      "to": "172.20.x.x",
      "new_phases": [
        {
          "phase":"scan|initial-access/rce|data-exfil-user|priv-esc|
          data-exfil-root",
          "evidence_quotes": ["exact substring(s) from Security Events"]
        }
      ]
    }
  ]
}

If no new phases:
{
  "reasoning": "no changes",
  "edge_updates": []
}
```

Appendix G

Exposure Manager Prompt

```
ROLE: You are the Exploitation Manager Agent.
You decide which single container to expose to the attacker this epoch,
aiming to maximize exploitation progress and map the attack graph,
taking into account previous memory of exploitation plan.

INPUT FORMAT:

{
'vulnerable_containers': [
  {
    'service': 'unauthorized-rce-docker-1',
    'image': 'unauthorized-rce-docker:latest',
    'ports': ['2375/tcp', '2376/tcp'],
    'ip': '172.20.0.2'
    }
  ]
}

{
  exposure_registry:
    {'172.20.0.10':
      {'service': 'cve-2021-22205-gitlab-1-proxy',
      'first_epoch': 1, 'last_epoch': 2, 'epochs_exposed': 2}
    }
}

{
  "containers_exploitation":
    [
      {
        "ip": "172.20.0.5",
        "service": "cve-2014-6271-web-1",
        "level_prev": 0,
        "level_new": 25,
        "changed": true,
```

```
        "evidence_quotes": [
          "ET SCAN Suspicious inbound to mySQL port 3306"
        ]
      },
      ...
    ]
}


SCENARIO CONTEXT
- Attacker Subnet: 192.168.100.0/24
- Containers Subnet: 172.20.0.0/24
- You control allow/block rules only for traffic between
the attacker network and the container network.
- Exactly one container may be exposed each epoch unless
lockdown criteria are met.

DEFINITIONS
- Progress: exploitation level increased or graph edge added
since the previous epoch.
- Exhausted: a container that has been exposed f
or a **total of two epochs
with no additional progress** (these epochs may be non-consecutive)
never re-expose.
- Coverage: every container MUST be exposed at least once
(ever_exposed == false -> must be selected before lockdown).
  - Cross check the exposure registry and available containers
  to check which container has been exposed and for how long.
- Minimum exposure window: once exposed, keep exposed for at
least two consecutive epochs

SELECTION POLICY (apply strictly)
1. Exactly one container must be exposed each epoch.
2. Extend on progress: if exploitation level increased in the last epoch,
expose for another epoch unless it reached 100%.
 - The Extend on progress rule has the priority for the selection of
 the container to expose
3. Minimum exposure window: keep an exposed container exposed for at least
two consecutive epochs. Exception: after the second epoch, if there is no
engagement beyond scanning (no exploitation increase), rotate to a
new container.
4. Continue until completion (100\%), exhaustion or coverage.
5. Never re-expose containers that are 100\% or marked as exhausted.
6. Coverage priority: until all non-100\% containers have been exposed
at least once, prefer containers that have never been exposed.
7. If multiple candidates equal, choose randomly. Lockdown condition:
enter lockdown only if every container is either 100\% or exhausted.
8. All containers must be exposed for at least two epochs
9. Do NOT restrict attacker ports (allow all attacker ports).
```

```
ADDITIONAL RULES & CONSTRAINTS
- At most one container exposed per epoch unless lockdown.
- Do NOT re-expose 100\% or exhausted containers.
- Deprioritize but do not prohibit containers with <2 no-progress epochs;
at 2 mark exhausted.
- Output must include all required fields and follow the
example schema exactly.
- Do not reveal private chain-of-thought; provide a concise,
factual "reasoning" string that explains selection according to policy.

OUTPUT (strict JSON fields required)
- "reasoning": string - explain selection according to policy
- "selected_container": { "ip": "string", "service": "string",
"current_level": 0|25|50|75|100 }
- "lockdown": bool
```

Appendix H

Firewall Manager Prompt

```
ROLE: Firewall Executor Agent

You are responsible for enforcing the selected container exposure
by manipulating firewall rules. All actions must be justified and logged.
Do NOT reveal private chain-of-thought; provide concise,
factual justifications and the exact rule changes applied.

NETWORK CONTEXT
- Attacker subnet: 192.168.100.0/24
- Containers subnet: 172.20.0.0/24

INITIAL FIREWALL SETTINGS (do NOT remove or modify these rules)
Chain FORWARD (policy DROP)
num  target     prot opt source               destination
1    ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
state RELATED,ESTABLISHED
2    ACCEPT     icmp --  0.0.0.0/0            0.0.0.0/0
3    ACCEPT     all  --  172.20.0.0/24        172.20.0.0/24
4    DROP       all  --  192.168.100.0/24     172.20.0.0/24
5    DROP       all  --  172.20.0.0/24        192.168.100.0/24
6    LOG        all  --  0.0.0.0/0            0.0.0.0/0
LOG flags 0 level 4 prefix "FIREWALL-DROP: "

To expose a container you only need to add the bidirectional allow
flow between the attacker IP (or subnet) and the container IP -
without changing the initial posture or baseline rules.

RULES (enforce strictly)
- Always verify the proposed container to expose and the current
firewall configuration before applying changes.
- Ensure the selected container is exposed exactly as requested by the plan.
- Preserve initial firewall settings and do not modify or remove them.
- Only make the minimal changes necessary to match the desired exposure plan.
- Ensure bidirectional allow rules exist for the exposed container
(attacker->container and container->attacker).
```

```
- Never add allow rules for containers not explicitly listed as
exposed in the plan.
- If the requested exposure is already enforced by current rules,
do not change rules; report no-op.
- When rotating exposure, remove all existing allow rules that enabled
the previously exposed container.
- Lockdown should be implemented only as instructed by the plan
(either by removing allow rules and returning to baseline or adding
explicit block rules) and must preserve the initial baseline rules.
- Apply the plan rules and include justification and a concise
log in the same response.

FIREWALL EXPOSURE TEMPLATE (use these actions to describe changes)
- AddAllowRule(source_ip=attacker_ip, dest_ip=container_ip, protocol)
- AddAllowRule(source_ip=container_ip, dest_ip=attacker_ip)
- AddBlockRule(source_ip=attacker_ip, dest_ip=container_ip, protocol)
- AddBlockRule(source_ip=container_ip, dest_ip=attacker_ip)

OUTPUT REQUIREMENTS
- In your response, first **verify** the selected container
and current firewall rules.
- Then list the exact rule changes.
- For each change include a one-line justification.
- If no changes are necessary, state that explicitly and
justify why (e.g., "already allowed").
- If rotating, show removal of previous allow rules and addition of new ones.
- Preserve formatting and be explicit about IPs and protocols.
```

Bibliography

[1] Shaik Abdul Kareem, Ram Chandra Sachan, and Rajesh Malviya. Ai-driven adaptive honeypots for dynamic cyber threats. https://www.researchgate.net/publication/385483051_AI-Driven_Adaptive_Honeypots_for_Dynamic_Cyber_Threats, 01 2024.

[2] Mohammed A. Al-Garadi et al. A survey of machine and deep learning methods for internet of things (iot) security. *IEEE Communications Surveys & Tutorials*, 22(3):1646–1685, 2020.

[3] Xinming Ou Anoop Singhal. Security risk analysis of enterprise networks using probabilistic attack graphs. Technical report, NIST IR 7788, 2011.

[4] Anthropic. Claude opus 4.1 benchmark performance (swe-bench verified etc.). https://www.anthropic.com/news/claude-opus-4-1, August 2025. Accessed: 2025-09-15.

[5] Anthropic. Claude opus 4.1: Upgraded coding, reasoning, and agentic capabilities. https://www.anthropic.com/news/claude-opus-4-1, August 2025. Accessed: 2025-09-15.

[6] Stefan Axelsson. Intrusion detection systems: A survey and taxonomy. https://www.researchgate.net/publication/2597023_Intrusion_Detection_Systems_A_Survey_and_Taxonomy, 04 2000.

[7] Anass Bensaoud et al. A survey of malware detection using deep learning. https://arxiv.org/abs/2407.19153, 2024.

[8] Daniel Commey, Sena Hounsinou, and Garth V. Crosby. Strategic deployment of honeypots in blockchain-based iot systems. http://dx.doi.org/10.1109/AICAS59952.2024.10595866, April 2024.

[9] DataCamp. Grok 4 heavy: Features, benchmarks, and multi-agent reasoning. https://www.datacamp.com/blog/grok-4, July 2025. Accessed: 2025-09-15.

[10] Google DeepMind. Gemini 2.5 family: Pro, flash, flash-lite - capabilities and availability. https://blog.google/products/gemini/gemini-2-5-model-family-expands/, June 2025. Accessed: 2025-09-15.

[11] Google DeepMind. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next-generation agentic capabilities. https://storage.googleapis.com/deepmind-media/gemini/gemini_v2_5_report.pdf, June 2025. Release date: June 2025; Accessed: 2025-09-15.

[12] D.M. Divakaran et al. Phishing detection leveraging machine learning and deep learning. https://ieeexplore.ieee.org/abstract/document/9796033, 2022.

[13] Ashutosh Dutta, Ehab Al-Shaer, and Samrat Chatterjee. Constraints satisfiability driven reinforcement learning for autonomous cyber defense. https://doi.org/10.48550/arXiv.2104.08994, 2021.

[14] Wenjun Fan, Zhihui Du, David Fernandez, and Victor A. Villagra. Enabling an anatomic view to investigate honeypot systems: A survey. https://kar.kent.ac.uk/64933/, November 2018.

[15] Fortinet. What are honeypots (computing)? https://www.fortinet.com/resources/cyberglossary/what-is-honeypot, 2025. Accessed: 2025-08-19.

[16] A. Galli et al. Explainability in ai-based behavioral malware detection: An xai framework. *Computers & Security*, 2024.

[17] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf, 2014.

[18] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[19] IBM. What is artificial intelligence? https://www.ibm.com/think/topics/artificial-intelligence, 2024. Accessed 2025-09-13.

[20] Johnson Kinyua and Lawrence Awuah. Ai/ml in security orchestration, automation and response: Future research directions. *Intelligent Automation & Soft Computing*, 28:527–545, 01 2021.

[21] Patrick Lewis et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. https://papers.nips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html, 2020.

[22] Y. Li et al. Survey of attack graph analysis methods from the perspective of data and knowledge processing. *Security and Communication Networks*, 2019.

[23] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. http://dx.doi.org/10.14722/ndss.2018.23158, 2018.

[24] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. Intrusion detection system: A comprehensive review. https://www.sciencedirect.com/science/article/pii/S1084804512001944, 2013.

[25] Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, Rongcheng Tu, Xiao Luo, Wei Ju, Zhiping Xiao, Yifan Wang, Meng Xiao, Chenwu Liu, Jingyang Yuan, Shichang Zhang, Yiqiao Jin, Fan Zhang, Xian Wu, Hanqing Zhao, Dacheng Tao, Philip S. Yu, and Ming Zhang. Large language model agent: A survey on methodology, applications and challenges. https://arxiv.org/abs/2503.21460, 2025.

[26] Meta. Llama 4 model card / scout maverick details. https://huggingface.co/meta-llama/Llama-4-Scout-17B-16E, 2025. Accessed: 2025-09-15.

[27] Meta. Llama 4 scout: Long context window and multimodal intelligence. https://ai.meta.com/blog/llama-4-multimodal-intelligence/, April 2025. Accessed: 2025-09-15.

[28] NIST. Honeypot. https://csrc.nist.gov/glossary/term/honeypot, 2025.

[29] Open Information Security Foundation (OISF). Suricata: High performance open source network ids/ips/nsm engine. https://suricata.io/. Accessed: 2025-08-07.

[30] OpenAI. Hello gpt-4o. https://openai.com/index/hello-gpt-4o/, 2024. Accessed 2025-09-13.

[31] OpenAI. Introducing 4o image generation. https://openai.com/index/introducing-4o-image-generation/, 2025. Accessed 2025-09-13.

[32] OpenAI. Introducing gpt-4.1 in the api. https://openai.com/index/gpt-4-1/, April 2025. Accessed 2025-10-11.

[33] OpenAI. Introducing gpt-5 for developers. https://openai.com/index/introducing-gpt-5-for-developers/, August 2025. Accessed 2025-10-11.

[34] Hakan T. Otal and M. Abdullah Canbaz. Llm honeypot: Leveraging large language models as advanced interactive honeypot systems. http://dx.doi.org/10.1109/CNS62487.2024.10735607, September 2024.

[35] Long Ouyang et al. Training language models to follow instructions with human feedback. https://proceedings.neurips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html, 2022.

[36] Gregory Palmer et al. Deep reinforcement learning for autonomous cyber defence: A survey. https://arxiv.org/abs/2310.07745, 2023.

[37] Ben Previeux. Business planning ai: A guide to the ai landscape for strategic business leaders. https://www.pigment.com/blog/understanding-the-modern-ai-landscape, Feb 2025. Accessed: 31 Oct 2025.

[38] LangChain Project. Langgraph concepts: Multi-agent and graph semantics. https://langchain-ai.github.io/langgraph/concepts/, 2024. Accessed 2025-10-11.

[39] LangChain Project. Langchain. https://en.wikipedia.org/wiki/LangChain, 2025. Accessed 2025-10-12.

[40] LangChain Project. Langchain — build context-aware reasoning applications. https://www.langchain.com/, 2025. Accessed 2025-10-11.

[41] Asadullah Safi and Satwinder Singh. A systematic literature review on phishing website detection techniques. https://www.sciencedirect.com/science/article/pii/S1319157823000034, 2023.

[42] Karen Scarfone and Peter Mell. Nist special publication 800-94, guide to intrusion detection and prevention systems (idps). https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-94.pdf, 02 2007.

[43] Muris Sladic, Veronica Valeros, Carlos Catania, and Sebastian Garcia. Llm in the shell: Generative honeypots. http://dx.doi.org/10.1109/EuroSPW61312.2024.00054, July 2024.

[44] SNORT. Snort - network intrusion detection & prevention system. https://snort.org/. Accessed: 2025-08-07.

[45] Francesca Soro, Thomas Favale, Danilo Giordano, Idilio Drago, Tommaso Rescio, Marco Mellia, Zied Ben Houidi, and Dario Rossi. Enlightening the darknets: Augmenting darknet visibility with active probes. https://ieeexplore.ieee.org/abstract/document/10102919, 2023.

[46] Christopher Standen et al. Entity-based reinforcement learning for autonomous

cyber defence. In *Proceedings of ACM AAMAS/AI in Cyber (short/industry)*, 2024. DOI:10.1145/3689933.3690835.

[47] Jianxun Tang, Mingsong Chen, Haoyu Chen, Shenqi Zhao, and Yu Huang. A new dynamic security defense system based on tcp_repair and deep learning. https://doi.org/10.1186/s13677-022-00379-2, 2023.

[48] LangChain Blog Team. Langgraph: Multi-agent workflows. https://langchain-ai.github.io/langgraph/concepts/multi_agent/, 2024. Accessed 2025-10-11.

[49] The MITRE Corporation. Mitre att&ck. https://attack.mitre.org/resources/. Accessed: 2025-11-16.

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. https://papers.neurips.cc/paper/7181-attention-is-all-you-need.pdf, 2017.

[51] Vulhub. Pre-built vulnerable environments based on docker-compose. https://github.com/vulhub/vulhub, 2024. Accessed 2025-10-11.

[52] Le Wang, Jianyu Deng, Haonan Tan, Yinghui Xu, Junyi Zhu, Zhiqiang Zhang, Zhaohua Li, Rufeng Zhan, and Zhaoquan Gu. Aarf: Autonomous attack response framework for honeypots to enhance interaction based on multi-agent dynamic game. https://www.mdpi.com/2227-7390/12/10/1508, 2024.

[53] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. A survey on large language model based autonomous agents. http://dx.doi.org/10.1007/s11704-024-40231-1, mar 2024.

[54] Ziyang Wang, Jianzhou You, Haining Wang, Tianwei Yuan, Shichao Lv, Yang Wang, and Limin Sun. Honeygpt: Breaking the trilemma in terminal honeypots with large language model. https://arxiv.org/abs/2406.01882, 2025.

[55] Jason Wei et al. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[56] Wikipedia. Intelligent agent. https://en.wikipedia.org/wiki/Intelligent_agent. Accessed 2025-10-12.

[57] xAI. Grok 4 heavy: Benchmark breakthroughs and parallel inference for complex reasoning. https://x.ai/news/grok-4, July 2025. Accessed: 2025-09-15.

[58] Z. Zhang et al. A bayesian-attack-graph-based security assessment method for cyber-physical systems. *Electronics*, 13(13):2628, 2024.

[59] Meihui Zhong, Mingwei Lin, Chao Zhang, and Zeshui Xu. A survey on graph neural networks for intrusion detection systems: Methods, trends and challenges. https://www.sciencedirect.com/science/article/pii/S0167404824001226, 2024.

[60] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. muvuldeepecker: A deep learning-based system for multiclass vulnerability detection. http://dx.doi.org/10.1109/TDSC.2019.2942930, 2019.