Alma Mater Studiorum · Università di Bologna

Scuola di Ingegneria e Architettura
Dipartimento di Informatica · Scienza e Ingegneria · DISI
Corso di Laurea Magistrale in Ingegneria Informatica

Cyber Deception through LLM-Driven Infrastructure as Code: A Fine-Tuned Pipeline for Context-Aware Honeypot Generation

Relatore:
Prof. Michele Colajanni

Correlatore:
Ing. Silvio Russo

Presentata da:
Riccardo Benedetti

Anno Accademico 2025/2026

Abstract

Cyberattacks against modern organizations are growing in both frequency and sophistication, exposing the limitations of traditional reactive defense strategies that rely on perimeter security devices such as firewalls and intrusion detection systems. Cyber deception has emerged as a promising proactive approach that shifts the attacker-defender asymmetry by introducing decoys and false targets into the network, forcing adversaries to expend significant resources distinguishing real assets from fake ones. Honeypots, systems designed to appear as legitimate targets while observing and recording attacker behavior, are at the core of this paradigm. However, deploying realistic and effective honeypots remains a labor-intensive process that requires deep knowledge of both the target infrastructure and the specific attack vectors being defended against. Manual configuration does not scale across diverse enterprise environments, and static deployments are vulnerable to fingerprinting by sophisticated adversaries.

This thesis presents an automated deception pipeline that dynamically generates and deploys cloud-based honeypots tailored to specific threat scenarios. The system receives as input a threat model expressed in the Meta Attack Language (MAL), a domain-specific language designed to describe attack scenarios in terms of assets, attack steps, and their causal dependencies. The pipeline processes this input through six sequential stages. A MAL Parser constructs a directed attack graph from the input file, extracting attack steps and their relationships. A Threat Classifier maps each extracted step to the corresponding MITRE ATT&CK Enterprise technique, assigning risk weights based on the kill chain phase. A Strategy Planner analyzes the graph structure using Betweenness Centrality to identify the techniques that act as bottlenecks in the attack progression, selecting the optimal placement for decoys under explicit resource constraints (a coverage threshold of 80% and a maximum of five deployed virtual machines). An LLM Generator, powered by a fine-tuned Llama 3.1 8B Instruct model, produces Terraform variable definitions and embedded Bash startup scripts for each selected technique. A Code Validator verifies the generated configurations through the Terraform CLI and, if errors

are detected, sends them back to the LLM for correction in an iterative feedback loop. Finally, a GCP Deployer provisions the validated configurations as fully functional honeypot instances on Google Cloud Platform.

A key design decision was to execute the entire pipeline locally on consumer-grade hardware, avoiding reliance on external LLM providers to protect the confidentiality of sensitive threat intelligence data. The model selection process evaluated three candidates in the seven-to-nine billion parameter range: Mistral-7B, Deepseek-Coder-7B, and Llama 3.1 8B Instruct. Preliminary generation trials revealed that Mistral-7B consistently introduced conversational preambles and natural language explanations within the generated code, breaking automated parsing, while Deepseek-Coder-7B produced structurally incorrect HCL syntax due to its training bias toward general-purpose programming languages. Llama 3.1 8B Instruct proved to be the most reliable candidate, consistently producing clean, structured output that adhered to the requested format. The model was fine-tuned using Quantized Low-Rank Adaptation (QLoRA) on a single NVIDIA A100 GPU in approximately 11.5 minutes, achieving a final training loss of 0.045 and a validation loss of 0.061 over three epochs.

The fine-tuning dataset was constructed through a three-phase process. An initial set of 115 high-quality samples was manually curated from open-source honeypot repositories, with each configuration mapped to specific MITRE ATT&CK techniques. This seed dataset was expanded to 800 samples through synthetic augmentation using Google Gemini with few-shot prompting, followed by a rigorous cleaning and validation pipeline that ensured 100% structural conformance. The final dataset covers 42 unique MITRE techniques distributed across all major kill chain phases and eight service types, including SSH, RDP, web applications, DNS, FTP, databases, SCADA/ICS, and IoT devices. Rather than generating complete Terraform configurations, the LLM was trained to produce only variable definitions (`.tfvars`) that are injected into a pre-validated base template, significantly reducing the error surface while allowing the model to focus on the semantic content of the honeypot (machine sizing, service selection, and startup script generation).

The prompt engineering strategy was inspired by the SPADE framework for adaptive cyber deception. Each generation prompt includes structured threat context (the MITRE technique name, tactical classification, and a natural-language description of the adversary's goals) alongside explicit output instruc-

tions that constrain the model to produce pure Bash code within a maximum length. For multi-VM deployments, separate specialization prompts are used for each technique to prevent the model from mixing configurations across different attack vectors.

Experimental validation was conducted on five representative attack scenarios: Credential Access, Lateral Movement, DNS Exfiltration, System and Network Discovery, and Persistence. The pipeline achieved a 100% deployment success rate across all test cases, with every generated Terraform configuration passing the `terraform validate` step. The deployed honeypots demonstrated effective detection capabilities: an SSH honeypot targeting password-guessing attacks (T1110.001) captured real brute-force attempts from five distinct IP addresses within 30 minutes of deployment, in addition to the controlled Metasploit test. Pipeline execution times ranged from 65 seconds for single-VM deployments to approximately 337 seconds for four-VM scenarios, with the LLM generation stage representing the primary computational bottleneck due to the auto-regressive nature of token generation and the quadratic scaling of the transformer's self-attention mechanism with increasing context length.

The system addresses several open challenges in automated cyber deception. The modular architecture enables future extensibility: adapting the pipeline to a different cloud provider requires only a new deployer module, supporting a different input format requires only a new parser, and upgrading the language model requires only retraining the generation component. The work also identifies limitations inherent to using an 8-billion parameter model for infrastructure code generation, including occasional hallucinations and the generation of structurally vulnerable configurations. These risks are mitigated through network isolation using Virtual Private Clouds and the iterative validation feedback loop. Future directions include integrating real-time telemetry from Endpoint Detection and Response (EDR) agents to trigger dynamic honeypot generation based on live attack behavior, experimenting with larger language models to reduce hallucination rates, and expanding the training dataset with additional manually curated samples to further improve generation reliability.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, the field of cybersecurity has undergone significant changes. Cyberattacks against modern organizations are becoming more frequent and increasingly complex, ranging from automated malware operations to state-sponsored actors' targeted Advanced Persistent Threats (APTs) [7]. Conventional defensive strategies operate in a primarily reactive manner, identifying and addressing threats only after an attack has begun. They are mostly based on perimeter security devices, such as firewalls and intrusion detection systems (IDS). This paradigm does not work well against adversaries who are constantly changing their Tactics, Techniques, and Procedures (TTPs) to avoid being detected [21]. There has never been a greater gap between the efforts of attackers and defenders; attackers only need to identify one vulnerability to succeed, while defenders must guard every potential vulnerability. To address this problem, the cybersecurity community has gradually transitioned to a proactive defense paradigm [33]. One of the most promising approaches that has emerged is "Cyber Deception", which fundamentally shifts the attacker-defender relationship by misleading, confusing, and exposing attackers through specifically tailored decoys and false targets, instead of attempting to block malicious activities. At the core of this approach is the honeypot, a system designed to appear as a legitimate target with the actual purpose of observing and detecting malicious attackers, allowing security teams to analyze their behavior in a controlled environment while learning new attack vectors and techniques before they are used on actual systems. However, creating realistic and effective decoys remains a significant challenge, requiring knowledge of both the target simulated infrastructure and the attack vectors against

which they are being defended. A manual approach to configuration is time-consuming and difficult to implement across enterprise environments, in which the infrastructure is diverse and non-standardized. The effectiveness of cyber deception heavily depends on its theoretical foundations. Game-theoretic models capture the interactions between attackers and defenders, allowing for the analysis of optimal strategies under conditions of incomplete information. On a cognitive level, deception operates by disrupting and disturbing an attacker's decision-making approach, known in military strategy as the OODA Loop (Observe, Orient, Decide, Act). By using false signals during the reconnaissance phase, defenders cause the adversary to build an incorrect model of the target environment, leading to suboptimal decisions and detectable errors. In Information Asymmetry Reversal the introduction of decoys forces the attackers to expend significant resources to distinguishing fake assets from real ones, shifting advantage from the offensive to the defensive side. Early deception systems relied on static configurations that remained unvaried over time, becoming vulnerable to sophisticated attacks that fingerprint decoys through systematic probing, response analysis, and network context evaluation. This led the field to shift to a dynamic deception approach, where honeypots continuously mutate their configurations, open ports, and responses to invalidate any reconnaissance data previously gathered by attackers, maintaining the decoy's credibility over time, at the cost of a higher resource consumption and greater complexity in the management. The honeypot ecosystem has changed considerably over time, introducing solutions with different levels of complexity and realism: Low-interaction honeypots emulate limited services to detect automated scans with minimal resource overhead, but their simple emulation makes them susceptible to detection by experienced adversaries who probe beyond basic services. Medium- and high-interaction systems provide increasingly realistic environments that capture detailed attacker behavior, even if they carry the operational risk of becoming entry points into the production network if not properly contained. Specialized implementations now target a wide variety of protocols, from SSH and web applications to industrial control systems (ICS/SCADA), and multi-honeypot platforms can combine several tools into a single deployment for broader coverage. Recent advancements in the field of Large Language Models have unveiled new opportunities for automating the creation of decoys. Tools such as Terraform enable the definition of computing infrastructure through declarative configuration files, facilitat-

ing reproducible and version-controlled honeypot deployments. Tailoring a large language model for this field involves unique challenges: the declarative characteristics of Infrastructure as Code (IaC) languages, the necessity for rigorous validation, and the potential for creating insecure setups all require consideration. Studies on AI-Generated code have proved that models that are trained on publicly available repositories are prone to reproduce insecure code patterns. Prompt engineering techniques like role-based prompting, chain-of-thought reasoning and structured output specifications help guide the model towards desired appropriate outputs, mitigating behaviors of hallucinations and incorrect resource configurations. Parameter-efficient fine-tuning methods such as QLoRA make it possible to adapt large models on limited hardware, while synthetic data generation addresses the scarcity of domain-specific training examples, a process that requires careful manual intervention to avoid propagating errors into the fine-tuned model.

Our approach attempts to overcome these limitations through an automated deception system that dynamically generates and deploys honeypots based on the analysis of a threat scenario, following a proactive decoy selection approach [42]. The system receives a threat model represented in the Meta Attack Language (MAL) [40], a domain-specific language designed to characterize cyberattacks and their steps and connections. The model is then classified with a dedicated module that maps the attacks described using the MITRE ATT&CK Enterprise framework, identifies the techniques, and calculates their potential impact on an infrastructure with risk scores based on their severity. A graph is also created representing the relationships between the techniques. A Strategy Planner module analyzes the graph structure and computes the Betweenness Centrality [42] to determine which nodes are critical, indicating where the decoy placement would be most effective at deceiving a malicious actor with hardware resources as a constraint. The central part of this architecture is a fine-tuned Large Language Model (Llama-3.1-8b-Instruct) [12] that generates Context-Aware Infrastructure as Code (Terraform) configurations, built specifically to counter the techniques specified in the threat model. To fine-tune this dataset, a custom dataset composed of 800 honeypot samples was built from the ground up using publicly available honeypots in the Awesome Honeypots [26] repository and enlarged using synthetic data augmentation techniques. The configuration is validated before the deployment on the Google Cloud Platform, resulting in fully functional hon-

eypots. The first chapter of this thesis presents the State Of the Art, where we will discuss current cyber deception techniques, honeypot technologies, and the application of machine learning and Large Language Models in cybersecurity. The second chapter details the methodology, detailing the architecture of our solution and the workflow through its six different stages: MAL file parsing, threat classification, strategy planning, Terraform generation, code validation, and deployment. The third chapter presents the creation process and the composition of the training dataset. The fourth chapter discusses the implementation, presenting the classifier components, including the MAL parser and attack graph analysis, LLM-based generation module, and deployment infrastructure. The fifth chapter covers validation, demonstrating the effectiveness of our approach through experimental evaluation and analysis of generated outputs, including system performance metrics. In the last chapter, we discuss our findings and analyze the implications and limitations of our approach.

Chapter 2

Background and Related Works

In this chapter, we review the current literature and techniques of the technologies used in the design of our deception solution. We cover the foundational principles of cyber deception, the taxonomy of honeypot technologies, the role of the MITRE ATT&CK framework in threat classification, Meta Attack Language for threat modeling, and the state of Large Language Models in cybersecurity and Infrastructure as Code generation.

2.1 Cyber Deception

Cyber Deception can be defined as an ensemble of planned actions that aim to deceive, confuse, and mislead attackers, causing them to take (or not take) specific actions that help computer security defense [13] [1]. Unlike traditional security countermeasures that rely on denying access or reactively blocking intrusions, this approach aims to manipulate an adversary's perception and decision-making process by introducing a "cognitive bias" into the attacker's observation, leading them to make incorrect or suboptimal choices and moves. [39] [33]. This strategy modifies the conventional cybersecurity asymmetry, where the defenders must protect every possible point of intrusion, and the attackers only need one vulnerability to succeed in malicious intent in favor of the defender, who now has the ability to make the attacker operate in an illusory environment while holding the true knowledge of its network [43]. In the field of Cyber Security the idea of deception is a relatively new concept, only emerging as a new discipline between the last years of the 1990s and the first years of the 2000s with the introduction of early honeypot

implementations. [37] Since then, it has gained attention and evolved into an important field, necessitated by the limitations of the conventional security approaches. Traditional approaches are often designed to be predictable and consistent, providing attackers with reliable feedback on refining and developing new techniques and exploits. The last years have also seen a rise in the presence of Advanced Persistent Threats (APTs), which employ a multi-stage attack taxonomy to evade signature-based detection, making the unpredictable and dynamic nature of the Proactive Deception method even more indispensable. [39] [7]

2.1.1 Taxonomy Of Deception Mechanisms

The deception methodology uses different techniques to protect against cyberthreats. [7] To create a strong defense plan, it's important to know how these techniques are classified. In the 2019 study "A Game-theoretic Taxonomy and Survey of Defensive Deception for Cybersecurity and Privacy, " [27], the authors use game theory to classify these techniques. They discuss **Mimesis**, which means creating false beliefs, such as fake databases, and **Crypsis**, which means hiding true beliefs, such as important servers. **Honey-X Systems** are the most common type. These include **Honeypots** (fake systems to catch hackers), **Honeynets** (networks of honeypots to monitor attacks), and **Honeytokens** (fake credentials and API keys). **Honeyfiles** work similarly by tracking unauthorized access. **Moving Target Defense (MTD)** is often mentioned with deception. This makes systems more complex and random using methods such as Address Space Layout Randomization (ASLR) or changing network layouts. MTD helps confuse attackers and works with **Decoys and Diversions**, such as fake entry points to mislead and redirect them from real targets. **Data Perturbation** changes system outputs or signals, like HTTP response headers, to confuse attackers' tools [27]. **Obfuscation** makes it harder for attackers by hiding the logic or structure of resources. This may involve randomizing variable names in scripts or utilizing encrypted payloads, demanding additional time and resources from attackers, thereby aiding in their detection [1].

2.1.2 Dimensions of Cyber Deception

Effective deception strategies have been implemented across various infrastructure-stack dimensions to ensure comprehensive protection against a wide range of attack vectors [7]. Within the **Network Dimension**, defenders manipulate network topologies and traffic patterns, such as through virtual IP shuffling, to detect scanning activities and lateral movement. In the **System Dimension**, high-interaction honeypots simulate kernels and hardware, extending their capabilities to include IoT and SCADA firmware. The **Data Dimension** involves the injection of false information, such as credentials or database records, which act as high-fidelity traps. Finally, the **Software/Application Dimension** incorporates deception directly into the application logic through methods such as **honeypatches** [16], ensuring that even if network defenses are circumvented, the application layer remains a threat to adversaries.

2.1.3 Strategic Phases of Deception

Deception is a continuous plan used throughout an attack. This process is typically divided into four stages [1, 7]. First, **Prevention** attempts to stop the attacks before they occur. Knowledge of advanced deception can cause attackers to think twice. Second, **Detection** finds intruders by noticing any interaction with a decoy that is always suspicious and can be detected. Third, **Reaction** involves taking action, such as slowing down connections, creating endless loops, or moving attackers to isolated areas. Finally, **Forensic Analysis** uses detailed traps to gather information on attacker methods, including new exploits and tools for later study.

2.1.4 Theoretical Foundations of Cyber Deception

Modern cyber deception is based on theories that model how attackers and defenders interact beyond hiding information.

Information Asymmetry

Traditional security models have a problem: defenders must fix all weaknesses, but attackers only need to find one to compromise the system. Cyber deception changes this **Information Asymmetry**. By adding fake targets to the network, defenders can confuse the environment: Attackers with limited

information cannot distinguish between real and fake infrastructure. This exponentially increases their effort to verify the targets, forcing them to spend more time on resources checking, raising the operational cost of the attack and making them more likely to commit a detectable error, giving the defenders an advantage [1].

Cognitive Manipulation with the OODA Loop

The OODA Loop **OODA Loop** (Observe, Orient, Decide, Act) is a cyclic process framework utilized in military strategy. Deception affects the first two steps: **Observe** and **Orient**. By providing false information to attackers during the reconnaissance phase, defenders can disrupt their understanding, leading to poor decisions and actions. Consequently, during the Orient phase, the attacker has formed an inaccurate model of the target infrastructure, leading to errors in the decision-making process. Continuous deception makes attackers restart the loop, slowing it down and giving defenders the time to act [39].

Game Theory Models

Deception is rooted in game theory, which captures the strategic interactions, conflicts of interest, and information asymmetry between rational decision-makers [45]. A game-theoretic approach evaluates optimal strategies by anticipating adversary responses. In cyber deception, we can consider the process in which a defender deploys deceptive strategies to increase uncertainty and an attacker chooses their actions as a game. Different types of games are employed depending on the specific characteristics of the deceptive mechanism and the attack scenario.

- **Stackelberg Games:** model sequential interactions where one player commits to a strategy and the other follows by observing the strategy to make their move. In cybersecurity, the defender acts as the leader, distributing defensive resources, and the attacker follows by recognizing and deciding how to attack. This scenario is widely used in Moving Target Defense and optimal honeypot allocation [45].

- **Signaling Games:** A signaling game is a sequential Bayesian game of incomplete information where one player (the sender) possesses private

information and sends a signal to another player (the receiver). This is particularly suitable for modeling *honey-X* mechanisms (like honeypots or honeytokens), where the defender controls systems with hidden true types (real or decoy) and emits signals (e.g., system responses) that the attacker observes to decide whether to attack [27].

- **Stochastic and Partially Observable Games:** To capture the dynamic, multi-stage evolution of complex attacks (like APTs), Partially Observable Markov Decision Processes (POMDP) and stochastic games are used. These models address imperfect monitoring, allowing a defender to optimize long-term strategies, such as when to engage or eject an attacker from the network, rather than making isolated, single-step decisions [45].

Reinforcement Learning

Reinforcement Learning (RL) enables deception systems to autonomously learn optimal defense strategies by continuously interacting with the environment. [7]. Rather than relying on hard-coded rules, the deception landscape is mathematically structured as a Markov Decision Process (MDP) or a Partially Observable Markov Decision Process (POMDP) [45]. In this environment, RL agents (using algorithms such as Q-learning or deep Q-networks) observe attacker behavior and dynamically determine actions. Through trial and error, the agent learns a behavior that maximizes a reward function, which is structured to penalize successful attacker progression or reward detection.

2.1.5 Advantages and Challenges

Cyber deception offers several benefits that can be used by defensive infrastructure to identify hidden threats and gather intelligence that conventional perimeter defenses might lack [7,43]. Generally, because decoy systems hold no legitimate business value, any interaction with them represents a nearly guaranteed true-positive alert, drastically reducing the alerting work commonly associated with traditional Intrusion Detection Systems (IDS).

Despite these advantages, deploying deception architectures introduces substantial challenges that must be carefully managed [18]. The primary difficulty lies in maintaining believability: defenders must design fake systems

that are sufficiently realistic to deceive sophisticated adversaries without becoming overly complex. As deception networks scale in size to cover larger infrastructure, management overhead increases proportionally, requiring careful orchestration to ensure that decoys blend seamlessly with genuine production assets. Furthermore, high-interaction deception systems carry an inherent operational risk; if a decoy is not properly isolated from the production network, an attacker could potentially use it as a pivot point to compromise real systems, effectively turning the defender's weapons against them.

2.1.6 Static versus Dynamic Deception

Early cyber-deception implementations were mainly static. Once a system is deployed, it remains unchanged, always exposing the same services and vulnerabilities indefinitely. While this strategy proved effective against automated scanning tools and less sophisticated adversaries, it quickly became inadequate against Advanced Persistent Threats (APTs) [33].

Sophisticated attackers can identify static deception environments by using systematic fingerprinting and analysis. By probing a system over time, adversaries look for discrepancies in service responses, network latency, or default configuration artifacts that may make them believe that the system is artificially designed to trap them. They may also analyze resource constraints, such as CPU utilization or memory footprints, that do not align with the expected behavior of a production server. Furthermore, a wider network context can expose a decoy if an attacker observes that a seemingly critical database server receives no legitimate traffic from other internal systems. Once an adversary successfully fingerprints a static decoy, its defensive value is entirely neutralized. An attacker can simply map its location, bypass it in future lateral movements, and deduce the structural patterns of the defender's deception strategy.

To counter the fragility of static deployments, the research field has evolved towards dynamic deception methodologies [33]. Rather than presenting a fixed target, dynamic deception continuously alters the artificial environment. This paradigm relies on temporal variation, periodically altering the configurations, open ports, and operating system fingerprints of the decoys to invalidate any reconnaissance data that an attacker may have previously gathered. Moreover, dynamic systems can exhibit behavioral adaptation, shifting their responses in

real-time based on the observed tactics of the attacker. By incorporating environmental diversity and context awareness, dynamic deception ensures that decoys never look identical to each other and that the deceptive landscape evolves synchronously with the threat level, forcing adversaries into a perpetual and resource-intensive loop of reevaluation.

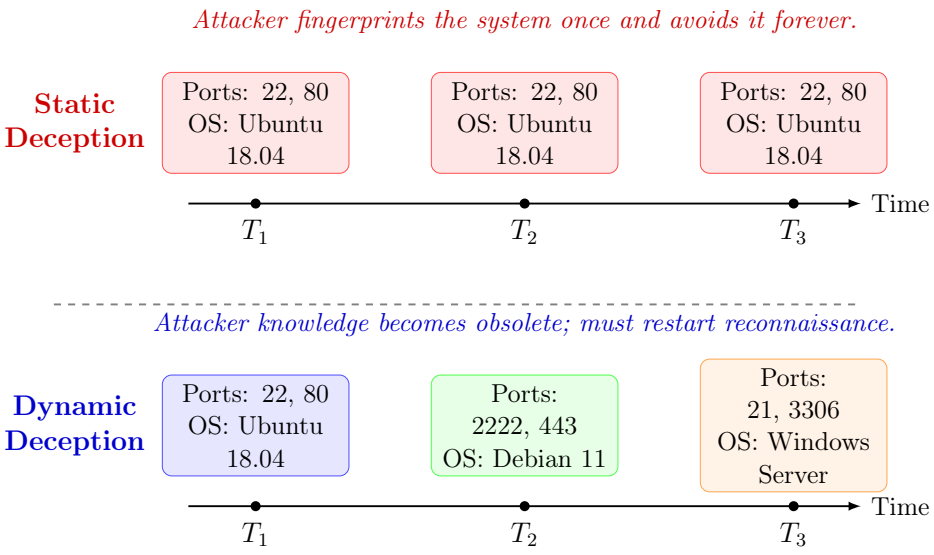

Figure 2.1: Comparison between static and dynamic deception strategies. A static honeypot remains unchanged, allowing an attacker to single fingerprint and permanently bypass it. A dynamic honeypot mutates its configuration over time, forcing the attacker into a perpetual state of reconnaissance.

2.2 Honeypot Technologies

A Honeypot is a security resource whose value lies in being probed, attacked, or compromised [37]. Differing from traditional preventive cyber-security mechanisms that try to deny unauthorized access, honeypots are designed to attract attackers (hence the name Honey Pot), and record their actions, providing the defenders intelligence that can be used to prevent real attacks.

2.2.1 Interaction Level Classification

Honeypots can be classified primarily based on their level of interaction with potential attackers. This classification is based on a fundamental tradeoff

between the depth of intelligence gathered and the resources required for deployment and maintenance [43].

Low-interaction Honeypots simulate only a limited set of services or network protocols. They do not run a full operating system; instead, they emulate specific behaviors (such as responding to an SSH handshake or presenting a login banner) without allowing actual access to a shell or file system. This makes them lightweight, easy to deploy in large numbers, and safe to operate because the attacker never gains real control over the system. The tradeoff is that sophisticated attackers can often distinguish them from real systems by sending unexpected commands or observing a limited range of responses. Honeyd [29] is an example of a low-interaction honeypot that can simulate entire network topologies with configurable service responses and has been widely used in research for studying scanning and reconnaissance activities.

Medium-interaction honeypots provide a more realistic environment. They typically simulate a full shell and filesystem, allowing attackers to execute commands, download tools, and attempt privilege escalation, whereas the honeypot logs every action. Cowrie [22] is the most widely used medium-interaction SSH and Telnet honeypot, it exposes a convincing UNIX-like environment to attackers, records all typed commands and downloaded files, and can even simulate the network connections initiated by an attacker. Cowrie's session logs have been used extensively in threat intelligence research because they capture the full sequence of attacker behavior from initial brute-force attempts through post-exploitation activity.

High-interaction honeypots run real operating systems and applications, giving the attacker full access to a genuine production-like environment. This provides the most amount of intelligence because the attacker behaves exactly as they would against a real target, using their actual tools and techniques without adapting for a simulated environment. However, high-interaction honeypots carry a significant risk; if not properly isolated, a compromised honeypot can become a launching point for attacks against other systems on the same network [18]. They also require more resources to deploy and maintain, because each honeypot is a full virtual machine that must be monitored and updated.

Multi-honeypot platforms, such as T-Pot, attempt to combine the advantages of different interaction levels. T-Pot simultaneously employs several honeypot technologies (including Cowrie and Dionaea) on a single host, integrating

their logs for centralized analysis. This approach allows defenders to cover a wider range of attack vectors while managing the infrastructure as a single deployment.

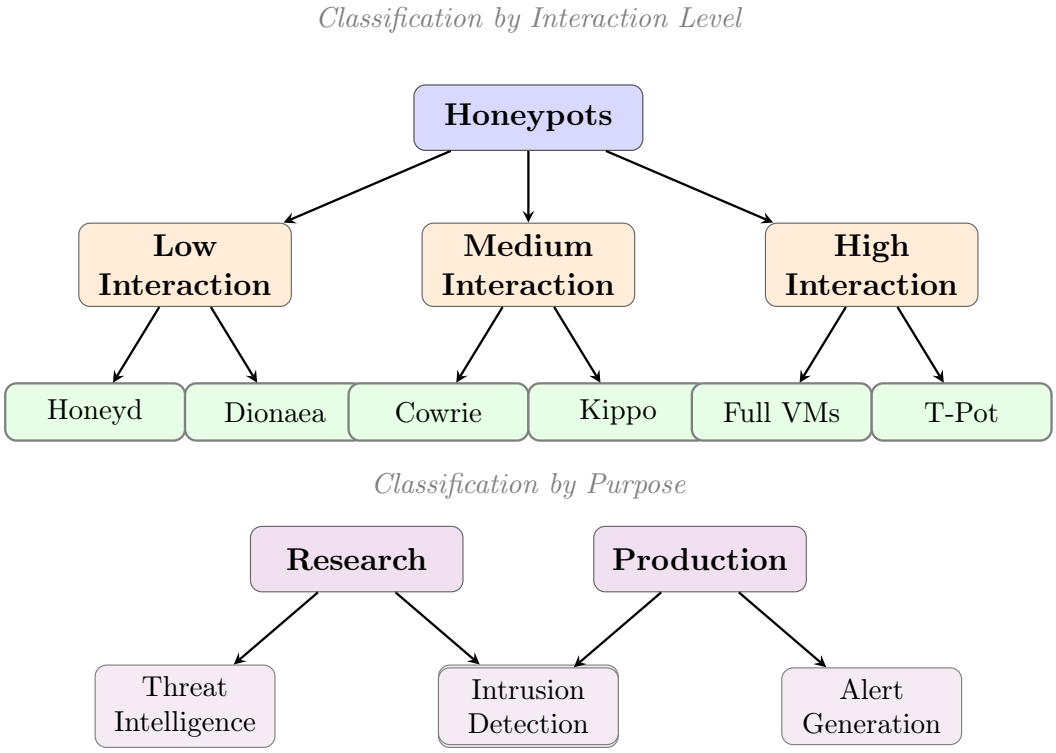

Figure 2.2: Honeypot classification by interaction level (top) and by purpose (bottom). Higher interaction levels provide richer intelligence but require more resources and carry greater operational risk.

2.2.2 Honeypot Implementations

The open-source community has been actively developing a range of tools that target specific protocols and attack surfaces. Dionaea [35] is a low-interaction honeypot designed to capture malware payloads by emulating vulnerable services, such as SMB, HTTP, FTP, TFTP, MSSQL, MySQL, and SIP. Its architecture is built around the concept of "catching" self-propagating malware by exposing services that worms and exploit kits commonly target and then storing the captured binaries for automated analysis. Dionaea is particularly effective at collecting samples from widespread, automated attack campaigns that scan entire IP ranges looking for unpatched services.

Conpot [15] addresses the growing threat landscape of Industrial Control Systems (ICS) and Supervisory Control and Data Acquisition (SCADA) environments. It emulates a range of industrial protocols, including Modbus/TCP, S7comm (used by Siemens PLCs), IEC 60870-5-104, BACnet, and EtherNet/IP. By presenting an authentic-looking human–machine interface (HMI) alongside protocol-accurate responses, Conpot can detect reconnaissance and exploitation attempts targeting critical infrastructure.

Glastopf [31] targets web application attacks by emulating a vulnerable web server capable of dynamically responding to attack payloads, such as Remote File Inclusion (RFI), Local File Inclusion (LFI), SQL injection, and command injection. Unlike static web honeypots that present fixed vulnerability pages, Glastopf uses a modular vulnerability emulation system that generates appropriate responses based on the type of attack detected, making it more difficult for attackers to fingerprint the honeypot through response analysis.

Table 2.1 provides a systematic comparison of the major open-source honeypot platforms discussed in this section, highlighting their differences in interaction level, target protocols, and primary use cases.

Table 2.1: Comparison of major open-source honeypot platforms.

Tool	**Interaction**	**Target Protocols / Services**	**Primary Use Case**
Honeyd [29]	Low	Configurable (TCP/UDP services, ICMP)	Network topology simulation, scanning detection
Cowrie [22]	Medium	SSH, Telnet	Brute-force and post-exploitation logging
Dionaea [35]	Low	SMB, HTTP, FTP, MSSQL, MySQL, SIP	Malware payload capture
Glastopf [31]	Low	HTTP (web application attacks)	SQL injection, RFI/LFI detection
Conpot [15]	Low–Medium	Modbus, S7comm, IEC 104, BACnet	ICS/SCADA threat intelligence
T-Pot [10]	Multi	All of the above (multi-honeypot)	Centralized deployment and analysis

2.2.3 Classification by Purpose

Beyond the interaction level, honeypots can also be classified according to their intended use [37]. **Research honeypots** are designed to collect information

about attack trends, new malware, and emerging techniques. They are typically deployed in academic or threat intelligence settings, where the goal is to understand the threat landscape rather than to protect a specific network. **Production honeypots** are integrated into an organization's existing infrastructure with the primary goal of detecting unauthorized access. They are simpler to deploy and maintain, and their alerts are detected and communicated immediately to the organization's security monitoring management.

2.2.4 Cloud-Based Honeypots

The diffusion of cloud computing has created new opportunities for honeypot deployment. A cloud-based approach to honeypots allows them to be deployed and destroyed quickly, making them suitable for deception strategies that require variations in their environment and configurations over time. While cloud hosting simplifies management and allows for the global distribution of decoys, it also enables attackers to detect new traces (cloud provider IP ranges, DNS configurations, and metadata service endpoints) [14].

2.3 MITRE ATT&CK Framework

The MITRE ATT&CK Framework (Adversarial Tactics, Techniques, and Common Knowledge; [21] is a publicly available source that catalogs the behaviors and methods used in real-world cyberattacks observed in past incidents. It has become the standard for describing offensive cyber operations across industries since 2015. The ATT&CK Framework was developed and maintained by the MITRE Corporation, a United States funded research and development center operating in the public interest, providing technical guidance to government agencies on matters of national security, defense, and cybersecurity [38]. It differs from other vulnerability databases, such as the CVE (also maintained by MITRE), that focus on specific software flaws by capturing complete adversary behavior at a higher level of abstraction. It documents the steps adversaries follow once they have access to a system without considering the specific vulnerability they used to gain access. This makes the framework extremely valuable for deception research, where the goal consists of anticipating and detecting attacker actions rather than patching individual software flaws.

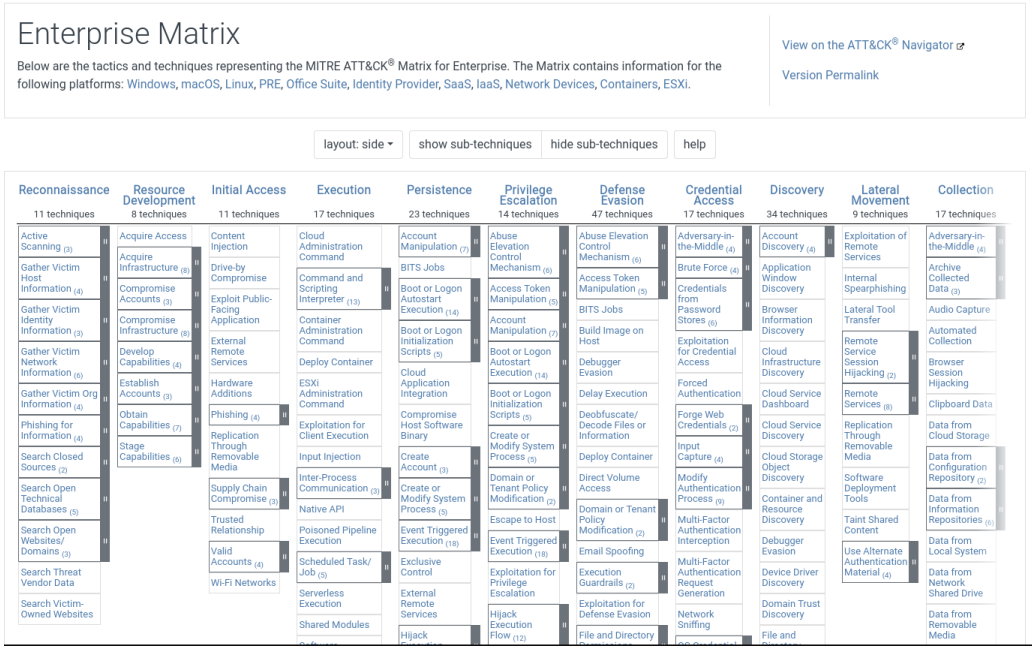

Figure 2.3: MITRE ATT&CK Enterprise Matrix showing tactics (columns) and their associated techniques (rows). Each tactic represents a phase of the attack lifecycle, progressing from Reconnaissance on the left to Impact on the right. Source: MITRE ATT&CK [20].

To organize adversary behavior, the system uses a hierarchical structure, with **tactics** at the top layer: the tactics represent the strategy objectives that an attacker pursues during an operation. They follow a natural progression that mirrors the lifecycle of an intrusion, referred to as the Cyber Kill Chain. The cycle begins with Reconnaissance and Initial Access (Infrastructure detection and Access entry to the network), following through Execution, Persistence and Privilege Escalation (the attacker consolidates its control on the infrastructure by gaining higher privilege in the infrastructure and leaving, for example, backdoors open to access again later). It then continues with Credential Access and Lateral Movement (for example, moving from a supply chain company to the actual target) and ends with Collection, Exfiltration, and Impact (methods that the attacker uses to achieve their final objective). The Enterprise matrix, which we used in our project, features 14 tactics and 200 techniques across Windows, Linux and other Operating Systems and platforms. Every tactic contains a set of multiple **Techniques**, describing specific methods used to achieve an objective. Many techniques can be further divided

in different **Sub-Techniques** that capture more granular variations. Every technique entry includes an alphanumeric identifier (ex. T1003 for Credentials Dumping), a detailed description of the attackers' behavior, references to real-world groups known to use it, the platforms it applies to, recommended detection strategies, and mitigation guidance. For example, the entry for T1003 (OS Credential Dumping) explains how attackers attempt to retrieve credential information from the memory of an operating system, Windows Registry, or password databases. It mentions that more than 30 threat groups have been observed employing this method, detailed sub-techniques for particular tools (such as T1003.001 for extracting LSASS Memory and T1003.008 for parsing /etc/passwd on Linux), and offers detection advice based on tracking specific API calls and file access patterns.

The sequence of phases in the Kill Chain, as described by ATT&CK, serves as a useful indicator of attack severity. Techniques linked to the later stages of an intrusion, such as Exfiltration or Impact, signify a more advanced and dangerous level of compromise compared to those in the initial stages, such as Reconnaissance or Discovery. When an attacker reaches the Exfiltration stage, they have already penetrated several defense layers, acquired credentials, and moved laterally within the network, making detection at this point crucial. This hierarchy of severity is directly relevant to deception strategies: setting up a honeypot that identifies Credential Access techniques (phase 7 out of 14) offers greater defensive benefit than one aimed at Reconnaissance (phase 1), as detecting an attacker at a later stage means they are deeper into the network and closer to inflicting significant harm.

The ATT&CK framework provides a common language between the offensive behaviors that deception tries to detect and defensive measures designed to take advantage of those behaviors.

The Proactive Decoy Selection Scheme [42] uses the ATT&CK taxonomy to map detected adversary techniques onto an attack graph and then applies optimization algorithms to determine the most effective placement for decoys.

The SPADE framework [2] uses ATT&CK technique metadata (names, tactical classifications, and descriptions) to construct structured prompts that guide a generative AI model to produce adaptive deception strategies. Their approach of contextualizing the prompt with information from the ATT&CK database proved effective in producing more targeted outputs than generic prompts. We adopted a similar technique for our prompt engineering design.

Rather than providing the LLM with only a technique ID (which is just an alphanumeric code), we included the full technique name, tactic classification, and description, allowing the model to understand the nature of the attack and generate a more appropriate honeypot configuration.

2.4 Meta Attack Language (MAL)

Threat modeling is the process of creating a representation of how an attack is performed on a system, identifying its core risks, possible attack paths, and the dependencies between them. Several formats can express threat intelligence: STIX [6] (Structured Information Expression) is commonly used in the industry to share compromise indicators (file hashes, IP addresses, and malware signatures) between organizations. Despite being useful in sharing the details of a threat, STIX does not describe the full progression of an attack through the infrastructure. MAL (Meta Attack Language) [40] was developed as a domain-specific language purposely built for threat modeling. It describes attack scenarios in terms of assets, attack steps, and relationships between them. An *asset* represents a component of the target system, such as a host, network segment, database, or authentication service. Each asset contains one or more *attack steps* describing the specific actions that an attacker can perform on that component. The relationships between attack steps are expressed with the -> operator, which indicates a deterministic causal link (if step A is reached, then step B becomes reachable), whereas the +> operator indicates a probabilistic or conditional dependency. The *associations* between assets define how different components are connected. MAL annotates individual attack steps with the expected time or difficulty required to execute them, allowing for quantitative risk analysis. The structure of MAL is particularly useful for our pipeline, as it offers a machine-readable depiction of the attack scenario that seamlessly converts into a directed graph. In this graph, each step of the attack is represented as a node, and each causal link is depicted as an edge, effectively illustrating the entire sequence of actions that an attacker must undertake to move from the initial breach to their ultimate goal. Our pipeline leverages this graph structure directly: the MAL parser identifies the nodes and edges, the classifier associates each node with a MITRE ATT&CK technique, and the Strategy Planner calculates Betweenness Centrality scores on the graph to pinpoint the most crucial attack steps for placing decoys.

MAL was originally developed within the SecuriCAD ecosystem [9], a commercial threat modeling platform that uses MAL specifications to run automated attack simulations and compute the probability of successful compromise across enterprise networks. SecuriCAD has been adopted by organizations to model threats against IT infrastructure, SCADA systems, and cloud environments. While our pipeline does not perform probabilistic simulations (we do not compute the likelihood of each attack step succeeding), we reuse the graph structure provided by MAL. The asset definitions and causal links provide sufficient information for our system to construct the attack graph, classify the techniques, and determine where decoys should be placed without requiring the full probabilistic analysis that SecuriCAD performs.

Listing 2.1 shows a simplified example of an MAL specification modeling a lateral movement scenario.

Listing 2.1: Example of a MAL specification for Lateral Movement

```
category System {
  asset Host {
    | compromise
      +> systemInfo

    | systemInfo
      user info: System Information Discovery T1082
      +> networkScan

    | networkScan
      user info: Network Service Scanning T1046
      +> credentialAccess

    | credentialAccess
      user info: Credential Dumping T1003
      +> lateralMovement

    | lateralMovement
      user info: Remote Services T1021
  }
}

associations {
  Host [client] 1 <-- Connection --> 1 [server] Host
}
```

In this example, the `Host` asset is defined in the `System` category. It con-

tains a sequence of five attack steps representing a lateral movement kill chain: starting from a generic `compromise`, the attacker progresses to `systemInfo` gathering (mapped to MITRE T1082), after to `networkScan` (T1046), then to `credentialAccess` (T1003), and finally, `lateralMovement` (T1021). The `+>` operator between each step establishes the causal chain: the attacker cannot perform credential dumping without first scanning the network and cannot move laterally without first obtaining credentials. The `user info` annotations provide human-readable descriptions that include the MITRE identifier extracted by the parser for classification. The `associations` block defines a bidirectional `Connection` between two `Host` assets, representing the network link over which lateral movement can occur. When our pipeline processes this file, it constructs a directed graph with five nodes and four edges, maps each node to its corresponding ATT&CK technique, and computes the Betweenness Centrality to determine which technique is the most strategically important target for a decoy.

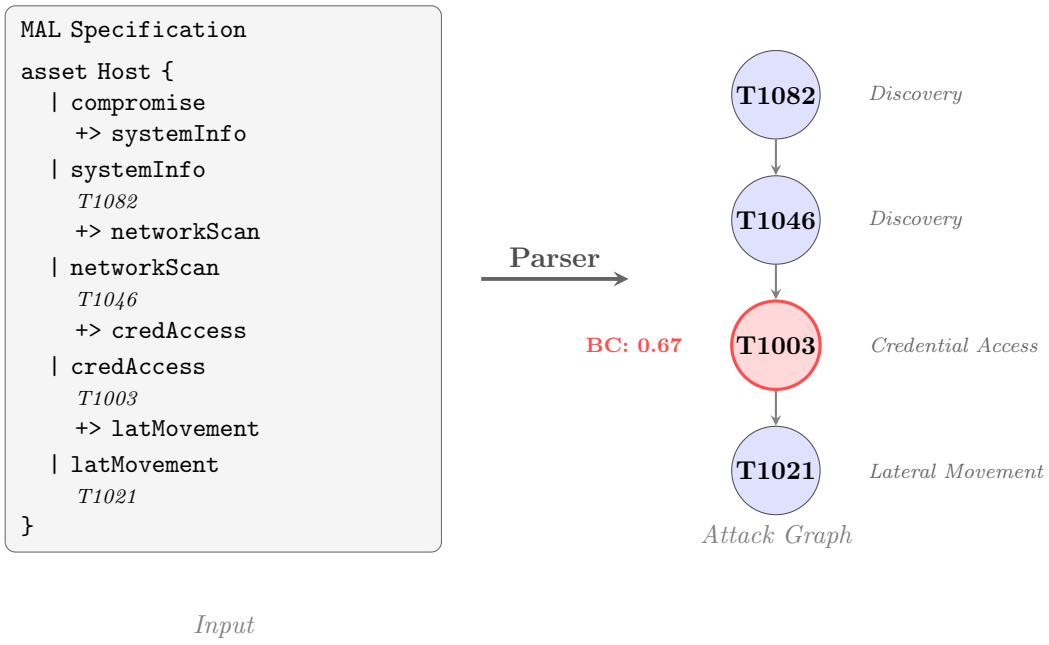

Figure 2.4: Transformation of a MAL specification into a directed attack graph. The parser extracts attack steps and their causal dependencies, producing a graph where each node represents a MITRE ATT&CK technique and edges represent the attacker's progression. The node with the highest Betweenness Centrality (BC) score (highlighted in red) is selected as the priority target for decoy placement.

2.5 Resource-Aware Honeypot Deployment

One of the challenges faced by deception is resource management. The deployment process of a honeypot requires extensive computing resources (virtual machines, network bandwidth, and storage); consequently, this can result in unmanageable costs in both cloud and self-hosting environments. A deception system that creates a honeypot for every possible attack technique could become too expensive to maintain quickly, while also becoming easier to detect. [18]. This gives the question of where to place decoys the same importance as how to build them. The goal is to select a small number of deployment points that cover as much of the attack surface as possible so that limited resources are used where they have the most impact. In the study "A Proactive Decoy Selection Scheme for Cyber Deception using MITRE ATT&CK" [42], the authors propose a method that uses attack graphs and Betweenness Centrality (BC) to identify the best positions for honeypot placement. Their approach represents the attack scenario as a directed graph, where nodes correspond to attack techniques and edges represent the dependencies between them. By computing the BC score for each node, they identified the techniques that act as bottlenecks in the attack progression, meaning that many attack paths pass through them. Placing a decoy at a high-BC node maximizes the chance that an attacker will encounter it, regardless of the specific path they follow. The problem of resource constraints in deception has also been studied in the context of microservice architectures. The study "Resource-Aware Cyber Deception for Microservice-Based Applications" [4] addresses the challenge of deploying decoys in containerized environments, where each service consumes a share of a limited resource budget. Their approach introduces explicit budget constraints into the decoy selection process, ensuring that the total cost of deployed honeypots remains within a predefined limit while still covering the most critical attack paths. This formulation aligns with the constraints we implemented in our Strategy Planner, which caps the number of deployed decoys and uses a coverage threshold to avoid over-provisioning. Automated tools, such as MulVAL [23] can generate attack graphs from vulnerability scanner data and network topology, producing comprehensive graphs with thousands of nodes for real enterprise networks. Our pipeline takes a different approach: it constructs a graph from a MAL threat model, where the security analyst has already defined the relevant attack steps and their dependencies at a higher

level of abstraction. This reduces graph complexity and makes BC computation more tractable while still capturing the essential structure of the attack scenario.

2.6 LLMs for Infrastructure as Code Generation

Recently, new discoveries in the LLM Generation field have provided new research fields in the cybersecurity space. Tools such as Terraform, Ansible, and CloudFormation allow users to define computing infrastructure using declarative configuration files. This approach brings the same advantages to infrastructure management that version control brings to software development: configurations can be reviewed, diffed, rolled back, and reproduced exactly across environments. The most widely adopted IaC tools include Terraform (which uses the HashiCorp Configuration Language, HCL), Ansible (which uses YAML playbooks), AWS CloudFormation (JSON or YAML templates), and Pulumi (which embeds infrastructure definitions in general-purpose programming languages like Python or TypeScript). Our pipeline specifically targets Terraform because of its cloud-agnostic design. The same HCL syntax can provision resources on Google Cloud Platform, AWS,Azure, or any provider with a Terraform plugin, making it the most flexible choice for systems that may need to deploy honeypots across different cloud environments. However, HCL presents unique challenges for language model generation. Unlike general-purpose programming languages, HCL is a declarative language where the order of blocks does not matter, resource references must match exactly (a misspelled resource name causes a hard failure rather than a runtime error), and provider- specific arguments change with each version. A model that generates syntactically valid HCL may still produce configurations that fail validation because they reference deprecated arguments, use incorrect resource types, or create circular dependencies between resources. The question of whether LLMs can reliably produce these configurations has surfaced in fields such as academia and industry. [5]. In the LLM IaC Feedback Loop [25], an approach is proposed where the output of the model is validated using Terraform's built-in functions (`terraform validate` and `terraform plan`), with any error being sent back to the model for error correction. The study affirms that this approach increased the model's success rate, with some models reaching over 90% validity after three correction cycles compared to 60% of the first

attempt. This result is highly relevant to the design of our validator module in our implementation. The idea of specializing a general-purpose language model for code-related tasks has been explored in several studies. Code Llama [32], a code-focused variant of the Llama family developed by Meta, showed that taking a pretrained language model and fine-tuning it on code-specific data leads to significant improvements in code generation tasks without losing the model's ability to understand natural language. This two-phase approach (first general pre-training, then domain-specific fine-tuning) is the same strategy that we followed in our pipeline. Instead of training a model from scratch on Terraform code, we fine-tuned Llama-3.1-8B-Instruct on a dataset of honeypot-specific configurations using QLoRA. This allows the model to keep its existing understanding of code structure while learning the specific patterns needed for honeypot Terraform variables. Base models already have some familiarity with HCL from their pre-training data but lack the specialized knowledge required to generate honeypot configurations (such as the correct setup for services like Cowrie, Dionaea, and T-Pot). Domain-specific fine-tuning fills this research gap. The idea of specializing a general-purpose language model for code-related tasks has been explored in several studies. Code Llama [32], a code-focused variant of the Llama family developed by Meta, showed that taking a pre-trained language model and fine-tuning it on code-specific data leads to significant improvements in code generation tasks without losing the model's ability to understand natural language. This two-phase approach (first a general pre-training, then a domain-specific fine-tuning) is the same strategy we followed in our pipeline: instead of training a model from scratch on Terraform code, we fine-tuned Llama-3.1-8B-Instruct on a curated dataset of honeypot-specific configurations using QLoRA. This allows the model to keep its existing understanding of code structure while learning the specific patterns needed for honeypot Terraform variables. Base models already have some familiarity with HCL from their pre-training data but lack the specialized knowledge required to generate honeypot configurations (such as the correct setup for services like Cowrie, Dionaea, and T-Pot). Domain-specific fine-tuning fills this research gap.

2.6.1 Prompt Engineering Techniques

To optimize Large Language Models outputs for complex tasks, we can rely on the technique of **Prompt Engineering** to design structured prompts that enhance the model's reasoning and understanding capabilities. Instead of interacting with the model in a more casual, conversational way, we can specify to the llm what we expect from the its output, obtaining a structured response suiting its user's needs. **Zero-Shot Prompting** is an approach where the model is asked to perform a task without any previous examples, relying completely on its pre-trained knowledge to understand the scope of a prompt. While being efficient in resource consumption, this method requires very clear instructions and a model that has been trained on a fine-tuning dataset that is useful for the domain of our request. **Few-Shot Prompting** expands the inputs given by the user with other examples, showing the correlation between input and the expected output, establishing a pattern for the model to follow at the cost of larger context windows and higher resource consumption. Other than simply giving the model examples, there other techniques we can use to get better outputs from llms: **Chain-of-Thought Prompting** focuses on the model's reasoning capabilities by forcing the LLM to divide complex tasks in sequential steps before arriving at a final answer, improving performance when complex tasks are requested. Another way to set boundaries to an LLM's behavior is by using **Role-Based Prompting.** With this technique we can define a *persona* to a model (ex. "You are a security engineer..."), establishing a style and technical boundary preventing irrelevant conversational filler. The SPADE (Structured Prompting for Adaptive Cyber Deception) [2] is an emerging framework making use of this approach by using six explicit instruction components: *Identity/Persona* (leveraging Role-based prompting) and a specific *Goal/Task* to align the model with defensive operations. To ensure the output is contextually relevant, it injects a *Threat Context* (such as specific MITRE ATT&CK techniques) and provides a *Strategy Outline* that dictates operational constraints and high-level tactics. Finally, to guarantee deployability, the framework provides an *Output Example* (Few-shot prompting) and strict *Output Instructions* (such as requiring a valid JSON schema). By combining multiple prompt engineering strategies into a single, rigorous template, SPADE avoids generalized outputs and hallucinations.

2.6.2 Security of AI-Generated Infrastructure

While LLMs are useful in the process of generating Infrastructure as Code, their use introduces various security implications. The base models are trained on a foundation of publicly available code, often containing insecure coding patterns, hardcoded secrets, and misconfigurations. Consequently, LLMs can learn these scenarios and reproduce them in their outputs. Systematic investigations into the security of AI-generated code, such as evaluations of GitHub Copilot's contributions, have revealed concerning rates of insecure outputs [28]. By prompting language models to generate code for scenarios relevant to high-risk Common Weakness Enumerations (CWEs), researchers found that approximately 40% of the generated programs contained security vulnerabilities. In the specific context of Infrastructure as Code, these vulnerabilities manifest as structural flaws rather than runtime memory corruption. An LLM generating Terraform for an AWS environment might correctly instantiate an EC2 instance but implicitly attach a default Security Group that allows inbound traffic from `0.0.0.0/0` (the entire Internet) or generate an S3 bucket configuration that lacks encryption at rest. Furthermore, when striving to complete a prompt, models may take "optimization shortcuts," omitting essential security controls such as proper Identity and Access Management (IAM) restrictions unless the prompt explicitly demands them.

2.6.3 Parameter-Efficient Fine-Tuning for Code Generation

The fine-tuning process of Large Language Models requires updating billions of parameters, requiring massive computational resources (high-end GPUs with high amounts of VRAM) that are often inaccessible to individual researchers or small organizations. To address this bottleneck, researchers have developed Parameter-Efficient Fine-Tuning (PEFT) techniques, which achieve comparable performance by training only a small subset of additional parameters while keeping the original model weights frozen.

One of the most widely adopted PEFT methods is Low-Rank Adaptation (LoRA), which updates dense neural network layers with pluggable low-rank matrices [41]. An optimization of this technique is quantized low-rank adaptation (QLoRA), which further reduces memory requirements by quantizing the base model to 4-bit precision while maintaining fine-tuning capability through

low-rank adapters [8].

Recent studies [36] have demonstrated that PEFT techniques, including LoRA and QLoRA, can successfully specialize LLMs for automated code generation tasks, often outperforming in-context learning (prompting) while significantly reducing memory consumption. These techniques are also useful for improving the security of the generated code (specifically in C/C++ environments) and consistently outperform prompt-based approaches for secure code generation [17].

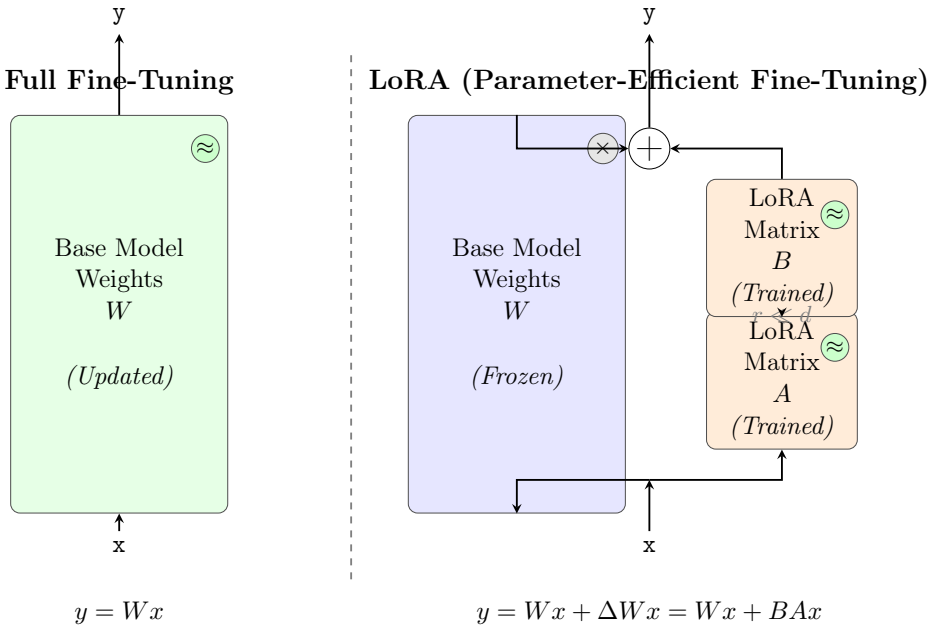

Figure 2.5: Conceptual diagram of Full Fine-Tuning versus Low-Rank Adaptation (LoRA). While full fine-tuning requires updating all parameters of the base model W, LoRA freezes the pre-trained weights and injects trainable rank decomposition matrices (A and B) into each layer, massively reducing the number of learned parameters.

2.6.4 Synthetic Data Generation for Cybersecurity

The lack of high-quality, labeled training data is a significant challenge in applying machine learning to cybersecurity, particularly in specialized areas such as deception technology. Enterprise security configurations are seldom publicly disclosed, and manually creating a sufficiently large dataset of honeypot IaC configurations for training an LLM is a daunting task.

To address this data bottleneck, the research community is increasingly

looking to LLMs themselves as a solution. As noted in a survey by Long et al. [19], LLMs can serve as "data-centric engines" to generate synthetic data, effectively mitigating the limitations posed by real-world data scarcity. In a typical synthetic generation pipeline, a highly capable (and often larger) *teacher* model is prompted to generate, augment, or annotate data samples, which are then used to train a smaller and more efficient *student* model.

This approach is gaining momentum in the cybersecurity sector, where large models are employed to synthesize network traffic logs, Indicators of Compromise (IoCs), and code snippets for threat detection models. Our pipeline directly implements this paradigm. We utilized a larger, highly capable general-purpose model (Google Gemini) as our automated data curator to process, clean, and enrich a raw collection of GitHub repositories into a structured JSON-formatted dataset suitable for fine-tuning our local Llama 3.1 model.

2.6.5 Automated Honeypot Generation

The symbSODA framework [34] approaches the problem of automated honeynet orchestration through formal verification and model checking rather than LLM generation. Its system uses symbolic analysis to verify that the deployed deception environment satisfies the specified security properties, ensuring that the decoys cannot be exploited to compromise the real infrastructure. While symbSODA provides strong formal guarantees, it requires manual specification of the honeypot configurations, which is the bottleneck our LLM-based approach aims to eliminate.

The two-tier deception model, described by various authors [3] separates the strategic (which assets to protect) and tactical (how to configure decoys) layers of deception. Our pipeline naturally implements this separation: the Strategy Planner handles the strategic layer by selecting which techniques deserve a dedicated decoy, while the LLM Generator handles the tactical layer by producing the actual configurations for each selected technique.

Chapter 3

Methodology

In this chapter, we describe the design of our experimental system, justifying its architecture, model selection, training methodology, and deployment that led to the implementation described in Chapter 5.

3.1 System Architecture

The architecture of our proposed solution follows a fully modular approach. Instead of building a single application that handles all operations, we decided to organize the structure into different independent modules that execute sequentially: the MAL file Parser, Threat Classifier, Strategy Planner, LLM Generator, LLM Output Validator, and GCP Deployer. This distinct approach was a crucial design decision that offered numerous practical and methodological benefits during project development. Primarily, it established clear input-output relationships between the various phases of the pipeline. Each module functions as an independent unit, receiving a specific data structure, processing it, and passing a standardized output to the subsequent stage. For example, the Parser transforms a text file into a graph, the Classifier extracts MITRE techniques from the graph, and the Generator converts these techniques into Terraform code.

Second, the modular design was crucial for debugging and testing, given the experimental nature of employing an AI model to generate Infrastructure as Code. Large Language Models are inherently nondeterministic and can sometimes produce unexpected or hallucinated outcomes. By isolating the LLM Generator from the deterministic components, such as the MAL parser or the

Strategy Planner's mathematical computations, we could more conveniently identify the source of any issue. If a deployment faced problems, we could determine whether to investigate a misconfigured Terraform template or an error in the graph analysis, rather than navigating through a complex, monolithic codebase. The modularity of the design also enabled the components to be developed in parallel, allowing faster development by implementing them before the modules that required a major priority. For example, we could test the generation part of our pipeline before finalizing the classification components by utilizing mock data as input.

Designing the system into separate modules significantly enhances the extensibility of the project for future development. For instance, if a future version of this work needs to support a different input data format, only the Parser module would require rewriting. Similarly, adapting the system to support a different cloud computing provider instead of Google Cloud would necessitate only a new Deployer module, leaving the intricate core logic of threat classification and code generation intact.

3.1.1 Local Execution

Our deception system features a fully local offline approach. The individual components of the pipeline (from the MAL Parser to the GCP deployer module) run entirely on the user's host machine hardware, with the only exception being the GCP platform where the honeypots are deployed. We explicitly designed the entire stack to avoid external LLM providers (OpenAI, Anthropic, etc.) for the following reasons. First, for data privacy and sovereignty in a deployed production environment, the entire pipeline analyzes and generates sensitive data (network topology, server configurations, and known vulnerabilities). Relying on third-party cloud platforms to handle highly sensitive threat intelligence for defense development poses a significant and unacceptable security risk. By implementing a local model with open weights and adopting a secure self-contained model, we ensured that no proprietary infrastructure data left the organization's boundaries.

The local execution of tasks enhances security and ensures essential operational autonomy. The Code Validator module utilizes a repetitive feedback mechanism that automatically directs the LLM to revise the code multiple times when syntax errors are detected. In a cloud-based system, this iterative

process rapidly consumes API tokens and can trigger rate-limiting restrictions, thereby disrupting the deployment process. Our self-hosted model incurs no additional cost per generation, allowing the pipeline to iterate as often as necessary to produce valid code.

From a long-term maintenance perspective, the utilization of an internally hosted model ensures strict reproducibility. Cloud providers frequently update, filter, or discontinue their models without prior notice; an update aimed at enhancing conversational safety may inadvertently diminish a model's ability to generate terraform infrastructure. In contrast, our fine-tuned Llama 3.1 model remained unchanged and stable. This guarantees consistent outcomes over time and allows the system to be deployed, even in air-gapped defense networks that are completely disconnected from the Internet, except for the necessary GCP endpoint.

3.2 Model Selection

We chose the correct language model for our system by considering a balance between model capabilities and hardware requirements. We established three crucial criteria for the candidate models. First, the model must be open source with open weights, allowing us to fine-tune it and run it locally to comply with the data privacy standards mentioned previously. Second, it had to be compact enough to fit into the Video RAM of a single high-end consumer GPU, such as an NVIDIA RTX 3090 with 24 GB of VRAM, when using 4-bit or 8-bit quantization. Finally, the model required a strong baseline capability to follow strict formatting instructions and generate structured code, specifically the HashiCorp Configuration Language (HCL) for Terraform, rather than merely producing conversational text.

During our model evaluation and research, we quickly eliminated models with flagship-level parameter sizes, such as LLMA 3 70b and Qwen 72 B, because they require an extensive amount of VRAM and a multi-GPU cluster, which do not meet our specifications. Consequently, we focused our research on models with fewer parameters, specifically those with seven to nine billion parameters. Good candidates were models like Mistral-7B, Meta's Llama family, and Deepseek-Coder-7b.

To decide the model that best fit our design needs, we ran a series of generation tests after fine-tuning each candidate model. Mistral-7b despite promising ex-

cellent performance with its benchmark results, showed significant difficulties in producing raw IaC code: Even with our prompt engineering approach to constraint its output, it frequently introduced conversational phrases before the code blocks ("Here is your Terraform configuration:"). Deepseek-Coder-7b, a model that is specifically designed for code generation, showed significantly better results than Mistral, however its behavior showed that its fine-tuning was biased towards general-purpose programming language (Python, Java, C++), making it unreliable in our Terraform syntax. During our tests, Deepseek-Coder frequently generated Terraform resource blocks using syntax patterns borrowed from other languages, such as using Python-style string formatting within HCL expressions or omitting the required assignment operators in variable definitions. The resulting configurations consistently failed the `terraform validate` step and could not be corrected through our feedback loop, as the model would reproduce similar structural errors in subsequent attempts.

We chose **Llama-3.1-8b-Instruct** because of its ability to strictly follow structured prompts. In our pipeline, it is essential that the model exclusively generates valid `.tfvars` definitions, steering clear of any introductory phrases such as "Here is your code." The 8 B parameter count of Llama 3.1 has proven to be the optimal balance, significantly enhancing coding intelligence compared to older 7 B models, while remaining easily manageable on our target local hardware through QLoRA quantization.

3.2.1 Acknowledged Limitations

From a methodological standpoint, it is crucial to acknowledge that an 8-billion parameter model cannot match the intricate logical reasoning or extensive contextual capabilities of advanced models, such as OpenAI's GPT-5 or Google's Gemini 3. In the initial trials, our local model occasionally experienced "hallucinations", such as creating non-existent Terraform resources, omitting a closing parenthesis, or incorrectly escaping Bash scripts within heredoc blocks.

However, rather than opting for a larger cloud-based model and abandoning local execution, we addressed these limitations through architectural adjustments. We significantly reduced the task requirements of the LLM by restricting it to generating only variable files instead of the entire Terraform infrastructure file and integrating its generation process into the multi-attempt

feedback loop of the Code Validator. This approach implies that the system does not require the LLM to be flawless on the first attempt; it only needs to be capable of correcting its own syntax errors when provided with the Terraform CLI error log.

3.3 Model Fine-Tuning

Our fine-tuning approach was driven by a notable gap between general language modeling and the specific requirements of cyber-defense applications. While base instruction models possess a basic understanding of HashiCorp Configuration Language (HCL), they lack the specialized knowledge needed to transform abstract threat intelligence, such as a particular MITRE ATT&CK technique, into a tangible and operational honeypot framework. Therefore, the primary aim of the fine-tuning process was to enforce syntax and achieve semantic mapping: guiding the model to understand the intricate links between adversary actions and the specific infrastructure setups (such as open ports, decoy services, and logging systems) necessary to detect them.

3.3.1 Low-Rank Adaptation (QLoRA)

Fine-tuning an 8-billion parameter model's core behavior through full-parameter fine-tuning is a highly resource-intensive endeavor. This requires specialized, multi-GPU enterprise server clusters, which directly challenge our fundamental architectural need for local, cost-effective execution on consumer hardware.

To overcome this limitation, we used Quantized Low-Rank Adaptation (QLoRA) [8]. Introduced by Dettmers et al., QLoRA offers an efficient fine-tuning method that significantly reduces memory usage without compromising task performance. By maintaining the extensive pretrained Llama 3.1 base model in an extremely efficient 4-bit precision format, we significantly reduced its memory usage to a small fraction of its initial size.

As detailed in the original paper [8], this memory efficiency is achieved through three key innovations: 4-bit NormalFloat (NF4), a new data type that is theoretically optimal for normally distributed weights; Double Quantization, which further reduces the memory footprint by quantizing the quantization constants; and Paged Optimizers, which prevent memory spikes during gradient checkpoints.

The training process was then set up to update only small, secondary low-rank matrices (adapters) inserted into the attention layers while backpropagating gradients through the frozen 4-bit model. This optimization enabled us to teach the model the rigorous structural rules of Terraform and Bash entirely on a single NVIDIA RTX 3090 GPU (24GB VRAM). This approach achieved state-of-the-art specialization without the prohibitive hardware entry barrier associated with traditional model retraining.

3.3.2 Supervised Fine-Tuning (SFT)

To achieve the desired behavioral adaptation, we utilized Supervised Fine-Tuning (SFT). Modern aligned language models often employ complex techniques such as Reinforcement Learning from Human Feedback (RLHF) [24] to understand subjective human preferences and conversational patterns. However, our generation pipeline required the objective and deterministic outcome of needing the generated code to compile correctly and deploys specified decoy services. SFT is mathematically optimal for this type of deterministic pattern recognition. Using the custom dataset detailed in Chapter 4, we provided the model with numerous high-quality examples that paired a threat intelligence prompt with the exact, correct Terraform variable configuration. During the supervised training phase, the model learned to generate correct outputs by comparing its predictions with examples in the dataset.

3.4 Prompt Engineering

While fine-tuning allows us to customize Llama-3.1-8b-Instruct for our purposes by learning HCL syntax and specialized Bash scripts, it does not provide the model with the specific context required to generate a customized defense measure for every threat. We require the model to act in a context-aware manner instead of producing generic configurations. For this purpose, we adopted a structured prompt engineering approach inspired by the SPADE framework [2], a modular prompt architecture composed of six different components: Identity/Persona, Goal/Task, Threat Context, Strategy Outline, Output Example, and Output Instructions. We chose this method because our generation task faces the same fundamental challenge that SPADE addresses: transforming threat intelligence into deployment-ready outputs using an LLM. In our ini-

tial trials, without a structured prompt design, the model frequently produced generic honeypot configurations that either ignored the specific attack context or were hallucinated, producing conversational responses instead of generating valid code. By adopting SPADE's modular approach, we mitigated these issues by providing context to the LLM and enforcing output instruction constraints.

3.4.1 Threat Context and Output Instructions

The first design principle aligns with SPADE's **Threat Context** component. A basic MITRE technique identifier like "T1003" is only an alphanumeric code that lacks semantic meaning for the model. For the LLM to create an effective honeypot, it must understand the nature of the attack, the Kill Chain phase it belongs to, and the type of infrastructure required for its detection. We designed the prompt to include the technique's readable name, tactical classification, and a natural-language explanation of the adversary's goals. With this information, the model can infer the nature of the attack (ex. recognizing that a Credential Access technique should result in a honeypot featuring authentication services and decoy credentials) rather than relying only on memorized associations from its training data.

The second design principle is related to SPADE's **Output Instructions** component. An unrestricted LLM often produces verbose, conversational, and inconsistently formatted outputs, which are problematic because the generated code must follow a strict format (Terraform variables with embedded Bash scripts) and pass automated validation. To address this, we incorporate explicit formatting guidelines directly into the prompt: the output must be pure Bash code, adhere to a maximum length, and begin with a standard shebang header. By constraining the output in this way, we reduce its variability and enhance the reliability of the post-processing and validation stages, while also preventing the model from producing overly complex scripts that would be more challenging to debug and more prone to failure during deployment.

3.5 LLM Generation

The Generation section is the core of our architecture, with the task of creating Infrastructure as a Code honeypots for deployment in Google Cloud. To ensure the reliability of our outputs, we implemented several design decisions that

impose strict constraints on the generated results.

3.5.1 Variable-Only Generation

One of the most important decisions is to limit the LLM's output scope. Instead of having the model generate a complete terraform configuration from scratch, including provider declarations, resource blocks, networking rules, and firewall settings, we confined its role to crafting only `.tfvars` variable definitions and an embedded startup script. The remaining infrastructure, such as the `google_compute_instance` resource block, provider configuration, and network setup, is detailed in a static, human-written base template that the LLM does not alter.

This choice was informed by a practical insight: in early trials, when we tasked the model with generating entire Terraform files, it frequently introduced structural errors, such as missing provider blocks, malformed resource references, or incorrect HCL syntax in nested blocks. It was difficult to correct these errors automatically because they affected the overall file structure rather than isolated mistakes. By limiting the model to variable definitions, we significantly reduced the potential for errors. The variables follow a straightforward key-value format (`machine_type = "..."`, `disk_size = ...`) that the model handles more reliably, whereas the startup script—the most creative aspect of the output—is contained within a heredoc block that the base template already anticipates.

This separation simplifies the training process; the dataset only needs to include input-output pairs of prompts and variable definitions, rather than complete Terraform projects. Consequently, all 800 training samples adhered to the same concise format, which aided the model in learning the mapping more consistently.

3.5.2 Multi-VM Generation

In most real-world threat models, the analysis identifies multiple techniques that target different services or network layers. If all these techniques were handled by a single honeypot, the resulting VM would need to expose several unrelated services simultaneously, for instance, an SSH server for Credential Access techniques alongside a DNS resolver for Command and Control techniques on the same machine. In addition to being unrealistic from an attacker's

perspective, this creates practical problems: the startup script must install and configure all services without conflicts, logging must separate events by technique, and a failure in one service's setup can break the entire deployment.

We designed the pipeline to support multi-VM generation, where each technique selected by the Betweenness Centrality optimization is deployed on a separate virtual machine with its own dedicated startup script. Each decoy runs only the services relevant to the technique it simulates, making it more credible during attacker reconnaissance. A machine that presents itself as a standalone SSH server behaves more like a real production host than one that runs five unrelated services. From a defensive standpoint, this also simplifies log analysis and attack attribution because each VM is tagged with its corresponding MITRE ATT&CK identifier, and any interaction with it can be directly linked to a specific attack vector.

The concern with this approach is resource consumption. Deploying one VM per technique in the threat model can generate dozens of virtual machines, leading to excessive cloud costs and an unrealistic deployment footprint. To prevent this, we defined two stopping criteria in the Strategy Planner's decoy selection logic. The first is a coverage threshold: the algorithm iteratively selects techniques in descending order of Betweenness Centrality score until at least 80% of all detected techniques are covered. The second is a hard limit of five decoys, which caps the maximum number of VMs regardless of the coverage level. These thresholds were chosen to balance detection coverage with deployment cost: 80% coverage ensures that the vast majority of the attack surface is monitored, whereas the 5-VM limit keeps the infrastructure manageable on a limited cloud budget.

Coverage is further optimized through sub-technique matching. When the planner selects a parent technique (e.g., T1021 – Remote Services), it also counts all its sub-techniques (T1021.001, T1021.002, etc.) as covered because a decoy designed for the parent technique will also attract interactions targeting the more specific variants. This hierarchical matching allows the pipeline to achieve high coverage with fewer deployed VMs.

Finally, each selected technique influences the VM resource allocation. The Strategy Planner scales the machine size based on the number of techniques and the overall risk score: higher-risk scenarios receive more powerful instances (e.g., `e2-standard-2` with two vCPUs and 8 GB of RAM) to handle more complex honeypot services, whereas lower-risk deployments use smaller machine

types to minimize costs.

This multi-VM design also affects the construction of prompts. In the single-VM case, the pipeline uses a prompt that matches the training data format, which works well because the model produces more reliable outputs when the input structure is consistent with what it has learned during fine-tuning. In the multi-VM case, however, a single prompt listing of multiple techniques leads to poor results: the model tends to mix configurations from different techniques into the same script. To avoid this, we use a separate **specialization prompt** for each technique, asking the model to generate one focused startup script at a time. Table 3.1 shows the structure of such a prompt for the T1003 technique.

Table 3.1: Example of a dynamically generated specialization prompt for MITRE technique T1003.

Component	**Content**
Threat Context (from MITRE database)	
Technique	OS Credential Dumping (T1003)
Tactic	Credential Access
Description	Adversaries may attempt to dump credentials to obtain account login and password material [...]
Output Instructions	
Task	Create a Debian 11 bash script that deploys a honeypot specifically designed to detect and log attacks matching this technique. The script should install appropriate services and create convincing decoy data.
Constraint	Max 50 lines, pure bash, start with `#!/bin/bash`

3.5.3 Validation Feedback Loop

As mentioned in Section 3.2, our 8-billion parameter model will not always produce a correct output on the first try, with the code containing syntax errors, missing variables, or invalid resource references, leading to deployment failure. To comply with this behavior, we introduced an iterative validation feedback approach with the help of the Terraform CLI. After each generation, the output is validated using the `terraform validate` command, and if the validation fails, we send the error messages and the original code to llm. The model is then asked to correct the invalid code by repeating this cycle up to five times until a valid IaC configuration is produced. The Terraform CLI is par-

ticularly useful in this scenario because it outputs precise errors (ex. "Missing required argument: machine_type") that the model is able to interpret. This choice allows us to maximize the resources at our disposal while minimizing the complexity of the overall architecture by reusing the fine-tuned model to correct its own errors, as it already understands the expected output format. Table 3.2 presents an example of the corrective prompt structure sent to the LLM when validation detects errors in the generated code.

Table 3.2: Structure of a corrective prompt sent to the LLM during the validation feedback loop.

Component	**Content**
Task instruction	
Directive	Fix this Terraform Code. Output ONLY the corrected code with no explanations.
Terraform CLI errors	
Error 1	Missing required argument “machine_type”
Error 2	Reference to undeclared resource “google_compute_firewall.allow_ssh”
Original code	
Code to fix	`<original Terraform configuration>`

3.6 Threat Modeling

The project began with the idea of using formal threat models, expressed in the Meta Attack Language (MAL) [40], as the basis for an automated deception system. MAL files describe attack scenarios in a structured format that machines can interpret, including asset definitions, technique references, and causal links between attack steps. Consequently, the pipeline was carefully designed to process this type of input and transform it into deployable honeypot configurations. In this section, we explain the reasoning behind each stage of this transformation: the methods we use to parse and interpret the MAL input, how we categorize the identified threats, and the criteria we use to decide which techniques deserve a dedicated decoy.

3.6.1 Meta-Attack-Language

MAL is a domain-specific language developed within the SecuriCAD ecosystem [9] for modeling threat scenarios against enterprise infrastructure [40]. A MAL file defines assets (servers, databases, authentication systems), groups attack entities within those assets, and expresses causal relationships between attack steps using explicit operators (`->` and `+>`). This format was suitable for our purposes, providing a parseable structure that we could process using our pipeline modules. The system can directly construct an attack graph from the input owing to the clear connections between entities, which form the basis for the Betweenness Centrality analysis discussed later in this section. Without these connections, the pipeline would have to infer the attack sequence solely from the list of techniques, resulting in a less precise graph.

It is important to understand that MAL is not the only format for expressing threat intelligence. Industry standards, such as STIX/TAXII [6] are frequently used to share compromise indicators. However, STIX mainly focuses on observable artifacts, such as IP addresses, file hashes, and network signatures, rather than modeling an attacker's progression through an infrastructure. MAL's asset-centric and relationship-based structure aligns more naturally with the type of reasoning that our pipeline performs. Despite this, the modular architecture described in Section **??** was designed to accommodate an interchangeable parser; replacing the MAL parser with a STIX parser would require rewriting only one module, leaving the rest of the pipeline unchanged.

3.7 MITRE ATT&CK Classification

To classify the detected techniques and understand their role in an attack, we decided to rely on the MITRE ATT&CK framework as our classification core [21]. MITRE ATT&CK provides a comprehensive and publicly maintained attack taxonomy of offensive techniques organized by Kill Chain Phases (Initial Access, Credential Theft, Lateral Movement, Persistance, etc.), with each technique having an assigned univoque identifier (ex. T1003), name, metadata about interested platforms, and description of adversary behavior. This standardization was important for two reasons. First, it provides the pipeline with a common vocabulary that is consistent across different threat models: whether the input describes a credential dumping attack or an SSH

brute-force attempt, the system maps it to the same well-defined technique ID. Second, the Kill Chain phase hierarchy provides a natural ordering of attack severity that we use directly in the risk-scoring algorithm. Techniques associated with later phases (Exfiltration, Impact) receive higher weights than those in earlier phases (Discovery, Initial Access), reflecting their greater potential damage.

From a practical standpoint, MITRE ATT&CK also integrates well with the LLM generation phase. Because the framework is widely documented and referenced in the cybersecurity literature, our fine-tuned model already has a baseline understanding of what each technique entails. This implies that when the prompt includes a technique name and description from the MITRE database, the model can draw on its pre-training knowledge to generate more relevant honeypot configurations.

Table 3.3: Example classification output for a lateral movement MAL file. Each extracted technique is mapped to its Kill Chain phase and linked to the honeypot service selected for deployment.

Technique ID	**Name**	**Kill Chain Phase**	**Honeypot Service**
T1082	System Information Discovery	Discovery	SNMP agent with decoy system info
T1046	Network Service Scanning	Discovery	Open ports with banner emulation
T1003	OS Credential Dumping	Credential Access	SSH with fake `/etc/shadow`
T1021	Remote Services	Lateral Movement	SSH/RDP honeypot (Cowrie)
T1570	Lateral Tool Transfer	Lateral Movement	SMB share with decoy binaries

3.8 Resource Awareness with Betweenness Centrality

Not all techniques in a threat model are equally important from a defense perspective. Some techniques act as critical junctions in the attack progression; if an attacker cannot execute them, the entire attack path is disrupted. Others are peripheral steps that the attacker can easily bypass. We needed a way to determine which are crucial to stop a threat execution to prioritize where to place decoys accordingly. We chose Betweenness Centrality (BC) as the metric for this prioritization, following the approach proposed by Zambianco et al. [42].

By calculating the betweenness centrality, we could capture the structural importance of a technique in the Kill Chain: a high-risk technique that appears at the end of a single, isolated attack path is less valuable as a decoy target

than a medium-risk technique that connects multiple paths across different kill chain phases. This allows us to minimize the number of honeypots needed to create a defense infrastructure to counter an enterprise attack.

3.8.1 Coverage and Resource Constraints

The Betweenness Centrality ranking produces an ordered list of techniques, but the pipeline cannot deploy an unlimited number of decoys. Cloud resources cost money, and an excessive number of honeypots could actually raise suspicion; a network segment with more decoy machines than real ones would be easy for an attacker to identify as a trap.

We define two constraints that govern the decoy selection process. The first is a coverage threshold of 80%, which means that the algorithm stops selecting techniques once at least 80

The second constraint is a hard limit of five decoys. This cap prevents the pipeline from generating an unreasonable number of VMs, even in scenarios with many high-centrality techniques. A limit of five was chosen as a practical balance: in our experiments, three to five dedicated honeypots were sufficient to cover the critical attack paths in most threat models while keeping the cloud deployment manageable and cost-effective.

These constraints are configurable by the system administrator, allowing experienced security teams to adjust the coverage percentage or the maximum number of decoys based on their organization's budget and threat tolerance.

3.9 Deployment Platform

The last step in the development of our LLM-powered honeypot solution was the choice of the cloud platform provider and infrastructure as the code tool.

3.9.1 Terraform

The choice of Infrastructure as Code (IaC) tool directly affects the reliability of the entire generation pipeline because it determines the syntax rules that the LLM must learn and the validation mechanisms available to catch errors. We selected HashiCorp's Terraform [25] as the IaC backbone of our system.

Terraform uses a declarative language (HCL), in which the user specifies the desired end state of the infrastructure rather than the procedural steps to

reach it. This declarative nature simplifies the LLM task; the model only needs to produce a description of what the honeypot should look like, not a sequence of imperative commands to build it. Equally important for our pipeline is Terraform's built-in `validate` command, which performs a static analysis of the generated code and returns precise machine-readable error messages. This feature is the foundation of our iterative feedback loop, in which validation errors are fed back to the LLM for correction. Terraform is also cloud-agnostic: while our current implementation targets the Google Cloud Platform, the same HCL configurations can be adapted to AWS or Azure by changing only the provider declaration, thereby preserving the long-term portability of the system.

Alternative IaC tools do not satisfy these combined requirements. Ansible uses a YAML-based procedural model that lacks a built-in static validation step, making it unsuitable for the feedback loop. AWS cloud formation is tightly coupled to a single cloud provider. Pulumi uses general-purpose programming languages (Python, TypeScript), which would have required the LLM to generate full programs rather than simple key-value declarations, significantly increasing the error surface.

3.9.2 Google Cloud Platform

For the deployment target, we chose Google Cloud Platform (GCP) Compute Engine instances. GCP was selected as the deployment platform because the Google Cloud Terraform provider is mature and well-documented, reducing the likelihood of provider-side bugs in the generated configurations.

From a technical standpoint, GCP Compute Engine provides a straightforward VM model in which each instance is defined by a machine type, disk image, and startup script (three parameters that map directly to the variables our LLM generates). This simplicity minimizes the semantic gap between the model's output and the actual infrastructure, reducing the common source of deployment failures.

It is important to note that GCP is not a hard requirement of the system. Owing to its modular architecture and the use of Terraform, migrating to a different cloud provider would only require swapping the base template and deployer module, without retraining the LLM or modifying the classification logic.

3.10 Pipeline Workflow

We can describe the complete workflow of the deception pipeline in Figure 3.1. The system processes a threat model through six sequential stages, each building on the output of the previous stage. Figure3.1 presents a visual overview of this flow.

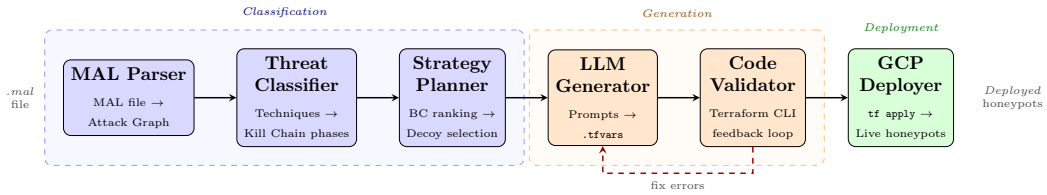

Figure 3.1: End-to-end pipeline workflow. The six stages are grouped into three functional phases: classification (blue), generation (orange), and deployment (green). The dashed arrow represents the iterative feedback loop between the Code Validator and the LLM Generator.

1. **MAL Parsing**: The pipeline ingests a Meta Attack Language file and constructs a directed graph of assets, attack steps, and their relationships.

2. **Threat Classification**: The parsed entities are mapped to the MITRE ATT&CK framework. Each technique is assigned a Kill Chain phase and a risk weight, and the overall scenario receives a composite risk score.

3. **Strategy Planning**: The attack graph is analyzed using Betweenness Centrality to identify the most critical techniques. The planner selects up to five decoy targets, covering at least 80% of the detected techniques, and determines the resource allocation for each.

4. **LLM Generation**: For each selected technique, the fine-tuned Llama 3.1 model generates a Terraform variable file containing machine specifications and a startup script tailored to the attack vector.

5. **Code Validation**: The generated code is post-processed to remove LLM artifacts, then validated through the Terraform CLI. If errors are detected, the code is sent back to the LLM for correction in an iterative feedback loop (up to three attempts).

6. **Deployment**: The validated Terraform configurations are applied to Google Cloud Platform, provisioning fully functional honeypot VMs that begin capturing attack traffic immediately upon startup.

This sequential workflow ensures that each stage can rely on validated inputs from the previous stage, thereby minimizing the propagation of errors through the pipeline. The entire process (from MAL file ingestion to live honeypot deployment) is executed without manual intervention, fulfilling our goal of a fully automated deception system.

Chapter 4

Dataset

In any system that relies on LLM-based generation, the training dataset limits the model's generation capabilities. This chapter outlines the two datasets created for our deception pipeline: one for fine-tuning the LLMs to generate honeypots and another for classifying threats using the MITRE ATT&CK framework. We begin by explaining why we chose to develop custom datasets, then describe the step-by-step construction process, and finally evaluate the resulting datasets in terms of technique coverage, service variety, and quality assurance.

4.1 Design

The effectiveness of our deception pipeline depends on the LLM's ability to generate realistic deployable honeypot configurations. This capability is entirely determined by the quality and specificity of the training data. Although several public resources for honeypot research exist, none of them met the requirements of our system, forcing us to design a new solution inspired by existing resources.

Existing honeypot datasets, such as those that aggregate logs from deployed honeypots, focus on **captured attack data** rather than on the **infrastructure configurations** required to deploy honeypots. Our pipeline requires the inverse: given an attack scenario, the model must produce the infrastructure-as-code (IaC) to deploy a matching decoy.

The few available configuration repositories (e.g., the "Awesome Honeypots" collection [26]) provide standalone deployment scripts and Docker Compose

configurations for individual honeypot utilities. These scripts vary widely in structure, target platform, and level of detail, with some consisting only of simple README instructions and others of complex multi-container files. This heterogeneity makes them unsuitable for direct use as fine-tuning data for a language model that must learn a consistent input-output mapping.

Furthermore, no existing resource links honeypot configurations to specific **MITRE ATT&CK techniques**. This mapping, which must select and generate decoys based on the threat classification output of the pipeline, is essential for the proposed system. Without the specific techniques tagged to the training data, it would not be possible for the model to associate correct defensive configurations for a specific attack.

These limitations motivated the creation of a dataset specifically designed for the task of *threat-aware honeypot IaC generation*, with the following design principles:

First, rather than generating complete Terraform HCL files (which would require the LLM to learn complex syntax, provider configurations, and resource dependencies), we chose to have the model generate only *variable definitions* (`tfvars`). These variables are then injected into a pre-validated Terraform template, ensuring syntactic correctness while allowing the LLM to focus on the semantic content: machine type, disk size, and most critically, the `startup_script` that configures the honeypot's deceptive services. Second, each training sample pairs an attack scenario prompt containing explicit MITRE technique IDs, a severity level, and a target service with the corresponding Terraform variables, teaching the model to map threat intelligence directly to infrastructure decisions. Third, the dataset prioritizes *diversity and quality* over volume. With 800 samples covering 42 MITRE techniques across more than eight service types, each entry was either manually picked from real-world honeypot architectures or generated under strict constraints and subsequently validated through the cleaning pipeline described in Section 4.2.

4.2 Construction Process and Evolution

The training dataset was developed using an iterative process that progressively improved both the quality and coverage of the samples.

4.2.1 Phase 1: Manual Creation

The initial dataset was constructed by manually selecting and adapting honeypot configurations from the **"Awesome Honeypots"** [26] public Github repository. For each configuration, we identified the MITRE ATT&CK techniques it was designed to detect, wrote an appropriate input prompt following the standardized format, and extracted the relevant Terraform variables. This phase produced approximately 115 high-quality seed samples, which established a solid baseline for the construction of the entire dataset. After initially fine-tuning and testing our chosen LLM with our newly created dataset, we encountered significant performance issues owing to both the model's size and the fine-tuning dataset. This prompted a shift in our initial approach regarding the model's capabilities. Instead of having the LLM attempt to create a complete Terraform script configuration that requires complex syntax knowledge, we opted for a template-based approach. The LLM will now generate only portions of a script using a pre-validated terraform template into which its output is injected. To address the dataset size limitation, we employed a synthetic data augmentation technique, as described in the next paragraph.

4.2.2 Phase 2: Synthetic Augmentation

To expand the dataset beyond that provided by manual curation alone, we employed a synthetic data augmentation strategy using Google Gemini [11] with a few-shot prompting approach. A strict prompt structure was used to ensure a consistent and valid output:

System Role:

```
You are an expert in Cyber Deception and Infrastructure as Code
(Terraform). Your task is to expand an existing dataset of honeypot
configurations used for fine-tuning a Large Language Model.
```

User Prompt:

```
I have a dataset containing pairs of security scenarios and their
corresponding Terraform configurations.
Task: Generate N new, unique JSONL entries following exactly
the same format and style as the examples provided below.
Constraints:
- Format: Each line must be a valid JSON object with two keys:
```

```
prompt (attack scenario with MITRE techniques, severity, and target
service) and response (Terraform HCL variables with startup_script).
- Variety: Generate diverse scenarios covering different protocols
and distinct MITRE techniques.
- Consistency: Use debian-cloud/debian-11 as the image and ensure
valid Bash startup scripts.
Input Examples: [3 examples from original dataset]
```

To offer the model stylistic and structural guidance, a selection of few-shot examples was meticulously chosen from manually curated samples. The augmentation process focused on addressing the underrepresented combinations of techniques and service types identified in the Phase 1 distribution, increasing the dataset from 115 to 800 samples.

4.2.3 Phase 3: Cleaning and Validation

Following the generation process, we applied a rigorous cleaning and preprocessing pipeline to ensure the integrity and consistency of the entire dataset. The pipeline processed all 614 samples from Phase 1 and additional synthetic entries, achieving a **100% retention rate** on the curated samples and removing or correcting invalid synthetic entries.

Preprocessing addressed several categories of issues:

- **Terraform syntax normalization**: Removal of deprecated resource references such as `google_security_center_source` that are not supported in recent provider versions.
- **Operator standardization**: Correction of verbose comparison operators (e.g., `COMPARISON_GREATER_THAN` → `COMPARISON_GT`) to match the current Terraform HCL syntax.
- **Timestamp normalization**: Replacement of dynamic `timestamp()` function calls with static timestamp values to prevent non-deterministic behavior during plan execution.
- **JSON validation**: Elimination of malformed JSONL entries produced during synthetic augmentation.
- **Script completeness**: Removal of samples containing incomplete or truncated startup scripts.

- **Deduplication**: Identification and removal of semantically duplicate entries to maximize training diversity.

4.3 Dataset Composition

The final training dataset comprises 800 samples designed to maximize the generation capabilities of the LLM. Each sample is composed of input-output pairs, where the input describes a specific attack scenario with associated MITRE ATT&CK techniques, and the output contains specific configuration variables (in `tfvars` format) and inline startup scripts for the scenario. These variables are designed to be applied to a pre-validated Terraform template, ensuring a syntactically correct deployment of the dynamically customized honeypot.

Figure 4.1 shows a representative selection of entries from the fine-tuning dataset, illustrating the diversity of attack scenarios and their corresponding Terraform outputs. Each entry pairs a prompt describing the attack context with real configuration variables generated for a specific scenario.

4.4 MITRE ATT&CK Coverage and Technique Distribution

The domain of our dataset contains 40 techniques from the MITRE ATT&CK Enterprise framework distributed across all major attack kill chain phases. This ensures that the trained models can be generalized to diverse threat scenarios.

The distribution of the techniques reflects real-world threat priorities, focusing on those that appear more often [20] and are distributed across the entire attack lifecycle, ensuring broad generalization. Key areas of coverage include **Credential Access** techniques such as Password Guessing (T1110) and LSASS Memory dumping (T1003.001); **Lateral Movement** methods involving RDP (T1021.001), SSH (T1021.004), and WMI (T1047); and **Initial Access** vectors such as Exploiting Public-Facing Applications (T1190). The dataset also heavily features **Discovery** tactics (e.g., File and Account Discovery) and **Persistence** mechanisms, including Scheduled Tasks and Registry Run Keys, reflecting realistic enterprise attacks.

Input (Prompt):

```
Network device SNMP reconnaissance attack implements 2 MITRE
ATT&CK techniques:  Network Device Configuration Gathering, Remote
System Discovery.  Attack severity:  standard.  Target services:
snmp.
```

Output (Response):

```
machine_type = "e2-standard-2"
image = "debian-cloud/debian-11"
disk_size = 50
startup_script = «EOT
#!/bin/bash
apt-get update -y
apt-get install -y snmpd snmp-mibs-downloader
...  EOT
```

Input (Prompt):

```
SQL injection with web shell deployment Attack implements 1 MITRE
ATT&CK techniques:  Exploit Public-Facing Application (T1190).
Attack severity:  urgent.  Target services:  web.
```

Output (Response):

```
machine_type = "e2-standard-8"
image = "debian-cloud/debian-11"
disk_size = 500
startup_script = «EOT
#!/bin/bash
apt-get update -y
apt-get install -y apache2 libapache2-mod-security2
modsecurity-crs php
...  EOT
```

Input (Prompt):

```
SCADA/ICS system compromise Attack implements 2 MITRE ATT&CK
techniques:  Windows Management Instrumentation (T1047), Exploit
Public-Facing Application (T1190).  Attack severity:  severe.
Target services:  scada.
```

Output (Response):

```
machine_type = "e2-standard-4"
image = "debian-cloud/debian-11"
disk_size = 100
startup_script = «EOT
#!/bin/bash
apt-get update -y
apt-get install -y rsyslog
...  EOT
```

Input (Prompt):

```
Database exploitation with credential extraction Attack implements
2 MITRE ATT&CK techniques:  Exploit Public-Facing Application
(T1190), LSASS Memory (T1003.001).  Attack severity:  critical.
Target services:  database.
```

Output (Response):

```
machine_type = "e2-standard-8"
image = "debian-cloud/debian-11"
disk_size = 500
startup_script = <<EOT
#!/bin/bash
apt-get update -y
apt-get install -y mysql-server
...  EOT
```

Input (Prompt):

```
FTP Data Exfiltration Campaign implements 2 MITRE ATT&CK
techniques:  Exfiltration Over Web Service (T1567.002),
Exfiltration Over C2 Channel (T1041).  Attack severity:  critical.
Target services:  ftp.
```

Output (Response):

```
machine_type = "e2-standard-4"
image = "debian-cloud/debian-11"
disk_size = 100
startup_script = <<EOT
#!/bin/bash
apt-get update -y
apt-get install -y vsftpd
...  EOT
```

Input (Prompt):

```
VoIP Command and Control Channel implements 2 MITRE ATT&CK
techniques:  Application Layer Protocol:  Web Protocols
(T1071.001), Exfiltration Over C2 Channel (T1041).  Attack
severity:  high.  Target services:  voip.
```

Output (Response):

```
machine_type = "e2-standard-4"
image = "debian-cloud/debian-11"
disk_size = 100
startup_script = <<EOT
#!/bin/bash
apt-get update -y
apt-get install -y asterisk
...  EOT
```

Input (Prompt):

```
IoT device exploitation and credential attacks Attack implements 1
MITRE ATT&CK techniques: Password Guessing (T1110.001). Attack
severity: moderate. Target services: iot.
```

Output (Response):

```
machine_type = "e2-medium"
image = "debian-cloud/debian-11"
disk_size = 20
startup_script = <<EOT
#!/bin/bash
apt-get update -y
apt-get install -y docker.io
... EOT
```

Input (Prompt):

```
Automated credential testing using leaked credentials Attack
implements 2 MITRE ATT&CK techniques: SSH (T1021.004), Password
Guessing (T1110.001). Attack severity: high. Target services:
ssh, web, ftp, rdp.
```

Output (Response):

```
machine_type = "e2-standard-2"
image = "debian-cloud/debian-11"
disk_size = 50
startup_script = <<EOT
#!/bin/bash
apt-get update -y
apt-get install -y xrdp openssh-server rsyslog fail2ban
... EOT
```

Figure 4.3: Representative entries from the fine-tuning dataset (Part 3 of 3). The `startup_script` installs service-specific packages for each honeypot type.

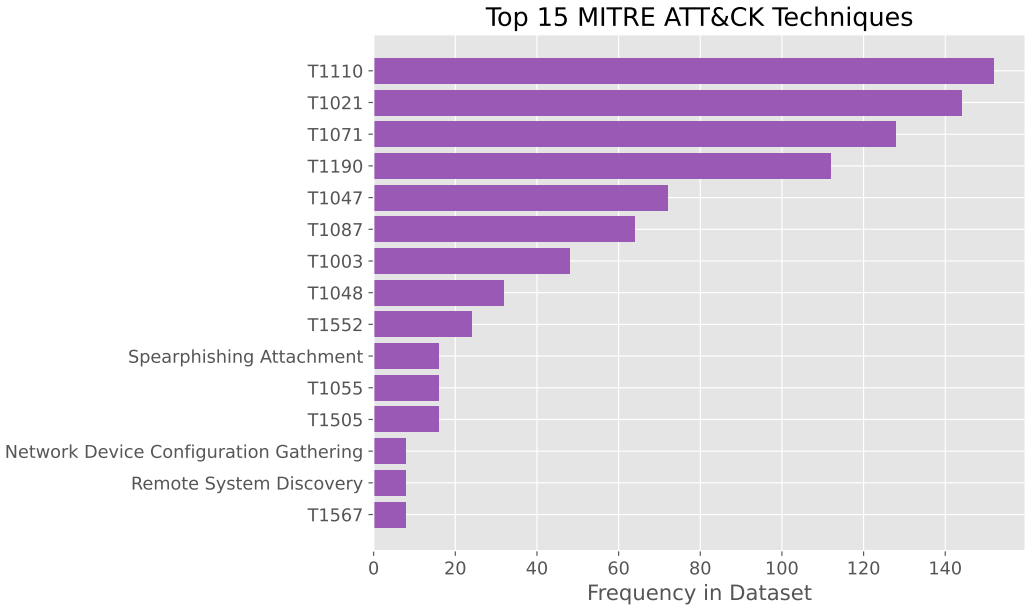

Figure 4.4: MITRE ATT&CK technique coverage across attack lifecycle phases, demonstrating comprehensive enterprise framework integration with 42 unique techniques.

4.5 Honeypot Type Distribution

To ensure that the LLM can work with different enterprise infrastructures, the 800 samples that compose the dataset vary for different service types. This distribution reflects common attack patterns, with Web Applications and RDP endpoints comprising 18% of the dataset. Other significant sections include the File Transfer Protocol(14.0%) and DNS services (13.0%). Other relevant mapped honeypot types are SSH endpoints (11.0%), SMB/Windows Shares (11.0%), Database Systems (9.0%), and selection of IoT device simulations (8.0%).

4.6 Dataset Sample Architecture

The dataset uses a *self-contained* honeypot architecture to ensure consistent deployment reliability and simplified management across the collection of all mapped scenarios.

In our architecture, all functional layers of the honeypot system are included within a single, self-contained compute instance—a `google_compute_instance`

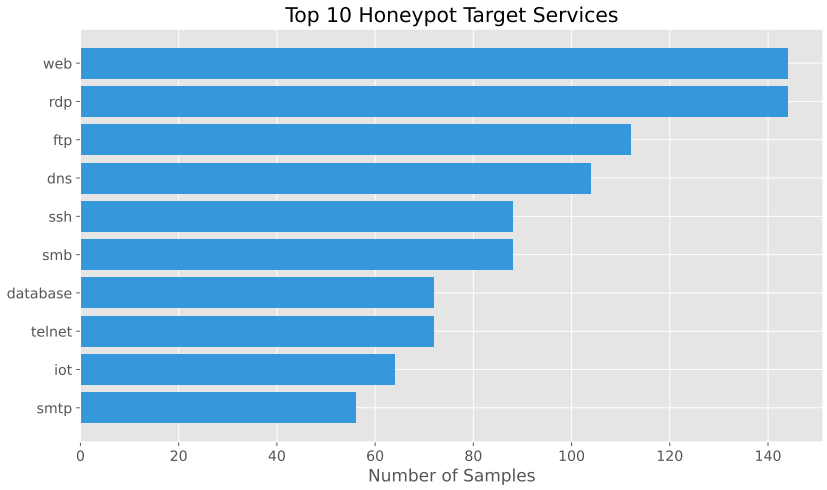

Figure 4.5: Distribution of the honeypot types across the 800 dataset samples, showing comprehensive coverage of organizational infrastructure components.

resource with a `tfvars` configuration and an inline startup script—structured in three distinct layers, as illustrated in Figure 4.6.

4.6.1 Layer 1: Exposed Services

The first layer defines network-facing services that simulate vulnerable targets to attract attackers. The startup script installs and configures real service implementations using standard system packages, ensuring authentic protocol-level behavior.

Across the 800 dataset samples, the most frequently deployed services are remote access endpoints, such as `xrdp` (34% of samples) and `openssh-server` (17%), which expose RDP and SSH interfaces with intentionally weak authentication. Web application scenarios representing 24% of the dataset deploy `apache2` with `mod_security` and `PHP` to simulate vulnerable servers with directory listings and login panels. DNS honeypots based on `bind9` accounted for 26% of the samples configured to respond to zone transfer requests and recursive queries. File transfer scenarios use `vsftpd` (10%) with anonymous upload-enabled and planted credentials, whereas database honeypots deploy `mysql-server` (9%) with databases populated with decoy records. The dataset also covers specialized protocols, including `snmpd` for SNMP reconnaissance, `postfix` for SMTP, `asterisk` for VoIP, and `mosquitto` for IoT/MQTT scenarios. Each service is configured to listen to its standard port (e.g., port 22 for

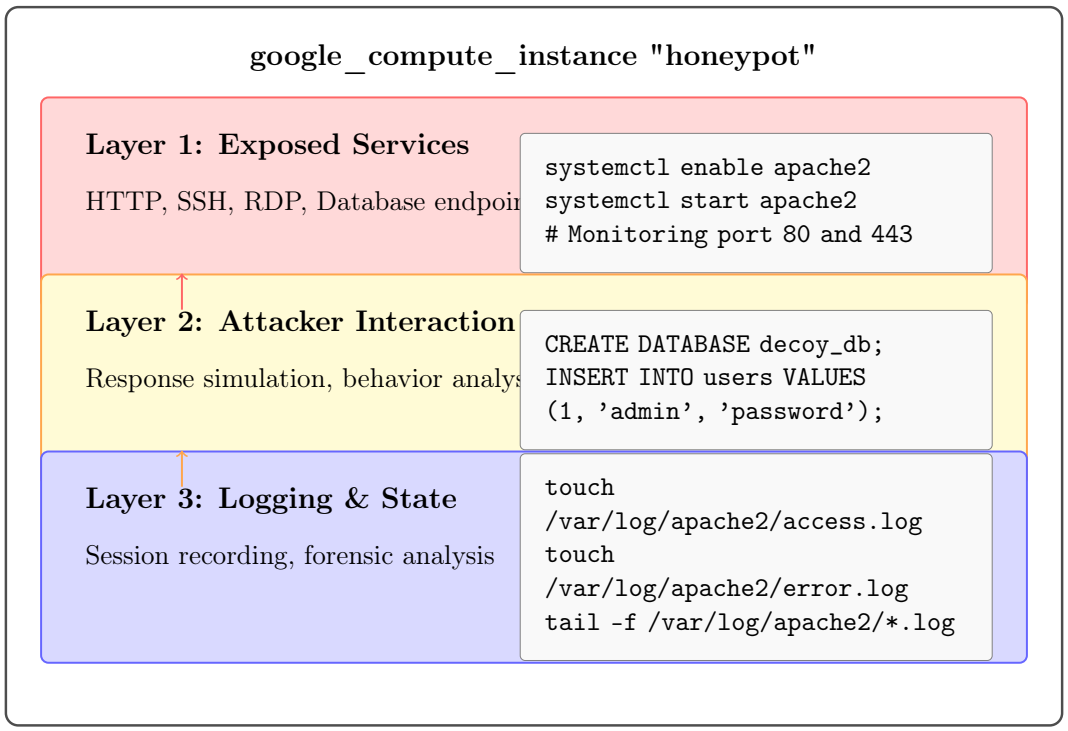

Figure 4.6: Monolithic honeypot architecture showing the three functional layers consolidated within a single compute instance: exposed services simulating vulnerable targets, modules handling attacker interaction, and local logging for forensic analysis.

SSH, 3389 for RDP, 80/443 for HTTP) to maximize realism and intercepted traffic.

Each service is configured to listen to its standard port (e.g., port 22 for SSH, 3389 for RDP, 80/443 for HTTP) to maximize realism and intercepted traffic.

4.6.2 Layer 2: Attacker Interaction and Deception Content

The middle layer enhances the honeypot by incorporating realistic crafted artifacts to encourage deeper engagement from attackers. Instead of offering empty services, startup scripts fill the environment with *deceptive content*, which closely resembles a genuine production system.

Analysis of the dataset reveals three main categories of deception content. Approximately 33% of the samples created fake user accounts via `useradd` and `chpasswd` with realistic usernames (e.g., `admin`, `developer`, and `backup`) and intentionally weak passwords, simulating credential harvesting targets. Approximately 9% of the samples deploy decoy databases—MySQL or PostgreSQL instances populated with fabricated records in tables such as `users`, `transactions`, and `employees`—designed to attract SQL injection attempts and data exfiltration. The most prevalent category, appearing in 63% of the samples, consists of planted files and directories. Paths such as `/opt/honeypot/`, `/var/www/html/`, and `/home/admin/` are populated with decoy configuration files, fake credentials, and simulated business data. This layer is critical for transforming a simple listening service into a convincing deception target that can capture post-exploitation behavior and lateral movement attempts.

4.6.3 Layer 3: Logging and State Monitoring

The innermost layer conducts thorough activity monitoring to capture all attacker interactions for forensic analysis. Each sample in the dataset includes a logging configuration structured across multiple complementary tools. The `auditd` daemon was installed in 33% of the samples, providing kernel-level system-call monitoring for detailed forensic analysis. Centralized logging via `rsyslog`, configured for 34% of the samples, aggregates logs from all the services into a structured format. Intrusion detection through `fail2ban` appears

in 17% of the samples, monitoring authentication failures and providing real-time alerting on brute-force attempts while deliberately maintaining permissive thresholds to avoid blocking attackers prematurely. Network-level visibility was achieved through `iptables` rules in 34% of the samples configured with logging targets to record all incoming connection attempts before accepting them.

Log files are stored locally within an instance, such as `/var/log/apache2/access.log` and `/var/log/auth. log`, ensuring a self-contained design while maintaining a comprehensive forensic trail of attacker activity.

4.6.4 Architectural Benefits

This self-contained approach offers several advantages for LLM-driven automated deployment. Each honeypot functions as an independent unit, eliminating the need for complex internode networking and orchestration. The lack of external dependencies ensures that deception capabilities remain operational, even during network degradation or partial infrastructure failures. By limiting the output to a single instance with an inline configuration, LLM can concentrate on crafting realistic deception content rather than managing the distributed system topology. Finally, each sample was fully reproducible: the startup script alone was sufficient to recreate the exact honeypot configuration from a clean OS image.

4.7 Severity Classification and Risk Modeling

To classify the severity of the various attack scenarios, we divided them into different severity tiers based on three main factors: the complexity of the attack, its potential impact on the enterprise environment, and the resources required for mitigation. This classification results in a balanced training set that is divided into three categories. **Medium Severity** scenarios, comprising approximately 24% of the dataset, focus on reconnaissance and discovery activities. **High Severity** cases, representing the largest portion at 43%, involve active exploitation and privilege escalation attempts. Finally, **Critical Severity** scenarios comprise 18% of the samples, simulating advanced persistent threats and immediate compromise risks that demand sophisticated defensive responses.

This distribution ensures balanced training across different threat levels, while providing an adequate representation of high-impact scenarios that require sophisticated defensive responses.

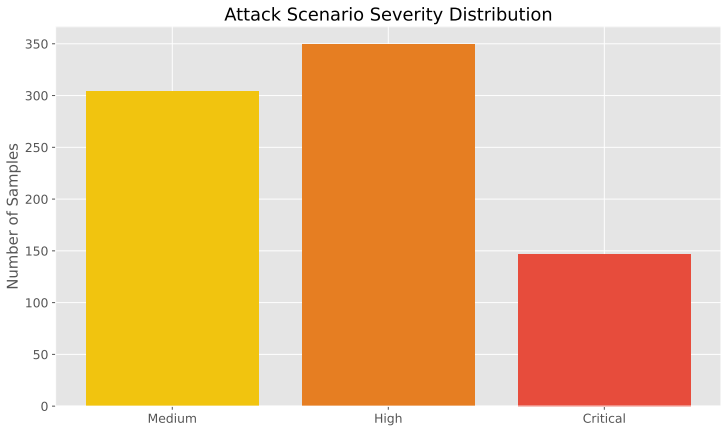

Figure 4.7: Attack severity classification showing balanced representation across threat levels for comprehensive model training.

4.8 Infrastructure as Code Integration

Each entry of the dataset includes complete Terraform infrastructure definitions with embedded startup scripts, providing full-stack honeypot deployment capabilities. The infrastructure leverages Google Cloud Platform components, such as computing instances, networking infrastructure, and storage buckets, along with security configurations, including firewall rules, service accounts, and identity and access management policies. The inline scripts, embedded within Terraform's `metadata_startup_script` blocks, handle the full lifecycle configuration of the honeypot: setting up application servers (such as Apache, MySQL, SSH, and RDP), populating the environment with realistic fake user accounts and databases, configuring logging services such as `rsyslog` and `fail2ban`, and hardening services to simulate authentic interaction patterns.

4.9 Dataset Quality Assurance

Before fine-tuning Llama-3.1-8b-Instruct, we validated the dataset to verify its full integrity and maximize the training results. This included checking

all the samples for syntax errors, verifying the technique assignments with the official MITRE documentation, testing of the inline scripts, and ensuring consistent patterns in severity classification. The result was a useful dataset that provides a strong baseline for training our LLM, enabling the system to generate high-quality, realistic honeypot samples across different scenarios.

4.9.1 Structural Validation

The first stage of validation ensures that each entry conforms to the expected format. All 800 samples were verified as valid JSONL entries, each containing exactly two keys: `prompt` (the attack scenario description) and `response` (the Terraform variable configuration). This structural conformance check confirmed a **100% pass rate** across the entire dataset, with no malformed or incomplete entries.

Within each response, we further verified the presence of four mandatory `tfvars` fields: `machine_type`, `image`, `disk_size`, and `startup_script`. All 800 samples contained these four fields, ensuring full compatibility with the Terraform template system described in Section 4.8.

4.9.2 MITRE ATT&CK Technique Validation

To verify the correctness of the technique assignments, we developed a dedicated validation script (`check_mitre_count.py`) that extracts all MITRE ATT&CK technique references from the prompt field and normalizes them against the official Enterprise framework identifiers. Listing 4.1 presents core extraction logic.

Listing 4.1: Core technique extraction and normalization from the validation script.

```
def extract_techniques(prompt):
    match = re.search(
        r"implements␣\d+␣MITRE␣ATT&CK␣techniques:␣"
        r"(.*?)\.␣Attack␣severity", prompt)
    if match:
        techniques_str = match.group(1)
        return [t.strip() for t in
                techniques_str.split(',')]
    return []
```

The validation confirmed that all 800 prompts reference at least one MITRE technique, yielding a total of **42 unique techniques** distributed across all major kill chain phases. Every technique identifier was cross-referenced against the official MITRE ATT&CK Enterprise matrix [20] to ensure the correctness of both the technique codes (e.g., T1190, T1021.004) and their human-readable descriptions.

4.9.3 Severity and Startup Script Validation

Each prompt was checked for the presence of a severity classification label. The validation confirmed that 100% of the samples contained an explicit severity assignment distributed across multiple tiers ranging from standard to critical, consistent with the classification described in Section 4.7.

Finally, the start-up scripts embedded in each response were validated for completeness. All 800 scripts contain a valid Bash shebang line (`#!/bin/bash`), at least one package installation command, and appropriately terminated the heredoc delimiters (`EOT` or `EOF`). No truncated or incomplete scripts were found, confirming that the cleaning pipeline described in Section 4.2 successfully eliminated all partial entries introduced during synthetic augmentation.

4.10 MITRE ATT&CK Classification Dataset

The threat classification component operates independently from the infrastructure generation system, utilizing a separate MITRE ATT&CK Enterprise dataset for attack analysis and strategic planning. While the LLM training dataset 4.1 is used to fine-tune the LLM on how to generate honeypot configurations, this classification dataset is used to provide context for the threats to help our system decide which type of honeypot to generate, based on the classification of a threat.

4.10.1 Dataset Specifications

The classification dataset provides a complete representation of the MITRE ATT&CK Enterprise framework [20], structured as a CSV file containing **656 entries**: 203 top-level techniques and 453 sub-techniques. Each entry includes eight fields: the technique `name`, its unique `id` (e.g., T1001, T1003.001), the official MITRE `url`, the target `platforms`, the associated `kill chain phases`,

a natural language `description`, the relevant `data sources` for detection, and specific `detection` guidance. Table 4.1 shows the representative entries from the dataset.

4.10.2 Dataset Coverage

The dataset includes all 14 tactical categories of the MITRE ATT&CK kill chain. The most common tactic is Defense Evasion, with 125 techniques. This shows the many ways in which attackers try to avoid being noticed. Other common categories are Resource Development (47 techniques), Credential Access (46), Reconnaissance (44), and Discovery (40). The dataset also includes less common but important tactics, such as Lateral Movement (16 techniques) and Initial Access (13 techniques). Some techniques fit into more than one phase of the kill chain. For example, 36 techniques are part of both Persistence and Privilege Escalation, showing that real attacks often involve many steps.

The dataset covers 10 different platforms. Windows has the most techniques (448), followed by macOS (331) and Linux (326). It also includes cloud and modern infrastructure platforms, such as IaaS (101 techniques), network devices (98), SaaS (63), and containers (47). This helps the classifier deal with threats to different types of enterprise systems.

Table 4.1: Sample entries from the MITRE ATT&CK classification dataset.

Name	**ID**	**Kill Chain Phase**	**Description**	**Data Sources**
Data Obfuscation	T1001	Command and Control	Adversaries may obfuscate C2 traffic using junk data, steganography, or protocol impersonation.	Network Traffic Content
OS Credential Dumping	T1003	Credential Access	Adversaries dump credentials to obtain login material, typically hashes or clear text passwords.	Command, Process, Windows Registry
Steganography	T1001.002	Command and Control	Adversaries use steganographic techniques to hide C2 traffic in digital messages or files.	Network Traffic Content

Chapter 5

Implementation

In this section, we detail the design and implementation of our experimental system. The deception pipeline follows a modular design, with components working together to generate a fully functional honeypot, specific to the threat described by a MAL file modeling a security attack in an enterprise scenario. All the components run locally on the host machine, providing an advantage for consumer and enterprise users, allowing to save resources for hosting. Another advantage is the customizability of our system: in this chapter we detail our architecture with the llama-3.1-Instruct Large Language Model, but the modularity of our proposal allows users to use other fine-tuned LLMs.

5.1 Threat Classification Components

The classification component receives a MAL file describing an attack scenario and outputs a deployment strategy that specificies which MITRE ATT&CK techniques should be the target for the decoys, how many virtual machines to generate, and what resources to allocate to each one. The classification component integrates 4 submodules that operate sequentially: the MAL Parser, the Threat Classifier, the Attack Graph module and the Strategy Planner.

5.1.1 MAL file parser

The MAL Parser module is the first point in the deception pipeline. It processes Meta Attack Language files and structures their contained information into a graph representation that will be used by the higher-level components. The parsing process begins by loading the MAL file that is to be classified

and splitting it into individual lines. The parser then extracts global metadata and identifies asset declarations to group-related entities. When a block describing an attack is detected (identified in the MAL syntax with "|"), the parser reads the entire context block, identifies specific MITRE techniques and relationships between entities (marked by the -> or +> operators) using regular expressions. Finally, the parsed components are merged to construct a MALGraph object representing the attack scenario.

Listing 5.1: Core parsing loop for MAL file processing

```
while i < len(lines):
    line = lines[i].strip()

    if line.startswith('asset␣'):
        # Save previous asset and start new group
        If current_asset and current_entity_list:
            entities[current_asset] = current_entity_list
        current_asset = self._extract_asset_name(line)
        current_entity_list = []

    elif line.startswith('|'):
        # Collect entity context block
        entity_block = [line]
        j = i + 1
        whereas j < len(lines).
            next_line = lines[j].strip()
            if next_line.startswith(('|', 'asset␣', '
               category␣')):
                break
            if next_line:
                entity_block.append(next_line)
            j += 1

        # Parse entity with full context
        entity = self._parse_entity_with_context(
            line, '\n'.join(entity_block), current_asset
        )
        if entity:
            current_entity_list.append(entity)
        i = j - 1
    i += 1
```

5.1.2 Classifier

The classifier module analyzes threats using two primary classes. The **MITREIntegration** class extracts the technique IDs, names, and kill chain stages during initialization by importing the categorization dataset described in Section 4.10. It exposes two methods (`get_category() and get_info()`) to map every technique to its primary kill chain phase (ex. T1003 to Credential Access), providing contextual information that the subsequent module requires.

The **ThreatClassifier** uses the mapping to predict the potential impact of an attack on an infrastructure. It extracts every MITRE technique from the processed *MALEntity* objects and assigns each one a severity weight based on its kill chain phase. A weighted average of these severities is used to obtain the final risk score, which ranges from 1 to 10. If the attack uses a large number of different approaches, an extra complexity bonus is added.

Kill Chain Phase	**Risk Weight**
Initial Access	3.0
Execution	4.0
Discovery	4.0
Persistence	5.0
Defense Evasion	5.5
Privilege Escalation	6.0
Command and Control	6.0
Collection	6.5
Credential Access	7.0
Lateral Movement	7.5
Exfiltration	8.5
Impact	9.0

Table 5.1: Risk weights assigned to each MITRE ATT&CK kill chain phase. Later and more dangerous phases receive higher weights.

5.1.3 Attack Graph and Betweenness Centrality

The attack graph module constructs a directed graph that represents the various attack phases and computes the Betweenness Centrality between each node to detect which are critical for optimal decoy placement.

The graph is constructed using the NetworkX library from the parsed MAL entities. Each node represents a `(host, technique)` combination, connecting a particular MITRE ATT&CK technique to its targeted asset. For exam-

ple, a node `(AuthenticationSystem, T1110)` indicates a brute-force attack directed at the authentication system. The edges represent the progression between attack steps: when a MAL entity defines a connection to another entity (using the `->` or `+>` operators), the graph creates directed edges from each technique of the source entity to each technique of the target within the same asset context. In cases where the MAL file does not specify explicit relationships, resulting in no edges, the module uses default values based on the importance in the kill chain. Here, techniques are organized by their MITRE kill chain phase, and edges are added between successive phases in the typical order (e.g., Initial Access → Execution → Persistence → Lateral Movement).

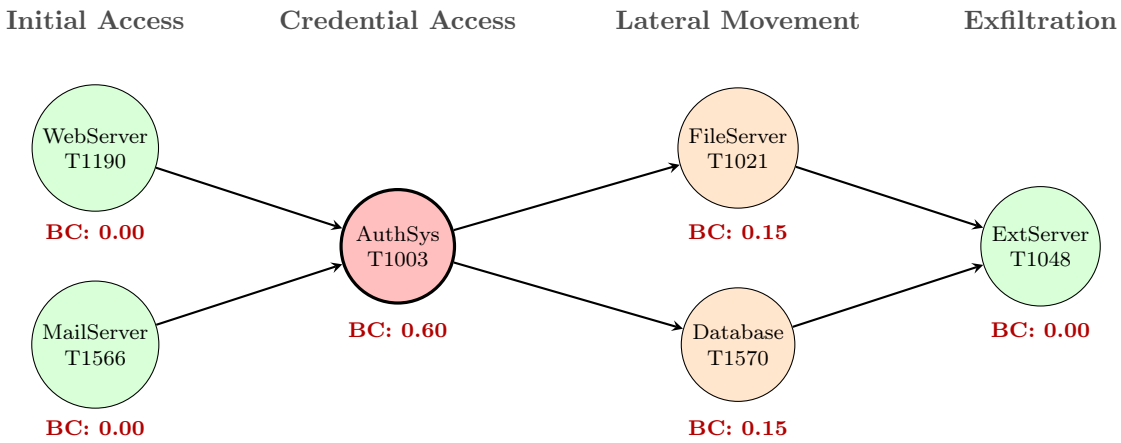

Figure 5.1: Example attack graph across four kill chain phases. Vertex color indicates BC score: red (high), orange (medium), and green (low). The `AuthSys, T1003` vertex is the bottleneck with BC = 0.60, making it the highest-priority decoy target.

Once a graph is constructed, the module computes the Betweenness Centrality (BC) score for every vertex. BC indicates the vertices that are on the shortest path between other vertices, meaning that a particular technique with a high BC score is a "bottleneck" for the attack.
The formula used is:

$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{5.1}$$

where σ_{st} is the number of shortest paths from s to t, and $\sigma_{st}(v)$ is the number of paths passing through v.

The resulting BC scores are organized by technique ID and forwarded to the Strategy Planner, which utilizes them to prioritize techniques and identify

the most strategically important targets for deploying decoys.

5.1.4 Strategy Planner

The strategy planner is the final component of the classification module. It takes all the information processed by the previous components (parsed techniques, classification results, and attack graph with the calculated scores for betweenness centrality) and prepares a strategy that will be used by the next components to generate appropriate resource-aware countermeasures for the attacks.

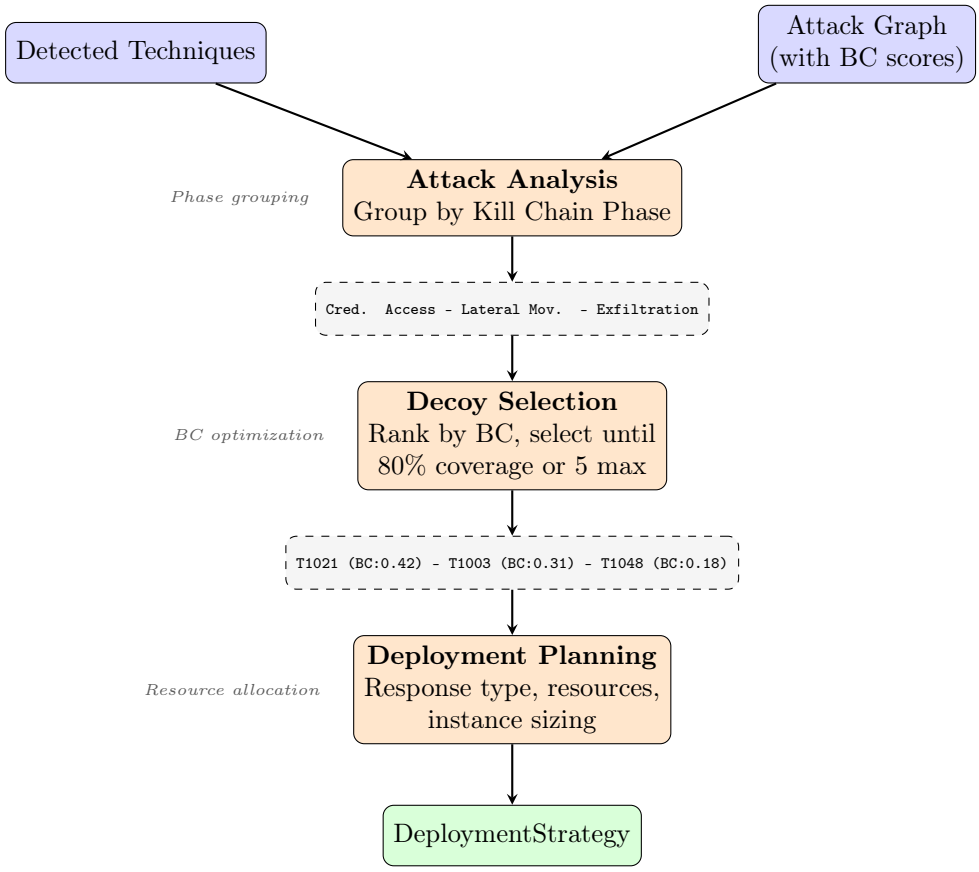

Figure 5.2: Strategy Planner workflow: from detected techniques and attack graph to a concrete deployment strategy.

The initial step involves analyzing the context of the attack. The planner categorizes the detected techniques according to their MITRE ATT&CK kill chain phase, such as Initial Access, Lateral Movement, and Exfiltration, and reconstructs the **attack progression** (the ordered sequence of phases present in the scenario). This process reveals how far the attacker has advanced through

the kill chain by reconstructing the **attack progression**, an ordered sequence of phases present in the scenario.

After the creation of an Attack Graph by the previous components, the planner utilizes the betweenness centrality scores calculated by the Attack Graph module to identify which techniques should be targeted by decoys. Each technique is evaluated based on the highest BC value among its corresponding vertices in the graph. This approach ensures that the most critical techniques (bottlenecks) in the attack progression are prioritized for decoy placement.

The selection algorithm ranks the techniques by their BC scores in descending order and iteratively selects decoys until one of two stopping conditions is met: either 80% of all detected techniques are covered, or a maximum of five decoys are chosen. Coverage extends beyond directly selected techniques by including related sub-techniques (e.g., selecting T1021 also covers T1021.001 and T1021.002), maximizing the attack surface coverage with a minimal number of deployed resources. Finally, the planner consolidates the analysis and decoy selection into a definitive deployment strategy. The response type is dictated by the risk level: a score of 6.0 or higher initiates a full `honeypot` deployment, whereas lower scores lead to the use of lightweight `monitoring` resources. The deployment priority is normalized from the risk score to a $[0, 1]$ range, enabling the ordering of concurrent deployments based on urgency.

Resource allocation uses a linear-scaling model based on the number of detected techniques. Starting with a base configuration of 2 vCPUs, 4GB of RAM, and 20GB of storage, each additional technique increases the allocation by a factor of $1.0 + (n - 1) \times 0.2$, where n represents the technique count. High-risk scenarios (risk $\geq$ 7.0) are assigned `e2-standard-2` instances, whereas lower-risk scenarios utilize the smaller `e2-medium` type.

The resulting `DeploymentStrategy` object is then forwarded to the Generation Components, which employ the selected techniques and configuration to generate the corresponding Terraform code.

5.2 Honeypot Generation Components

The generation component receives the deployment strategy produced by the classifier and uses a fine-tuned Large Language Model to produce a deployment-ready Terraform configuration.

5.2.1 Fine-Tuning Process

We fine-tuned the base Llama-3.1-8b-Instruct model using the dataset described in 4.6 with the aid of **LLaMa-Factory** [44], an open source framework that unifies a variety of efficient fine-tuning methods through scalable modules, enabling the fine-tuning of LLMs with minimal resources and high throughput. To run the fine-tuning, we used an NVIDIA A100 GPU with 40GB of memory. After considering the resource constraints, we adopted **QLoRa**(Quantized Low-Rank Adaptation), an efficient technique that combines 8-bit model quantization (via BitsAndBytes) with LoRa adapter layers, reducing memory usage during training and maintaining the quality of the tuned model by allowing the whole 8-billion parameters model to fit in a single graphics accelerator. We configured the LoRA with a rank of 16, an alpha scaling factor of 32, and a dropout rate of 0.05, applying adapters to all linear layers within the transformer architecture. We utilized a 10% split of the original dataset for evaluation during the training process. To tokenize the input sequences, we employed the LLAMA3 chat template, which has a maximum context length of 4096 tokens, ensuring compatibility with the instruction-based tuning format of the base model. The training process was set up for three epochs with a batch size of 2 and eight gradient accumulation steps, resulting in an effective batch size of 16. The learning rate was set at 2×10^{-5} with a 10

5.2.2 LLM IaC code Generator

The LLM Generator is responsible for producing Terraform configurations using a fine-tuned LLM (llama3.1 [30]) to generate context-aware honeypots based on the detected attack techniques. The component combines three essential components to create a functional honeypot: a multi-VM generation logic that generates differently tuned instances for each technique identified by the Betweenness Centrality optimization, LLM-generated `tfvars` parameters that customize the configuration, and a pre-validated Terraform template to ensure correct syntax.

The generation process begins by constructing a structured prompt using the extracted MITRE techniques, which is then sent to the fine-tuned LLM to generate the **tfvars** IaC code. The configurations were extracted using regular expressions. If the Betweenness Centrality optimization selects multiple deploys, a dedicated prompt is sent for each technique to generate different

Figure 5.3: Training and Validation loss curves for the three training epochs. The model shows rapid convergence during the first epoch, with the training loss dropping from 1.14 to 0.28. By the end of the second epoch, the loss stabilizes below 0.07, ultimately reaching a final training loss of 0.045 at step 130. The validation loss exhibits a consistent downward trend, decreasing from 0.221 at step 50 to 0.061 at step 135. The total training time on the A100 GPU was approximately 11.5 minutes for 135 optimization steps.

google_compute_instance resources with appropriate tags. Finally, the parsed parameters were used to populate the Terraform template. The resulting configuration was subsequently post-processed to ensure syntactical validity.

Prompt Format

The prompt structure matches the format used during model training:

Listing 5.2: Example LLM prompt for honeypot generation

```
Monitoring security implements three MITRE techniques:
Web Services (T1583.006), File and Directory Discovery
(T1083), Data from Local System (T1005).
Target: monitoring. Requires monitoring.
```

Output Format

The model responds with Terraform variable definitions that specify the machine type, disk size, and a specialized startup script containing the honeypot configuration commands, thereby ensuring optimal coverage of the attack

Listing 5.3: Example LLM prompt for honeypot generation

```
machine_type = "e2-standard-2"
image = "debian-cloud/debian-11"
disk_size = 20
startup_script = <<EOT
#!/bin/bash
# SSH Honeypot Setup - Cowrie Installation
apt-get update -y
...
EOT
```

surface.

5.2.3 Post-Processing

The Post-Processor module cleans the raw LLM output before the validation process. Given the constraints of the 8B-parameter model, the generated text often includes artifacts that could lead to Terraform parsing errors. The `TerraformPostProcessor` class employs a sequence of four regex-based transformations via a single `process()` method.

The first transformation removes markdown artifacts from the output. The LLM frequently encapsulates its response in markdown code fences (e.g., ```` ```terraform ... ``` ````) and adds conversational sections like *"Here's the configuration:"*. These elements are removed to isolate the pure HCL code.

The second transformation deals with the Bash variable escaping within Terraform heredoc blocks. Heredocs (`<<EOT ... EOT`) are used to embed startup scripts that configure the honeypot. Within these blocks, Bash variables such as `${PACKAGE}` might be misinterpreted by Terraform as interpolation expressions, leading to parsing errors. The post-processor corrects this by doubling the dollar sign (`${PACKAGE}` → `$${PACKAGE}`), ensuring that Terraform passes the variables to the shell unchanged.

The third transformation eliminates hallucinated commentary. The model occasionally generates explanatory natural language text before the `#!/bin/bash` shebang line inside heredoc blocks. As a shell script must begin with the shebang to be valid, any content preceding it within a heredoc is automatically removed.

Finally, the fourth transformation standardizes resource names to align

with GCP naming conventions. All resource names are converted to lowercase, and special characters, such as dots and underscores, are replaced with hyphens. The post-processor also removes duplicate `provider "google"` blocks that the LLM sometimes generates, keeping only the first instance.

Listing 5.4: Post-Processor transformation example: raw LLM output (top) and cleaned result (bottom)

```
# Before post-processing (raw LLM output):
```terraform
Here's the Terraform configuration:
name = "Honeypot_T1003.001"
startup_script = <<EOT
This script sets up a credential honeypot.
#!/bin/bash
apt-get install -y ${PACKAGE}
EOT
```

# After post-processing (cleaned output):
name = "honeypot-t1003-001"
startup_script = <<EOT
#!/bin/bash
apt-get install -y $${PACKAGE}
EOT
```

5.2.4 Validation and Feedback loop

The Code Validator ensures that the LLM-generated Terraform configurations are syntactically correct and deployable on the Google Cloud Platform. Recognizing the constraints of our model and the risk of syntax or configuration errors in a generated honeypot, we employed this component to verify the model output. This verification is conducted using Terraform alongside an iterative LLM-based feedback loop.

The validation system consists of two modules. The `TerraformValidator` acts as a high-level orchestrator: it first replaces common LLM-generated placeholders (such as `my-project` or `your-project`) with the actual user configuration values (project ID, region, and zone); it then fixes invalid characters

in resource names (for example, converting MITRE IDs such as `T1134.001` to `T1134_001` to comply with Terraform naming rules); and finally, it delegates the validation to the `TerraformCLIValidator`.

The `TerraformCLIValidator` performs configuration syntax verification by creating a temporary directory containing the generated Terraform code (`main.tf` with a specific provider configuration file (`provider.tf` declaring the Google Cloud Platform provider). The validator then uses the `terraform init` command to download the required plugins for the provider and `terraform validate` to check the configuration for correct syntax.

If validation fails, the module employs an iterative loop to rectify any errors and sends the LLM the errors found in the CLI, combined with the originally generated Terraform code.

Listing 5.5: Corrective prompt sent to the LLM during the fix loop

```
Please fix the following Terraform code: Output ONLY the
   corrected
code, no explanations.

ERRORS:
- Missing required argument "machine_type"
- Reference to undeclared resource

CODE TO FIX:
<original terraform code>

CORRECTED CODE:
```

The LLM's response is sanitized and validated once more through the same CLI pipeline. This process can be repeated up to three times, halting as soon as a valid configuration is achieved, or if the LLM fails to generate a significantly different output.

5.3 Deployment

The final step of the pipeline is the deployment module. By using the the validated Terraform configuration produced by the Code Validator it it provisions the corresponding honeypot infrastructure on Google Cloud Platform.

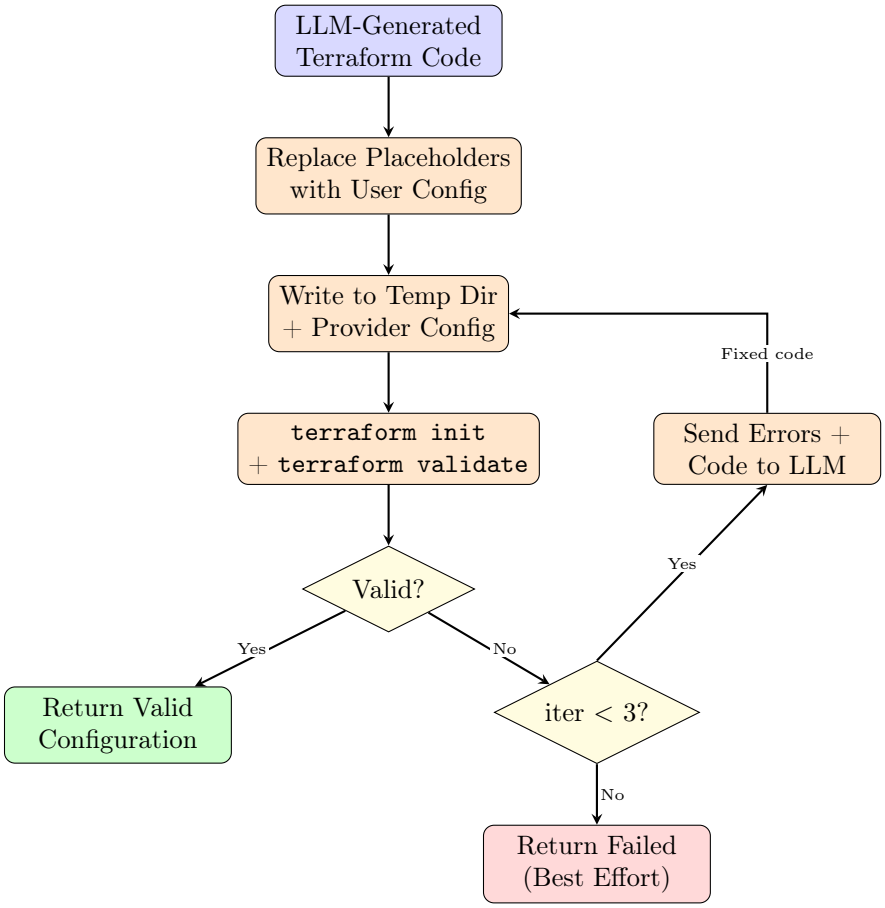

Figure 5.4: Code Validator feedback loop: configurations are validated through Terraform CLI and, if errors are found, sent back to the LLM for correction up to 3 times.

This module is executed by the `GCPDeployer` class, which oversees the entire process from authentication to resource provisioning.

5.3.1 Prerequisites and Authentication

Before initiating any deployment, the system ensures that the `terraform` and `gcloud` command-line tools are installed and accessible. It then retrieves the active GCP project ID using `gcloud config get-value project` to confirm that the target project is set up. If any of these checks fail, the deployment is stopped with a detailed error message.

Authentication is managed through a GCP service account. The deployer searches for a JSON key file at a specified path and, if located, sets the correct `GOOGLE_APPLICATION_CREDENTIALS` environment variable to the file path. As

a backup, the system checks for Application Default Credentials in the user's local `gcloud` configuration.

After the code is validated, the module executes the following Terraform commands in sequence:

1. `terraform init`: to donwload the required provider plugins for GCP;
2. `terraform plan`: to generate an execution plan.
3. `terraform apply`: deploys the IaC configuration to Google Cloud Platform.

Table 5.2: Fine-tuning hyperparameters

Parameter	**Value**
Model & Method	
Base model	Llama-3.1-8B-Instruct
Fine-tuning method	QLoRA (SFT)
Quantization	8-bit (BitsAndBytes)
LoRA Configuration	
Rank (r)	16
Alpha (α)	32
Dropout	0.05
Target modules	All linear layers
Training	
Epochs	3
Batch size (per device)	2
Gradient accumulation	8 steps
Effective batch size	16
Learning rate	2×10^{-5}
LR scheduler	Cosine annealing
Warmup ratio	10%
Precision	bf16
Dataset	
Training samples	720
Validation samples	80
Max sequence length	4096 tokens
Hardware	
GPU	1× NVIDIA A100 40 GB
Attention	FlashAttention-2
Results	
Total training steps	135
Final training loss	0.045
Final validation loss	0.061
Training runtime	∼11.5 minutes

Chapter 6

Validation

In this chapter, we present the performance evaluation of our experimental system. The validation phase is based on how the implementation described in 5 can efficiently generate honeypots for our cyber deception approach.

6.1 Experimental Setup

To conduct our test, we used a VM-instance of Google Cloud Compute using a Debian 11 image for the Operating System, a 12 Core CPU with 85GB of RAM memory, and an NVIDIA A100 40GB GPU as a graphics accelerator. We used the Remote-SSH extension in Visual Studio Code to connect and operate on the remote virtual machine.

6.2 Methodology

We ran the entire deception pipeline on a set of five MAL files describing different categories of cyberattacks from the MITRE ATT&CK Framework. For every test, we validated the correct terraform syntax, used Metasploit INSERT CITE to test the deception capabilities of the single honeypot, and gathered performance metrics. We tested the following attack scenarios:

- Credential Access: Password Guessing (T1110.001)
- Lateral Movement (T1018, T1021.001, T1570, T1068, T1083)
- DNS Exfiltration (1048.003)

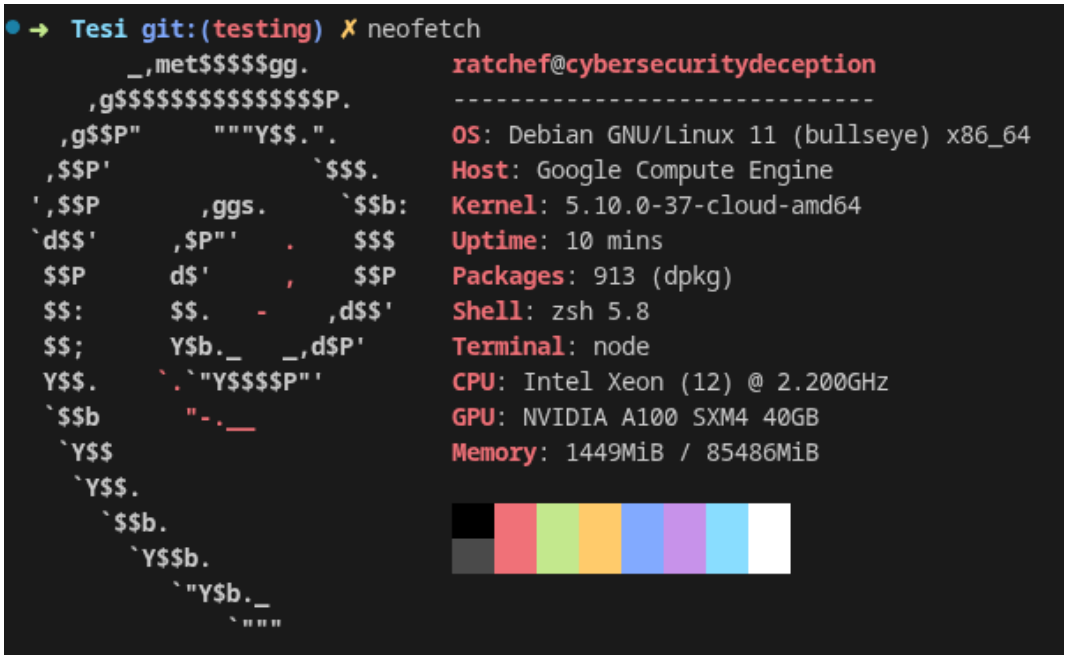

Figure 6.1: Experimental Setup System Specifications

- System and Network Discovery (T1082, T1016, T1033, T1518)
- Persistence (T1105, T1547.001, T1068, T1543.003, T1083)

For every generated decoy, we validated the terraform syntax using `terraform validate` to ensure that the LLM output was correctly formatted and deployable without manual intervention. After successfully deploying to the Google Cloud Platform, we tested the honeypots using the Metasploit Framework, launching attacks with the `msfconsole` interface targeting the specific MITRE techniques for which the honeypot was generated (ex. executing the `auxiliary/scanner/ssh/ssh_login` module against the Credential Access honeypot or SMB enumeration modules for Lateral Movement targets). We then verified that the honeypot successfully recorded the interaction by inspecting local system logs (for example, `/var/log/auth.log` and application-specific logs) on the VM generated via SSH. Finally, we gathered the execution time metrics for each stage of the pipeline to assess the system's performance overhead.

Listing 6.1: Lateral Movement MAL file

```
#id: lateral_movement
#version: 1.0.0

category System {
  asset Host {
    | compromise
      +> systemInfo

    | systemInfo
      user info: System Information Discovery T1082
      +> networkScan

    | networkScan
      user info: Network Service Scanning T1046
      +> credentialAccess

    | credentialAccess
      user info: Credential Dumping T1003
      +> lateralMovement

    | lateralMovement
      user info: Remote Services T1021
  }
}

associations {
  Host [client] 1 <-- Connection --> 1 [server] Host
}
```

6.3 Decoy Results

The outputs generated during our test managed to achieve a 100% success rate in the deployment, signifying that all the decoys contain correct terraform syntax.

6.3.1 Example Generated Configurations

The following examples show the actual startup scripts generated by the LLM specialized for different MITRE techniques.

SSH Honeypot for Credential Access (T1110.001)

For password-guessing attacks, the LLM generates an SSH honeypot using Cowrie with decoy files:

Listing 6.2: Startup script for SSH honeypot targeting T1110.001

```
#!/bin/bash
# SSH Honeypot Setup - Cowrie Installation
apt-get update -y
apt-get install -y python3 python3-pip git

# Install Cowrie
git clone https://github.com/cowrie/cowrie.git
pip3 install -r cowrie/requirements.txt

# Create fake filesystem
mkdir -p honeyfs/etc honeyfs/root
echo "root:x:0:0:root:/root:/bin/bash" > honeyfs/etc/passwd

# Create decoy files to entice attackers
mkdir -p /var/www/html/uploads
echo "user,password,email" > /etc/users.csv
mkdir -p /home/admin/.ssh

echo "SSH␣Honeypot␣ready␣-␣monitoring␣port␣22"
```

DNS Honeypot for Exfiltration (T1048.003)

For DNS-based data exfiltration, the LLM generates a BIND9 honeypot with extensive logging and iptable rules:

Listing 6.3: Startup script for DNS honeypot targeting T1048.003

```
#!/bin/bash
# DNS Honeypot Setup - Bind9 with Logging
apt-get install -y bind9 bind9utils

# Configure logging
logging {
    channel query_log {
        file "/var/log/bind/query.log";
        print-time yes;
    };
    category queries { query_log; };
};

# Log outbound connections on exfiltration ports
iptables -A OUTPUT -p tcp --dport 53 -j LOG --log-prefix "DNS_OUTBOUND: "
iptables -A OUTPUT -p tcp --dport 4444 -j LOG --log-prefix "SHELL_OUTBOUNI

echo "DNS Honeypot ready - monitoring port 53"
```

6.3.2 Attack Detection Results

Within minutes of deployment, the honeypots began capturing real attack attempts from the automated scanners. Table 6.1 shows the SSH brute-force attempts logged on the Credential Access honeypot (T1110.001).

The following log excerpt from `/var/log/auth.log` demonstrates the honeypot's detection capabilities:

Listing 6.4: SSH brute force attempts logged by the honeypot

```
Jan 30 17:05:55 honeypot-t1110-001-7569 sshd[18074]:
  Disconnected from authenticating user root 91.224.92.54 [preauth]
Jan 30 17:11:56 honeypot-t1110-001-7569 sshd[18132]:
  Disconnected from authenticating user root 45.148.10.147 [preauth]
Jan 30 17:18:10 honeypot-t1110-001-7569 sshd[18217]:
```

Table 6.1: SSH brute force attempts captured by honeypot within 30 minutes of deployment

Timestamp	Attacker IP	Target User
17:05:55	91.224.92.54	root
17:11:56	45.148.10.147	root
17:18:10	45.148.10.157	root
17:24:21	91.224.92.190	root
17:30:38	195.178.110.15	root
17:31:02	34.42.81.231	root (Metasploit test)

```
  Disconnected from authenticating user root 45.148.10.157 [preauth]
Jan 30 17:24:21 honeypot-t1110-001-7569 sshd[18283]:
  Disconnected from authenticating user root 91.224.92.190 [preauth]
Jan 30 17:31:02 honeypot-t1110-001-7569 sshd[18351]:
  Connection closed by authenticating user root 34.42.81.231 [preauth]
```

6.4 Performance Results

Table 6.2 presents the execution times for each component of the deception pipeline across the various tested attack scenarios. Timing metrics provide critical insights into the computational bottlenecks of LLM-driven automated defense systems.

The **MAL Parse** and **Classification** components execute in ultralow, sub-millisecond timeframes. This is expected, as these stages involve deterministic, localized operations (parsing XML/JSON structures and regex-based string matching) that are not resource-intensive.

The **Strategy Planning** stage shows that the Betweenness Centrality computation requires approximately 65ms of execution time per technique. Even for the most complex graphs containing dozens of nodes, the performance of the `networkx` library ensures that graph calculations remain negligible relative to the overall pipeline execution time.

The **LLM Generation Component** is the most resource-intensive and time-consuming task, requiring a time range between 38.2s and 309.0s depending on the complexity of the prompt and the number of required honeypots. Unlike the deterministic stages, text generation is determined by the model's auto-regressive token generation speed (measured at 2.77 tokens per second during evaluation). For complex attack vectors such as *System and Network*

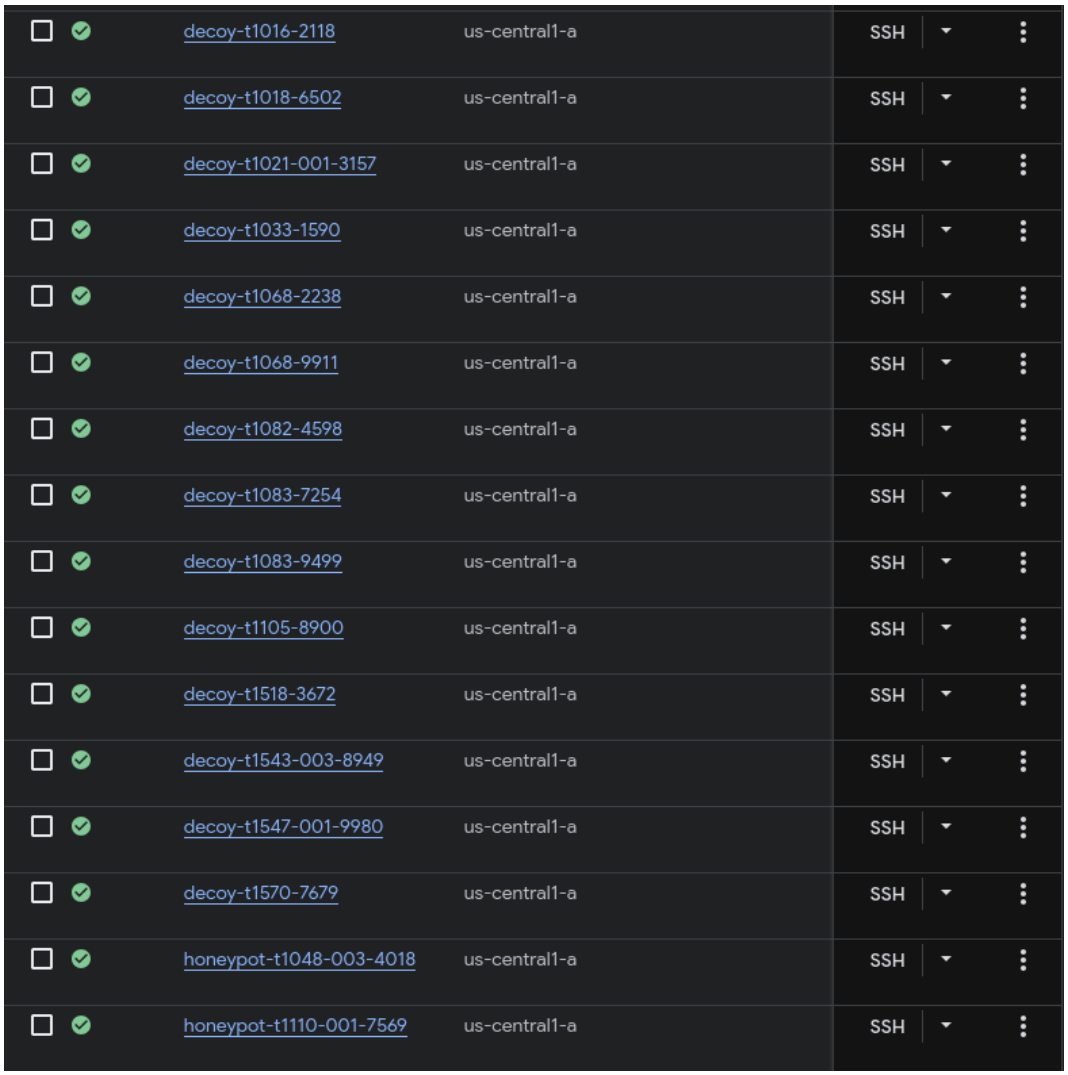

Figure 6.2: Decoys Deployed in Google Cloud Platform

Discovery (T1082+), where the pipeline determined that four distinct VMs were necessary, the LLM was queried multiple times to generate independent Terraform module files and startup scripts. The nonlinear scaling of time (309 s for four VMs versus 38 s for one VM) is due to the increasing context window size: as previous scripts are appended to the context to maintain consistency, the self-attention mechanism of the LLM requires quadratically more computation, slowing down generation.

The **Deployment** phase (Terraform `apply`) maintains a relatively consistent execution time across all tests (typically around 27 s). This consistency suggests that provisioning a Google Cloud Compute instance has a fixed baseline overhead, regardless of the VM's specific OS image or the complexity of the `metadata_startup_script` string being injected. The sole exception is the Persistence scenario (16.8s), which likely benefited from cached API responses, an already warmed Google Cloud Platform zone, or transient network fluctuations.

Table 6.2: Pipeline execution times across different attack scenarios

Test Scenario	**VMs**	**MAL Parse**	**Classification**	**Strategy**	**LLM Gen**	**Deploy**	**Total**
Credential Access (T1110.001)	1	0.002s	< 0.001s	0.065s	38.2s	27.0s	65.2s
Lateral Movement (T1021.001+)	5	0.003s	< 0.001s	0.066s	199.4s	27.2s	226.6s
Exfiltration (T1048.003)	1	0.003s	< 0.001s	0.063s	50.8s	27.4s	78.3s
Discovery (T1082+)	4	0.003s	< 0.001s	0.066s	309.0s	27.7s	336.8s
Persistence (T1547.001+)	5	0.004s	< 0.001s	0.067s	195.5s	16.8s	212.4s

Chapter 7

Conclusions

The choice of implementing an automated pipeline for the generation of IaC Honeypots comes from a preliminary literature review on the cyber defense field, where security teams compromise between honeypot realism and maintainability. Our proposed solution aims to introduce an innovative approach to honeypot deployment by leveraging the idea of dynamic cyber deception. We integrated the MITRE ATT&CK Enterprise Matrix for threat classification with a custom fine-tuned Llama-3.1-8b model to generate dynamic, detailed honeypots designed to prevent specific attacks. The pipeline validation demonstrates that the system efficiently creates targeted, syntactically accurate environments much faster than manual human efforts. More significantly, we observed a clear semantic alignment between the identified techniques and the produced infrastructure. Rather than deploying generic services, the system sets up context-aware decoys, such as a Samba server filled with authentic-looking credential files (e.g., *passwords.txt*) when an adversary attempts a Lateral Movement (T1021.002). This ability to dynamically adjust the deception strategy allows the defense to adapt to the attacker's specific approach. Regarding performance and resource consumption, our solution can run on machines equipped with consumer-grade hardware. This allows enterprise users to access a low-cost defense tool for their infrastructure, but also allows private users to have access to a powerful automatic honeypot deployment system that may be useful for home-labbing hobbies or self-hosting solutions. The main bottleneck of our application is the generation of the LLM output; depending on the complexity of the attack, the performance in terms of execution varies in a nonlinear manner, and using a more powerful GPU

may mitigate this problem. We must also note that using AI-Generated code in the cyber security field may raise multiple concerns regarding the intrinsic risks of Large Language Models: During our development and testing process we encountered numerous cases of hallucination or generation of structurally vulnerable configurations. We mitigate this issue by isolating the single honeypots using Virtual Private Clouds, eliminating any possible case of an attacker using the generated honeypot to enter the legitimate network. We also heavily use enforcement of the prompts the llm receives.

This work provides a foundation for possible future automatic honeypot deployments. While the implementation described in this thesis uses static MAL files to describe threats, a real-world implementation could integrate a process monitor with a detection agent to receive real-time telemetry. The pipeline would dynamically detect behavior that may constitute an alert (for example, abnormal application programming interface (API) calls associated with T1082 Discovery) and instruct the large language model (LLM) to generate honeypots before the adversary proceeds with its attack. We may also experiment with larger models to verify whether the hallucinations or misconfiguration errors decrease by expanding the reasoning capabilities of the system. Last, an expansion of the experimental dataset we propose, would benefit the generation capabilities of Llama-3.1-8b-Instruct, increasing the quantity of manually written samples and decreasing the synthetically augmented ones.

Bibliography

[1] Mohammed H. Almeshekah and Eugene H. Spafford. Cyber security deception. In Sushil Jajodia, V.S. Subrahmanian, Vipin Swarup, and Cliff Wang, editors, *Cyber Deception*, pages 23–50. Springer, Cham, 2016.

[2] David Lopes Antunes, Pavlos Cheimonidis, Eleftherios Batzolis, Kyriakos Ovaliadis, Salvador Llopis Sanchez, and Konstantinos Rantos. Spade: Enhancing adaptive cyber deception strategies with generative ai and structured prompt engineering. In *Proceedings of ARES Conference*, 2025. From Ref folder.

[3] Various Authors. Probing the depths: assessing the efficacy of the two-tier deception-driven security model. *Conference Paper*, 2024. From Ref folder.

[4] Various Authors. Resource-aware cyber deception for microservice-based applications. In *Conference Proceedings*, 2024. From Ref folder.

[5] Various Authors. A survey of using large language models for generating infrastructure as. *Survey Paper*, 2024. From Ref folder.

[6] Sean Barnum. Standardizing cyber threat intelligence information with the Structured Threat Information eXpression (STIX). Technical report, The MITRE Corporation, 2012.

[7] Pedro Beltrán López, Manuel Gil Pérez, and Pantaleone Nespoli. Cyber deception: State of the art, trends, and open challenges. *arXiv preprint arXiv:2409.07194*, 2024. arXiv:2409.07194 [cs.CR].

[8] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized language models. *Advances in Neural Information Processing Systems*, 36, 2023.

[9] Joar Ekelund. Security evaluation of damper system's communication and update process: Threat modeling using vehicleLang and securiCAD. Degree project in information and communication technology, KTH Royal Institute of Technology, Stockholm, Sweden, 2021.

[10] Deutsche Telekom Security GmbH. T-Pot: The all in one multi honeypot platform. https://github.com/telekom-security/tpotce, 2024. Accessed: 2025-10-01.

[11] Google. Google gemini: Ai models. https://deepmind.google/technologies/gemini/, 2024. Accessed: 2025.

[12] Aaron Grattafiori, Abhimanyu Dubey, et al. The llama 3 herd of models, 2024.

[13] Xiao Han, Nizar Kheir, and Davide Balzarotti. Deception techniques in computer security: A research perspective. *ACM Computing Surveys (CSUR)*, 51(4):1–36, 2018.

[14] Stanislava Ivanova and Naghmeh Moradpoor. Fake plc in the cloud, we thought the attackers believed that: How ics honeypot deception gets impacted by cloud deployments? In *2023 IEEE 19th International Conference on Factory Communication Systems (WFCS)*, pages 1–8. IEEE, 2023.

[15] Adam Jicha, Mark Patton, and Hsinchun Chen. Scada honeypots: An in-depth analysis of Conpot. In *Proceedings of the 2016 IEEE Conference on Intelligence and Security Informatics (ISI)*, pages 196–198. IEEE, 2016.

[16] Mario Kahlhofer and Stefan Rass. Application layer cyber deception without developer interaction. In *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2024.

[17] Junjie Li, Fazle Rabbi, Cheng Cheng, Aseem Sangalay, Yuan Tian, and Jinqiu Yang. An exploratory study on fine-tuning large language models for secure code generation. *arXiv preprint arXiv:2408.09078*, 2024.

[18] David Liebowitz, Surya Nepal, Kristen Moore, Cody J. Christopher, Salil S. Kanhere, David Nguyen, Roelien C. Timmer, Michael Longland, and Keerth Rathakumar. Deception for cyber defence: Challenges and

opportunities. In *2021 Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, pages 1–9. IEEE, 2021.

[19] Lin Long, Rui Wang, Ruixuan Xiao, Junbo Zhao, Xiao Ding, Gang Chen, and Haobo Wang. On llms-driven synthetic data generation, curation, and evaluation: A survey. *arXiv preprint arXiv:2406.15126*, 2024.

[20] MITRE Corporation. Mitre att&ck enterprise matrix. https://attack.mitre.org/matrices/enterprise/, 2024. Accessed: 2025.

[21] MITRE Corporation. Mitre att&ck framework. https://attack.mitre.org/, 2024. Accessed: 2025.

[22] Michel Oosterhof. Cowrie ssh/telnet honeypot. https://github.com/cowrie/cowrie, 2024. Accessed: 2025.

[23] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. Mulval: A logic-based network security analyzer. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, pages 8–8. USENIX Association, 2005.

[24] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

[25] Mayur Amarnath Palavalli and Mark Santolucito. Using a feedback loop for llm-based infrastructure as code generation. *arXiv preprint arXiv:2411.19043*, 2024. arXiv:2411.19043v1 [cs.SE].

[26] Paralax. Awesome honeypots. https://github.com/paralax/awesome-honeypots, 2024. Accessed: 2025.

[27] Jeffrey Pawlick, Edward Colbert, and Quanyan Zhu. A game-theoretic taxonomy and survey of defensive deception for cybersecurity and privacy. *ACM Computing Surveys (CSUR)*, 52(2):1–28, 2019.

[28] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2022.

[29] Niels Provos. A virtual honeypot framework. In *Proceedings of the 13th USENIX Security Symposium*, pages 1–14. USENIX Association, 2004.

[30] Saurabh Pujar, Luca Zheng, Luca Buratti, Christy Lewis, Tyler Kimball, Benjamin Tan, Limyee Yee, and Maxim Tabachnyk. Automated code generation for infrastructure as code using large language models. *arXiv preprint arXiv:2304.14188*, 2023. arXiv:2304.14188 [cs.SE].

[31] Lukas Rist. Glastopf – a dynamic, low-interaction web application honeypot. In *Proceedings of the 2010 FIRST Conference*, 2010. https://arxiv.org/abs/2409.07194 / https://www.first.org/conference/2010/.

[32] Baptiste Rozière et al. Code llama: Open foundation models for code. *arXiv preprint*, 2023. From Ref folder - codellama.pdf.

[33] Silvio Russo, Claudio Zanasi, and Michele Colajanni. Cyber defense through strategic dynamic deception. In *Proceedings*, 2024. Department of Computer Science and Engineering.

[34] Md Sajidul Islam Sajid, Jinpeng Wei, Ehab Al-Shaer, Qi Duan, Basel Abdeen, and Latifur Khan. symbsoda: Configurable and verifiable orchestration automation for active malware deception. *ACM Transactions on Privacy and Security*, 2024.

[35] Vasu Sethia and A. Jeyasekar. Malware capturing and analysis using Dionaea honeypot. In *2019 International Carnahan Conference on Security Technology (ICCST)*, pages 1–5. IEEE, 2019.

[36] Weizhou Shen et al. Exploring parameter-efficient fine-tuning techniques for code generation with large language models. *arXiv preprint arXiv:2308.10462*, 2024.

[37] Lance Spitzner. *Honeypots: tracking hackers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[38] The MITRE Corporation. About mitre. https://www.mitre.org/about/corporate-overview, 2024. Accessed: 2025.

[39] Cliff Wang and Zhuo Lu. Cyber deception: Overview and the road ahead. *IEEE Security & Privacy*, 16(2):80–84, 2018.

[40] Wojciech Wideł, Simon Hacks, Mathias Ekstedt, Pontus Johnson, and Robert Lagerström. The meta attack language - a formal description. *Computers & Security*, 130:103284, 2023.

[41] Zhiqiang Xu et al. A survey on lora of large language models. *arXiv preprint arXiv:2407.11046*, 2024.

[42] Marco Zambianco, Claudio Facchinetti, and Domenico Siracusa. A proactive decoy selection scheme for cyber deception using mitre att&ck. *Computers & Security*, 2024. arXiv:2404.12783v3 [cs.CR].

[43] Li Zhang and Vrizlynn L. L. Thing. Three decades of deception techniques in active cyber defense - retrospect and outlook. *Computers & Security*, 106:102288, 2021.

[44] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Ma, and Yongqiang Feng. LlamaFactory: Unified efficient fine-tuning of 100+ language models. *arXiv preprint arXiv:2403.13372*, 2024. arXiv:2403.13372 [cs.CL].

[45] Mu Zhu, Ahmed H. Anwar, Zelin Wan, Jin-Hee Cho, Charles A. Kamhoua, and Munindar P. Singh. A survey of defensive deception: Approaches using game theory and machine learning. *IEEE Communications Surveys & Tutorials*, 23(4):2460–2493, 2021.