TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Nikolaus Riedl 242934IVCM

AUTOMATED HONEYNET GENERATION FROM NATURAL LANGUAGE: AN LLM-BASED PIPELINE WITH MULTI-STAGE VALIDATION AND EMPIRICAL EVALUATION

Master's Thesis

Supervisor: Hayretdin Bahsi
Research Professor

Tallinn 2026

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Nikolaus Riedl 242934IVCM

AUTOMAATNE HONEYNET'I GENEREERIMINE LOOMULIKU KEELE PÕHJAL: LLM-PÕHINE KONVEIER MITMEETAPILISE VALIDEERIMISE JA EMPIIRILISE HINDAMISEGA

Magistritöö

Juhendaja: Hayretdin Bahsi
Research Professor

Tallinn 2026

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Nikolaus Riedl

27.05.2026

Abstract

Honeynets are vulnerable systems designed to attract and monitor attackers. Those forged networks are valuable tools for cybersecurity research and threat intelligence, but their manual construction is time-consuming and difficult to scale. This thesis presents the design, implementation, and evaluation of a framework that generates container-based honeynet environments from natural-language descriptions using large language models (LLMs).

The framework implements a multi-stage pipeline. First, an LLM extracts a structured intermediate representation (the WorldModel) from the user's natural-language prompt. The WorldModel is then validated for structural correctness, container images are verified against registries and corrected where necessary, and the result is compiled into a deterministic infrastructure-as-code artifact. This artifact is deployed as a set of Docker containers and verified through automated health checks. The framework uses a multi-agent extraction mode that splits generation into three specialized LLM agents (one for architecture, one for configuration, and one for enrichment) to address token budget limitations.

Following a Design Science Research methodology, the framework was evaluated against a benchmark corpus of 30 scenarios across three difficulty tiers, using operationalized metrics for deployability, scenario fit, repair effectiveness, and complexity sensitivity. The pipeline completed the full deployment path in 97% of scenarios. The container realization ratio was 91% for easy, 85% for medium, and 38% for hard scenarios. The overall scenario fit rate was 23% (7/30), with the medium tier (5/10) outperforming the easy tier (2/10). This unexpected result is explained by the LLM's stronger familiarity with well-known technology stacks (e.g., web servers, databases, message queues) compared to domain-specific infrastructure types (e.g., DNS sinkholes, VPN gateways), which affected the easy tier more than the medium tier.

The results demonstrate that LLM-based honeynet generation reliably produces deployable infrastructure, but scenario fidelity, specifically whether the deployed system matches the requested architecture, remains the primary bottleneck. The thesis contributes an end-to-end pipeline architecture, a reusable benchmark corpus with formal references, and empirical evidence on where LLM-based infrastructure generation succeeds and where it fails.

The thesis is written in English and is 121 pages long, including 9 chapters, 16 tables and 8 figures.

List of Abbreviations and Terms

API	Application Programming Interface
APT	Advanced Persistent Threat
BM25	Best Matching 25 (information retrieval ranking function)
CLI	Command-Line Interface
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DS	Design Science
DSRM	Design Science Research Methodology
FEDS	Framework for Evaluation in Design Science
HCL	HashiCorp Configuration Language
HTTP	Hypertext Transfer Protocol
IaC	Infrastructure as Code
IoT	Internet of Things
JSON	JavaScript Object Notation
LLM	Large Language Model
NL	Natural Language
OCI	Open Container Initiative
QA	Quality Assurance
RAG	Retrieval-Augmented Generation
RAM	Random Access Memory
RQ	Research Question
SCADA	Supervisory Control and Data Acquisition
SLR	Systematic Literature Review
TCP	Transmission Control Protocol
VM	Virtual Machine
VPN	Virtual Private Network
WSL	Windows Subsystem for Linux
YAML	YAML Ain't Markup Language

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Motivation and Problem Statement

Honeynets are networks of intentional vulnerabilities designed to capture, monitor and examine adversarial behaviour. They have become a well-established tool in cybersecurity research and active protection. Unlike purely passive defences, such as intrusion detection systems and firewalls, honeynets create controlled environments in which attackers can operate without exposing real infrastructure. They have been proven effective in gathering threat intelligence and understanding the tactics, techniques and procedures of advanced persistent threats.

Despite their value, honeynets are challenging to build and maintain. A realistic honeynet is only possible if certain services are selected with care, network segmentation is done in a way that seems possible, configuration objects such as credentials and environment variables are realistic, service dependencies are declared, and health monitoring is in place. All of these things are necessary to stop attackers from discovering the deception through inconsistencies.

In practice, creating even a moderately complex honeynet demands significant manual effort from an operator familiar with container orchestration, networking, and infrastructure-as-code tooling, in addition to the target environment being replicated.

To put concrete numbers on this manual effort, a default Cowrie SSH/Telnet honeypot can be stood up in roughly 30 minutes [1], but such an out-of-the-box deployment is routinely fingerprinted at Internet scale by automated scanners [2, 3]. Producing a scenario-realistic honeynet that evades these tools requires configuration across service banners, protocol stacks, filesystem contents, and credential planting, a manual burden that has motivated dedicated research frameworks for adaptive honeypot deployment [4] and Cowrie-specific deceptiveness tuning [5]. There is no published source that quantifies full multi-service honeynet construction in person-hours. The closest proxy is the cyber range literature, which characterises equivalent multi-service scenario construction as taking weeks to months of specialist effort [6, 7], in contrast to the roughly 30-minute single-honeypot reference point.

This manual approach is not scalable. Organisations that need to deploy honeynets across

multiple network segments, adapt them to evolving threat landscapes, or provision scenario-specific environments on demand cannot rely on ad-hoc, hand-crafted deployments. Several frameworks partially address this need. HoneyFactory [8] and HoneyChart [9] automate the deployment of container-based honeynets by scanning a protected network and mapping discovered services to predefined honeypot images through rule-based configuration. However, both systems are limited by a fixed catalogue of images and silently omit services for which no matching template exists.

Template-based systems like HoneyChart and HoneyFactory are reliable but they are limited: they can only deploy services for which a curated honeypot template already exists, and they require the operator to express requirements either by filling in structured configuration files or by feeding in the output of a scan of an existing network. Two consequences follow. First, any scenario that needs a service without a ready template cannot be deployed until someone writes a new template, which brings back the manual effort the automation was meant to remove. Second, the operator must already know which services to ask for. The tool cannot translate a high-level intent (such as "a small fintech back office with KYC and a trading API") into a concrete service list.

LLMs offer a different capability: open vocabulary translation from natural language. An LLM can map "a recursive DNS resolver that redirects suspicious domains" to specific images, environment variables, and zone assignments without a pre authored template, and it can generate the service list itself from a high level scenario description. This shifts the operator's effort from authoring infrastructure to specifying intent. The cost of this flexibility is reliability. Since LLM outputs are probability-based rather than definitive, they may contain made-up service names, invalid container images, incorrect port mappings, unrealistic network topologies or incorrect configuration syntax [10, 11]. An LLM-generated honeynet configuration is unlikely to be directly deployable without systematic validation, correction and verification, and even if deployment is successful, the configuration may not match the intended scenario or satisfy basic security requirements.

The central question this thesis addresses is therefore not merely whether an LLM can generate honeynet configurations. It also asks whether an LLM-based generation framework can produce results that are deployable. These results should be structurally correct with respect to network segmentation and dependency rules. They should also be semantically aligned with the input scenario. Finally, they should be robust in the sense that errors during generation or deployment can be automatically detected and repaired.

1.2 Objectives

This thesis presents a framework that transforms natural-language descriptions of target infrastructures into deployable, verified honeynet environments, and evaluates its design and implementation. The framework covers the entire process, from the text prompt to running containers. This includes structured world model extraction and validation, as well as deterministic compilation into infrastructure-as-code artefacts. It also covers deployment through OpenTofu and post-deployment verification.

The thesis aims to explicitly achieve the following objectives:

1. **Automated extraction:** The framework will extract a structured intermediate representation, the WorldModel, from a natural-language prompt. This will capture systems, network zones, dependencies, deployment configurations and deception metadata.
2. **Validation and repair:** The framework must validate the extracted WorldModel against structural, semantic and policy constraints and automatically repair common generation errors without human intervention.
3. **Deterministic deployment:** The framework compiles the validated WorldModel into a deployment projection, rendering it as an OpenTofu artefact that can be planned and applied automatically to produce a running container-based honeynet.
4. **Systematic evaluation:** The framework is evaluated against a structured benchmark corpus covering scenarios of different complexities. This uses metrics traceable to concrete pipeline stages and execution artefacts.

This thesis is restricted to Docker-based container deployments orchestrated through OpenTofu. Virtual machine provisioning, cloud-native infrastructure and runtime interaction simulation beyond static configuration are outside the scope of the framework. Deception quality is evaluated architecturally rather than through adversarial red-team testing.

1.3 Research Questions

Based on the objectives described above, the thesis addresses the following research questions:

RQ1 (Deployability): Can an LLM-based pipeline reliably generate deployable container-based honeynets from natural-language prompts?

RQ2 (Fidelity): How well do the generated honeynets match the structural intent of the input scenario?

RQ3 (Scalability): How does generation quality change as scenario complexity grows?

These questions are evaluated empirically using a benchmark corpus of 30 scenarios across three difficulty tiers. The evaluation metrics and their operationalisation are defined in Chapter 7.

1.4 Scientific Contribution and Novelty

The novelty of this work lies in combining natural-language understanding with multi-stage validation and deterministic infrastructure compilation into a single end-to-end system for honeynet generation. Prior frameworks either automate deployment from fixed catalogues (HoneyChart [9], which lets a defender pick honeypots from a web interface and produces ready-to-deploy configuration files for the Kubernetes container platform, and HoneyFactory [8], a five-module container-based architecture that builds a honeynet from a model of the protected business network and uses a statistical attacker-stage model to score how convincing the deception is) or use LLMs for infrastructure-as-code generation without domain-specific validation [12]. No existing system addresses the full path from unconstrained natural-language input to verified, running honeynet containers with automated error recovery. This combination of LLM-based extraction, OCI registry introspection, deterministic compilation, and multi-level repair into a single pipeline for honeynet generation is not present in the systems catalogued by recent surveys of LLM-powered honeypots [13] or LLM-based infrastructure-as-code generation [12].

This thesis makes the following contributions:

1. **An end-to-end framework for LLM-based honeynet generation.** The framework combines natural language processing with structured world model extraction, catalogue-guided image selection, contract-based validation, deterministic infrastructure-as-code compilation, and automated deployment. Unlike prior work that addresses individual aspects of honeynet automation, this system covers the entire path from prompt to running infrastructure.
2. **A multi-stage validation and repair architecture.** The framework includes structural validators, OCI image introspection, command policy enforcement, dependency inference, placement correction, and an iterative repair loop that can recover from common LLM generation errors. This addresses the fundamental unreliability of LLM outputs in safety-critical infrastructure contexts.

3. **A structured benchmark corpus with formal scenario references.** The thesis introduces a corpus of 30 benchmark scenarios, each paired with a machine-readable reference that specifies required services, zones, dependencies, and forbidden placements. This enables reproducible, automated evaluation of honeynet generation quality.
4. **An empirical evaluation across multiple quality dimensions.** The evaluation measures deployability, scenario fit, repair effectiveness, and complexity sensitivity using metrics that are each traceable to a specific pipeline stage or execution artefact. This provides a more complete assessment than prior work, which typically evaluates only whether a configuration is syntactically valid.

1.5 Structure of the Thesis

The remainder of this thesis is organised as follows:

Chapter 2 (Foundations) introduces the technical background required to understand the framework, including honeypot and honeynet concepts, large language models and structured generation, infrastructure-as-code with Docker and OpenTofu, validation and repair loops, and the dimensions along which AI-based infrastructure systems can be evaluated.

Chapter 3 (State of Research and Related Work) reviews existing automated honeynet and cyber range frameworks, research on LLM-based infrastructure-as-code generation, validation and repair of LLM-generated outputs, multi-agent and multi-phase LLM architectures, and honeypot detectability and deception quality. It then identifies the research gap that motivates this thesis.

Chapter 4 (Research Methodology and Evaluation Design) describes the research methodology, states the research objectives, defines the functional requirements, specifies the benchmark corpus and scenario references, and lays out the evaluation design.

Chapter 5 (Conceptual Design of the Honeynet Framework) presents the architecture of the framework, including the pipeline stages, core data models, world model extraction, validation and repair, deterministic compilation, deployment and runtime verification, quality assurance, scenario-fit evaluation, and an ablation analysis of the design.

Chapter 6 (Implementation) describes the technical realisation of the framework, covering the project structure, CLI and orchestration layer, LLM provider abstraction, world model extraction, validation and repair, compilation and deployment, and the testing and

QA infrastructure.

Chapter 7 (Evaluation Framework) defines the evaluation metrics in detail, describes the experimental setup, and reports the results across deployability, scenario fit, repair and robustness, complexity sensitivity, and timing.

Chapter 8 (Discussion) interprets the results by answering each research question, positions the findings with respect to prior work, discusses the limitations of the framework, and derives implications for research and practice.

Chapter 9 (Conclusion and Future Work) summarises the thesis, highlights the key findings and contributions, and outlines directions for future work.

2. Foundations

The technical ideas needed to fully understand the framework presented in this thesis are discussed in this chapter. Honeypots, honeynets, and honeytokens are defined in Section 2.1. Large language models and the issue of hallucination in structured output production are introduced in Section 2.2. Infrastructure as code with Docker and OpenTofu is covered in Section 2.3. Loops used for validation and repair are described in Section 2.4. The evaluation dimensions used in this thesis are defined in Section 2.5.

2.1 Honeypots, Honeynets, and Cyber Deception

2.1.1 Definitions

A *honeypot*, in Spitzner's classical definition [14], is "a security resource whose value lies in being probed, attacked, or compromised." Unlike production systems, which serve legitimate users, honeypots exist solely to attract and observe adversary behaviour. Any interaction with a honeypot is by definition unauthorised, which simplifies detection: all traffic directed at a honeypot is suspicious.

A *honeynet* is a network of honeypots designed to present a realistic multi-service environment to an attacker [15]. Where a standalone honeypot emulates a single service (e.g., an SSH server), a honeynet emulates an interconnected infrastructure with multiple services, network segments, and inter-service dependencies, that is, explicit declarations of which services require other services to be running before they can start (e.g., a web application depends on its database, which in turn depends on an authentication service). The additional complexity provides richer intelligence about attacker behaviour, including lateral movement patterns and multi-stage attack sequences.

A *honeytoken* is a data-level deception artefact: a fake credential, a bogus database entry, or a misleading API key planted within a system to detect unauthorised access [14]. Unlike honeypots, which are systems, honeytokens are individual data items. Their use or exfiltration signals a compromise.

2.1.2 Interaction Levels

Honeypots are classified by the depth of interaction they permit with attackers [14, 15]:

Low-interaction honeypots don't have a working backend; instead, they just mimic a service's network-facing surface (e.g., replying to connection attempts with a protocol banner). Although they are low-risk and lightweight, they only gather a little amount of intelligence.

Without hosting an entire operating system or program, medium-interaction honeypots mimic certain aspects of the service protocol, such as logging email contents and receiving SMTP commands.

High-interaction honeypots allow full attacker involvement, including file uploads, privilege escalation, and lateral movement, by running actual services on real operating systems. Although they require rigorous isolation and pose a higher operational risk, they capture the richest data.

The methodology described in this thesis creates honeynets based on Docker containers. The output falls into the high-interaction category in terms of service fidelity because the deployed containers execute actual service images (such as PostgreSQL, Nginx, and Redis). Nevertheless, runtime interaction simulation (e.g., producing realistic shell answers) is not provided by the framework; this complementary layer is covered by independent research reviewed in Chapter 3.

2.1.3 Network Isolation

Network isolation is necessary for effective honeynet deployment in order to stop compromised honeypots from being utilised as attack platforms against production systems. Honeypots are typically placed in dedicated network segments that are shielded from production traffic by firewalls or virtual network boundaries [15].

In this thesis, Docker networks with the `internal` option are used to establish network isolation; containers on an internal Docker network are able to communicate with one another but are unable to access the host network or the internet (Section 2.3.3).

2.2 Large Language Models and Structured Generation

2.2.1 Transformer Architecture

The large language models (LLMs) used in this thesis are based on the *transformer* architecture introduced by Vaswani et al. [16]. A transformer processes input tokens through layers of self-attention, which lets each token look at every other token in the

input directly rather than having to pass information through intermediate tokens as in older recurrent architectures like RNNs and LSTMs. For example, when the LLM in this thesis generates a web container that should depend on a database, self-attention lets the model look directly back at the database system it declared earlier in the same response (so it can copy the database name into the web container's `depends_on` list and into environment variables such as `DB_HOST`), even if many other systems and configuration lines were generated between the two declarations. This makes transformers better at capturing long-range relationships and easier to train in parallel. This thesis does not train or modify a transformer; it consumes existing transformer-based LLMs through their provider APIs as a black-box service.

Contemporary LLMs, like GPT-4 [17], train on massive text and code corpora and stretch the transformer architecture to hundreds of billions of parameters. This thesis takes advantage of these models' ability to provide structured outputs (JSON, YAML, and HCL) in response to natural-language prompts for infrastructure configuration creation.

2.2.2 Hallucination

A central limitation of LLMs is *hallucination*: the generation of output that appears fluent and coherent but is factually incorrect, logically inconsistent, or fabricated [18]. Ji et al. distinguish two forms:

Factuality hallucination: The model generates claims that contradict established facts (e.g., referencing a Docker image that does not exist in any registry).

Faithfulness hallucination: The model generates output that deviates from the provided context or instructions (e.g., placing a database in a public zone when the prompt specified internal placement).

When it comes to the creation of infrastructure, hallucinations may appear as invalid port mappings, fake service names, inaccurate image references, nonexistent environment variables, and implausible dependence structures. Liu et al. [10] classify these as "resource hallucinations" in their taxonomy of LLM-generated code errors. The multi-stage validation and repair architecture described in Chapter 5 is meant to identify and address these flaws.

2.2.3 Structured Output Generation

When an LLM is instructed to produce structured output (e.g., a YAML document conforming to a specific schema), two failure modes arise beyond factual hallucination:

1. **Schema violations:** The output does not conform to the expected structure (e.g., missing required fields, wrong data types, malformed syntax).
2. **Truncation:** For large outputs, the model reaches its token budget and produces incomplete documents.

To prevent truncation, this thesis uses a multi-phase extraction approach (Chapter 5, Section 5.3.2): generation is split into several smaller, focused calls, each producing an output of manageable size. Schema violations are handled separately by a structural validation step (Section 5.4.1).

An alternative way to handle schema violations is grammar-constrained decoding, as implemented in frameworks such as Outlines [19]. These frameworks enforce validity directly during generation: token sampling is modelled as transitions in a finite-state machine, so every generated token is guaranteed to keep the output structurally valid. Geng et al. [20] evaluate six such frameworks on 10,000 real-world JSON schemas and show that, without constrained decoding, models often violate complex schema constraints. This finding motivates the explicit schema-validation layer used in this thesis.

The framework presented here takes a different approach from grammar-constrained decoding. Rather than preventing invalid tokens during generation, it performs *post-hoc validation*: the LLM is allowed to generate freely, the resulting output is parsed and checked against a structural validator, and detected violations trigger deterministic repair or a further LLM call. This is conceptually closer to the *Self-Refine* pattern described in Section 2.4 than to constrained decoding. The choice reflects two considerations: constrained decoding requires provider-specific integration that is not uniformly available across LLM backends (Ollama, OpenAI, Anthropic), and post-hoc validation can enforce domain-specific constraints (e.g., dependency graph acyclicity, zone placement rules) that are not easily expressible as context-free grammars.

2.3 Infrastructure as Code with Docker and OpenTofu

2.3.1 Infrastructure as Code

Infrastructure as Code (IaC) is the practice of managing and provisioning computing infrastructure through machine-readable definition files rather than interactive configuration tools [21]. The core principles of IaC are:

Declarative specification: The desired end state is described rather than the sequence of steps to reach it.

Reproducibility: The same definition applied to the same environment always produces the same result.

Version control: Infrastructure definitions are stored in version control alongside application code, enabling audit trails and rollback.

In this thesis, the framework generates IaC artefacts in HashiCorp Configuration Language (HCL), which are then executed by OpenTofu.

2.3.2 OpenTofu and the Deployment Lifecycle

OpenTofu [22] is an open-source fork of Terraform [23], created in 2023 under the Linux Foundation after HashiCorp relicensed Terraform to the Business Source License. OpenTofu maintains full compatibility with Terraform providers and HCL syntax.

The OpenTofu deployment lifecycle consists of four stages:

1. `init`: Initializes the working directory and downloads required provider plugins.
2. `validate`: Checks the HCL configuration for syntax errors, undefined references, and type mismatches. This is a static analysis step that does not contact any infrastructure providers.
3. `plan`: Computes the difference between the current state and the desired state defined in the configuration. The plan shows which resources will be created, modified, or destroyed.
4. `apply`: Executes the plan, creating or modifying resources to match the desired state. In this thesis, the apply step creates Docker networks and containers.

Each stage produces a distinct artefact that the framework's validation pipeline inspects: the validate stage produces a pass/fail result, the plan stage produces an execution plan, and the apply stage produces state files recording the created resources.

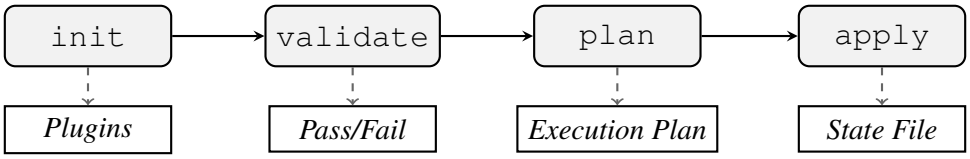

Figure 1. The OpenTofu deployment lifecycle. Each stage produces a distinct artefact that the validation pipeline can inspect to ensure correct infrastructure generation.

2.3.3 Docker Containers and Networks

Docker containers are lightweight, isolated process environments built from container images. A container image is a read-only template that includes the application, its runtime dependencies, and metadata (entrypoint, default command, exposed ports, environment variables) [24].

Docker networks provide communication channels between containers. The two network modes relevant to this thesis are:

Bridge networks: The default network driver. Containers on the same bridge network can communicate via their container names. Bridge networks are reachable from the host.

Internal networks: A bridge network with the `internal` flag set. Containers can communicate with each other but have no connectivity to the host or external networks. This mode is used for honeynet backend services that should not be directly accessible.

The framework maps each honeynet zone to a Docker network: public zones use standard bridge networks, internal zones use internal bridge networks.

2.3.4 OCI Image Distribution

Container images are distributed through registries conforming to the OCI Distribution Specification [25]. The specification defines a REST API for pushing and pulling image manifests and layers. A key concept is the *digest*: a content-addressable SHA-256 hash that uniquely identifies an image manifest. Pinning a deployment to a digest (e.g.,

`nginx@sha256:abc...`) ensures that the exact same image is used across deployments, regardless of whether the tag (e.g., `nginx:1.25`) has been updated to point to a newer build.

The framework uses OCI HEAD requests to validate image existence and retrieve digests without pulling the full image. This approach does not require a running Docker daemon and does not count towards registry pull rate limits.

2.4 Validation and Repair Loops

LLM-generated outputs are probabilistic and may contain errors at multiple levels: syntactic (malformed structure), semantic (incorrect relationships), and operational (configurations that fail at deployment). Correcting these errors requires mechanisms that detect failures and feed structured error information back to the generator.

2.4.1 The Generate-Validate-Repair Pattern

By treating flawed code as input and corrected code as output, Xia et al. [26] show how LLMs can be used for automated program repair. This is generalised into the *Self-Refine* pattern by Madaan et al. [27]: an LLM produces output, gets feedback (from itself or an external tool), and then produces a revised version.

This pattern can be applied with two types of feedback:

Internal feedback: The LLM critiques its own output without external signals. This is effective for surface-level errors but limited by the model's inability to detect errors it does not know it has made.

External feedback: A validator, compiler, or runtime environment produces structured error messages that are fed back to the LLM. This provides ground-truth signals that the model cannot obtain from its own weights.

The framework in this thesis uses external feedback at four levels: structural validation errors (e.g., "system X references undefined zone Y"), OCI registry responses (e.g., "image not found"), pre-deploy container probe output (Docker logs from brief startup tests), and OpenTofu error output (e.g., "container exited immediately"). Detailed repair strategies are described in Chapter 5, Section 5.4.5.

2.4.2 Diminishing Returns in Iterative Repair

Iterative repair becomes less successful with each iteration, according to research on feedback loops for IaC generation [28]. Most potential improvements are captured in the initial repair pass; subsequent passes produce progressively smaller benefits and may introduce new problems. This result guides the framework's design choice to invest in front-loading quality through multi-phase extraction instead of depending just on post-hoc repair, and to restrict repair attempts to a configurable maximum.

2.5 Evaluation Dimensions

Evaluating the quality of LLM-generated infrastructure requires metrics that go beyond syntactic validity. The literature on code generation evaluation [29] and IaC evaluation [11] establishes a hierarchy of quality dimensions:

Syntactic validity: The generated output conforms to the grammar of the target language (e.g., valid YAML, valid HCL). This is a necessary but insufficient condition.

Semantic correctness: The generated output expresses the intended meaning (e.g., the correct services are present, the correct dependencies are declared). Kon et al. [11] demonstrate that LLMs can achieve high syntactic validity while failing semantic correctness: GPT-4 achieves only 19.36% pass@1 on IaC-Eval despite generating syntactically valid Terraform in most cases.

Deployability: The generated output successfully deploys (e.g., `tofu plan` and `tofu apply` succeed, and the resulting containers start and remain healthy, meaning that Docker reports them as running and any declared healthcheck probes continue to pass). Zhang et al. [30] show that 42.7% of syntactically correct IaC templates fail during deployment, making deployability a distinct evaluation dimension.

The multi-dimensional nature of IaC quality is independently confirmed by TerraMetrics [31], which defines 40 structural quality metrics extracted from the Terraform AST, demonstrating that syntactic structure and configuration quality are independently measurable and frequently uncorrelated dimensions of an IaC artefact's correctness.

Two more dimensions are added to this structure by the evaluation framework used in this thesis (Chapter 7): *scenario fidelity* (i.e., whether the deployment satisfies the structural requirements of the input scenario) and *repair effectiveness* (i.e., whether the framework

can recover from failures). In essence, the standard *pass@k* metric from code generation evaluation [29], which is the likelihood that at least one of k generated samples passes all tests, is modified: every scenario is assessed in relation to a formal reference that outlines necessary services, zones, dependencies, and prohibited placements.

The concepts introduced in this chapter (honeypot classification, LLM hallucination, IaC lifecycle, validation-repair loops, and evaluation dimensions) form the technical foundation on which the framework's conceptual design (Chapter 5) and evaluation methodology (Chapter 7) are built.

3. State of Research and Related Work

This chapter reviews the state of research across five areas that intersect with this thesis: automated honeynet and cyber range frameworks (Section 3.1), LLM-based infrastructure-as-code generation (Section 3.2), validation and repair of LLM-generated outputs (Section 3.3), multi-agent and multi-phase LLM architectures (Section 3.4), and honeypot detectability and deception quality assessment (Section 3.5). Section 3.6 synthesizes the findings into a research gap statement.

3.1 Automated Honeynet and Cyber Range Frameworks

Automated deployment of honeypots and cyber ranges has been pursued along three lines: template-based orchestration, event-driven adaptation, and specification-driven generation.

Template-based orchestration. HoneyChart [9] automates honeypot deployment on Kubernetes by generating Helm charts from user-selected service types. Defenders choose the desired honeypots from a web interface; the system produces deployment-ready charts. Chouliaras et al. [32] describe a Docker-based cyber range platform on OpenStack that uses Ansible and Heat templates for orchestration. HoneyKube [33] is a honeypot designed to mimic the microservices style of modern web applications, where one application is split into many small interacting services rather than running as a single monolithic program. It is deployed on Google's managed Kubernetes service and attaches a small traffic-capture container alongside each service to record attacker activity. The deployment collected 850 GB of network and system data over its measurement period, illustrating the data yield possible from a realistic multi-service honeynet but also the manual scenario-authoring effort that this thesis seeks to automate. These systems demonstrate the maturity of container-based deployment for honeypots but rely on predefined configurations that must be manually authored for each new scenario.

Event-driven adaptation. Bartwal et al. [34] describe a SOAR engine that dynamically deploys honeypots in response to detected attacks. In a four-day live deployment, the engine orchestrated honeypots 40 times and detected 7823 attacks; dynamic honeypots engaged attackers for an average of 3148 seconds compared to 102 seconds for static honeypots. HoneyFactory [8] generates container-based honeynets reflecting a protected business network and uses a Hidden Markov Model, a statistical model that tracks how an attacker progresses through a sequence of hidden internal states based on the externally

observable actions, to score how convincing the deception is at each stage of the attack. This thesis takes a different evaluation route: instead of statistically modelling attacker engagement, it evaluates the deployment structurally against a machine-readable scenario reference. Moeller [35] proposes a three-layer architecture (sensor, decision, honeypot) where a reinforcement-learning agent, that is, a software agent that learns through trial and error by receiving rewards for actions that lead to a desired outcome, determines when to escalate sessions from low-interaction sensors to high-interaction honeypots. This is a runtime adaptation mechanism, complementary to the static infrastructure generation this thesis addresses. These systems adapt at runtime to observed attacker behaviour but do not accept natural-language specifications and do not use LLMs for configuration generation.

Specification-driven generation. Costa et al. [36] propose VSDL, a domain-specific language for cyber range scenario generation that compiles to deployment scripts via a Satisfiability Modulo Theories (SMT) solver, that is, a logic engine that checks whether a set of constraints can simultaneously be satisfied and produces a concrete configuration if so. The solver guarantees that the specification is satisfiable before deployment scripts are generated, which is possible because VSDL is a formal language with precise semantics. This thesis cannot rely on such guarantees because its input is unstructured natural language, so it uses structural validation and OCI-image introspection to catch errors after generation rather than before. Caturano et al. [37] extract information from public exploit databases to automatically create vulnerable WordPress Docker environments, generating 484 scenarios (39% of ExploitDB's WordPress entries). These systems show that scenario generation can be automated, but they require either a formal domain-specific language (VSDL) or a structured input source (exploit database entries). Neither accepts an unconstrained natural-language description of the target infrastructure, which is the input mode this thesis aims to support.

The closest precedent to this thesis is the work of Ahmed et al. [38], who present a multi-agent system that transforms natural-language prompts into Docker-based cybersecurity training environments. Their system uses three specialised agents (master agent, machine worker agents, QA agent) and was evaluated across 14 attack scenarios. Four key differences distinguish this thesis: (1) Ahmed et al. target cyber ranges for training rather than honeynets for deception; (2) their system lacks a structured intermediate representation comparable to the WorldModel; (3) it does not perform OCI-level image validation or deterministic compilation; and (4) it does not include a formal benchmark evaluation framework with machine-readable scenario references.

None of the reviewed honeynet or cyber range frameworks combine natural-language input with LLM-based generation and formal benchmark-driven evaluation.

While template-based and event-driven approaches address honeynet deployment automation, and specification-driven systems formalize the generation process, the broader field of LLM-based infrastructure generation offers complementary techniques for bridging natural language and deployable configurations.

3.2 LLM-Based Infrastructure-as-Code Generation

The use of LLMs to generate infrastructure-as-code configurations is an active research area. Srivatsa et al. [12] provide the first survey dedicated to LLM-based IaC generation, finding that GPT-3.5-Turbo achieves 50–60% success rates on Terraform generation tasks, while smaller models such as CodeParrot do not exceed 10%.

Subsequent benchmark work has quantified this gap further. Kon et al. [11] introduce IaC-Eval, a benchmark of 458 human-curated AWS Terraform scenarios with automated validation through `terraform plan/apply`. GPT-4 achieves only 19.36% pass@1 on IaC-Eval, compared to 86.6% on the Python benchmark HumanEval [39], a more than four-fold gap that confirms IaC generation is substantially harder than general code generation. Abhyankar et al. [40] extend this to multiple IaC formats (CloudFormation, Terraform, CDK), finding that LLMs achieve over 95% syntactic validity but struggle with semantic alignment: generated configurations may parse correctly but fail to deploy or fail to match the requested architecture.

Two recent systems address these quality gaps through iterative refinement. Zhang et al. [30] propose IaCGen, which uses format verification, syntax checking, and live deployment feedback in an iterative loop. With 15 iterations, models achieve an average 75.2% pass rate, representing approximately 200% improvement over first attempts. Jana et al. [41] take a different approach with TerraFormer, combining supervised fine-tuning with reinforcement learning guided by formal verifiers for syntax, deployability, and policy compliance. TerraFormer improves correctness by 15.94% on IaC-Eval over its base LLM and outperforms models 50x larger.

Beuter et al. [42] evaluate LLMs for Kubernetes manifest generation using Zero-Shot, Few-Shot, Prompt-Chaining, and Self-Refine strategies. A notable finding is that sequential prompt chaining often degraded output quality rather than improving it, motivating tool-supported feedback approaches over pure prompt engineering. Palavalli and Santolucito [28] study feedback loops for CloudFormation generation and find that effectiveness decreases with diminishing returns per iteration, suggesting that front-loading generation quality through structured prompting may be preferable to relying solely on post-hoc repair.

Mehta et al. [43] evaluate GPT-3.5 and GPT-4 for generating GitHub Actions YAML workflows, finding that GPT-4 achieves substantially higher syntax correctness than GPT-3.5, a pattern consistent with Abhyankar et al.'s [40] finding of over 95% syntactic validity for state-of-the-art LLMs on IaC tasks, confirming that LLMs can produce syntactically valid structured configuration, while semantic correctness remains the harder problem.

The adjacent area of network configuration generation shows similar patterns: Wang et al. [44] benchmark LLMs on four network configuration task types (translating requirements to formal specifications, generating API calls, developing routing algorithms, and generating low-level protocol configurations) and find that while GPT-4-Turbo achieves the best overall results, performance drops substantially on tasks requiring integration of multiple constraints, motivating tool-supported feedback over pure prompt engineering.

The thesis presented here differs from this body of work in three respects: it targets honeynet-specific infrastructure rather than general cloud services; it uses a structured intermediate representation (WorldModel) between natural language and IaC rather than generating IaC directly; and it evaluates against a scenario-fit benchmark with formal references rather than only syntactic or deployment-success criteria.

The quality gaps identified in LLM-based IaC generation motivate the need for post-generation validation and repair, an area with a substantial and growing body of research.

3.3 Validation and Repair of LLM-Generated Outputs

The unreliability of LLM-generated code and configurations has motivated substantial research on validation and repair mechanisms.

Self-refinement without external tools. Madaan et al. [27] propose Self-Refine, where the same LLM serves as generator, critic, and refiner in an iterative loop without additional training, achieving up to 13% absolute improvement on code generation tasks. Shinn et al. [45] introduce Reflexion, where agents maintain verbal self-reflections in episodic memory, achieving 91% pass@1 on HumanEval. These approaches demonstrate that LLMs can improve their own outputs through iteration, but both rely on the LLM's internal assessment rather than external validation signals.

Tool-augmented repair. Chen et al. [46] teach LLMs to self-debug using code execution feedback, achieving state-of-the-art results on text-to-SQL and code translation benchmarks. Bi et al. [47] demonstrate that compiler feedback used to iteratively refine LLM-generated code improves repository-level generation by over 80%. Mastropaolo

et al. [48] evaluate LLMs for automatically patching IaC projects, confirming that IaC errors combine complexities from multiple infrastructure layers requiring domain-specific knowledge.

Repair as search. Tang et al. [49] formalize iterative LLM repair as an exploration-exploitation tradeoff and propose REx, which navigates a tree of refinements using Thompson Sampling. REx requires approximately $2\times$ fewer LLM calls on competition programming tasks and up to $5\times$ fewer on loop invariant synthesis compared to greedy and breadth-first baselines. Eshghie [50] identifies a related problem: self-refinement pipelines with competing constraints can enter oscillatory failures, where satisfying one constraint violates another.

Code hallucination taxonomies. Liu et al. [10] establish a taxonomy of code hallucinations with three primary categories and twelve specific types, distinguishing knowledge hallucinations (fabricating APIs or parameters that do not exist) from faithfulness hallucinations (generating code that contradicts the specification). Their category of "resource hallucinations," which refers to referencing nonexistent libraries, APIs, or configuration parameters, is directly applicable to IaC generation, where the LLM may reference Docker images, environment variables, or service configurations that do not exist.

Implications for this thesis. The framework presented in this thesis uses tool-augmented repair (OCI validation, `tofu plan`, runtime health checks) rather than pure self-refinement. The decision to limit repair attempts and to invest in front-loading quality through multi-phase extraction was informed by Palavalli and Santolucito's finding [28] that feedback loop effectiveness decreases per iteration. The framework does not implement tree-structured repair [49] or meta-repair [50]; these represent potential extensions.

Beyond repair strategies applied to a single generation pass, an orthogonal approach to improving output quality is to decompose the generation task itself into multiple specialised phases or agents.

3.4 Multi-Agent and Multi-Phase LLM Architectures

Decomposing complex generation tasks into multiple specialised LLM invocations has been studied extensively in recent work on code generation.

Hong et al. [51] encode Standardized Operating Procedures into prompt sequences for multi-agent collaboration, assigning roles such as architect, engineer, and tester to separate

agents. Structured document exchange between agents reduces cascading hallucination errors compared to unstructured dialogue-based systems. Qian et al. [52] introduce ChatDev, where agents communicate through a "chat chain" mechanism with "communicative dehallucination," that is, cross-validation of outputs in structured dialogue to reduce error propagation across phases.

Huang et al. [53] decompose code generation into programmer, test designer, and test executor agents, achieving 77.4% pass@1 on HumanEval-ET with GPT-3.5. Islam et al. [54] use four agents emulating the human programming cycle (recall, plan, code, debug), achieving 93.9% pass@1 on HumanEval. He et al. [55] survey multi-agent systems for software engineering, identifying role specialisation and structured feedback loops as the architectural properties most consistently correlated with quality improvements. This is empirically confirmed in the IaC domain by Hussain Khan et al. [56], whose MACOG framework raises GPT-5 from 54.90 to 74.02 and Gemini-2.5 Pro from 43.56 to 60.13 on IaC-Eval (relative gains of approximately 35% and 38% respectively over a RAG baseline) by decomposing generation into specialised agents (Architect, Engineer, Reviewer, Security Prover) combined with deployment-feedback loops.

Wu et al. [57] provide AutoGen, a general-purpose framework for multi-agent conversations where customizable agents can combine LLM, human, and tool backends. Jin et al. [58] propose a six-layer vision framework for LLM-based code generation with distinct Input, Orchestration, Development, and Validation phases, noting that prompt sensitivity makes single-prompt generation unpredictable for complex tasks.

The multi-phase extraction pipeline in this thesis follows this body of work by decomposing the generation task into focused sub-tasks (architecture, configuration, enrichment). The key difference is that the phases operate on a shared structured intermediate representation (WorldModel) rather than communicating through dialogue or document exchange. This design is informed by MetaGPT's finding that structured intermediaries reduce cascading errors, and by the observation from the code generation literature that decomposition consistently improves quality for complex tasks.

The preceding sections addressed how honeynets are generated, validated, and refined. A complementary question is how the quality of the resulting deployment is assessed, specifically whether it can withstand scrutiny by an attacker.

3.5 Honeypot Detectability and Deception Quality

The quality of a honeypot deployment is ultimately determined by whether it can deceive attackers. This dimension is not evaluated in this thesis, but the related literature is reviewed to position the thesis's structural evaluation within the broader landscape.

Srinivasa et al. [3] present a multistage framework for honeypot fingerprinting, identifying detection vectors across protocol responses, timing inconsistencies, and behavioural patterns. Aradi et al. [59] develop a quantitative evaluation framework using a test suite grounded in the MITRE ATT&CK framework, testing 15 honeypots across five service types and deriving realism scores from response-time deviations and behavioural fidelity. Javadpour et al. [60] survey deception techniques for honeypots, providing a mathematical model for comparing honeynet approaches and identifying deception fidelity as a key factor in effectiveness. Spahn et al. [61] deployed high-interaction container honeypots emulating Docker and Kubernetes services, collecting attack data over 94 days and demonstrating that attackers engaged within minutes of launch.

A distinct line of research uses LLMs not for infrastructure generation but for runtime honeypot interaction. Sladic et al. [62] describe shelLM, a Linux shell honeypot that uses LLMs to generate realistic command-line responses, achieving a True Negative Rate of 0.90 in human evaluation. Bridges et al. [13] provide the first systematization of knowledge on LLM-powered honeypots, covering the evolution from log analysis to automated intelligence generation. Ahmed et al. [63] propose SPADE, which uses structured prompt engineering to automate generation of adaptive cyber deception strategies.

This thesis uses LLMs at the *infrastructure generation* layer (designing and deploying the honeynet topology) rather than at the *runtime interaction* layer (generating responses to attacker commands). These are complementary concerns: the infrastructure generation approach produces the deployment that a runtime interaction system would operate within. The evaluation in this thesis is limited to structural correctness and does not assess runtime deception fidelity. The fingerprinting and deception quality literature reviewed above defines the evaluation dimensions that would be required for a full deception quality assessment, which remains future work.

3.6 Research Gap and Positioning

Table 1 summarises the reviewed approaches along six dimensions relevant to this thesis.

System	NL input	LLM	Intermediate rep.	Validation	Deployment	Benchmark
HoneyChart	✗	✗	—	—	Kubernetes	✗
HoneyFactory	✗	✗	—	HMM scoring	Docker	✗
SOAR Engine	✗	✗	—	—	Docker	✗
VSDL	✗ (DSL)	✗	SMT model	SMT solver	Terraform	✗
ExploitWP2Docker	✗ (exploit DB)	✗	—	—	Docker	✗
Ahmed et al.	✓	✓	—	QA agent	Docker	✗
IaC-Eval	✓	✓	—	tofu validate	Terraform	✓
TerraFormer	✓	✓	—	RL + verifier	Terraform	✓
IaCGen	✓	✓	—	tofu + iteration	Terraform	✓
shelLM	✗	✓	—	—	runtime layer	✗
This thesis	✓	✓	**WorldModel**	**OCI + structural + tofu**	**OpenTofu**	✓

Table 1. Comparison of related systems. ✓ indicates the dimension is addressed; — indicates it is not. SMT = Satisfiability Modulo Theories. HMM = Hidden Markov Model. RL = reinforcement learning. DSL = domain-specific language. OCI = Open Container Initiative image registry (see Section 2.3.4). "tofu" refers to the OpenTofu validate and plan steps (see Section 2.3.2). shelLM operates at the runtime interaction layer rather than the infrastructure layer.

The review reveals that the intersection of three capabilities, namely (1) natural-language-driven generation, (2) honeynet-specific deployment, and (3) formal benchmark evaluation, is not addressed by any existing system.

Existing honeynet frameworks (Section 3.1) automate deployment through templates or event-driven orchestration but do not accept natural-language specifications and do not use LLMs for configuration generation. Existing LLM-based IaC generation systems (Section 3.2) target general cloud infrastructure and evaluate primarily syntactic validity and deployment success, not scenario-specific structural fidelity. Multi-agent architectures (Section 3.4) demonstrate that task decomposition improves generation quality but have not been applied to honeynet-specific infrastructure generation. Validation and repair mechanisms (Section 3.3) are well-studied for general code but have not been combined with OCI-level image introspection and runtime container health verification in a single pipeline.

There is limited relevant literature on honeynet production using LLM. Rather than a lack of activity in the constituent fields—honeynet automation, LLM-based IaC generation, and multi-agent code creation are all active areas with significant recent papers. The lack of researchshows the novelty of the overlap. In particular, the difference is in how they are combined.

This thesis addresses this gap by developing a pipeline that combines natural language driven LLM extraction with a structured intermediate representation (WorldModel), multi-stage validation (structural, OCI, deployment, runtime), deterministic IaC compilation, and formal benchmark evaluation against machine-readable scenario references. The contribution is not the individual components (LLM extraction, IaC compilation, and benchmark evaluation each have precedent in the reviewed literature) but their integration into an end-to-end pipeline for honeynet infrastructure generation, and the design knowledge that emerges from building and evaluating this integration.

4. Research Methodology and Evaluation Design

This chapter presents the research methodology (Section 4.1), derives the research objectives and requirements (Sections 4.2–4.3), describes the benchmark corpus construction (Section 4.4), and defines the evaluation design (Section 4.5).

4.1 Research Methodology

4.1.1 Design Science as Research Framework

This thesis follows the Design Science Research Methodology (DSRM) as formulated by Peffers et al. [64]. Design science research produces *prescriptive knowledge*, that is, knowledge about how artefacts can and should be designed to achieve a desired set of goals [65]. Unlike explanatory research, which seeks cause-effect relationships, design science investigates means-ends relationships: given a problem, what design choices lead to effective solutions, and under what conditions do they succeed or fail?

The research contribution of a design science thesis is not the artefact itself, but the *design knowledge* that emerges from its construction and evaluation. Hevner et al. [65] define this as follows: "knowledge and understanding of a problem domain and its solution are achieved in the building and application of the designed artefact." The artefact serves as a vehicle through which the researcher generates insights about the problem domain, in this case insights about the capabilities and limitations of LLM-based infrastructure generation for honeynet systems.

This framing is analogous to experimental science in other domains: a developed artefact (such as a prototype vaccine) is intended to be useful, but the scientific contribution lies in the development and validation process: what is learned about the problem, what design decisions prove effective, and what boundaries are discovered. The framework developed in this thesis is a research artefact, not a production system. Its value for the field lies in the design knowledge it produces about how to architect LLM-based generation pipelines, what quality dimensions are measurable, and where current LLM capabilities fall short.

4.1.2 DSRM Process Model

Peffers et al. [64] define six process steps for design science research. Table 2 maps these steps to the activities performed in this thesis.

DSRM Step	Thesis Activity
1. Problem identification	Manual honeynet construction is time-consuming, error-prone, and does not scale. LLM-based generation introduces unreliability (hallucinations, schema violations, invalid configurations) that must be systematically addressed. (Chapter 1)
2. Objectives of a solution	Three research questions operationalise the objectives: deployability (RQ1), fidelity (RQ2), and scalability (RQ3). (Chapter 1, Section 4.2)
3. Design and development	Multi-stage pipeline with WorldModel intermediate representation, multi-phase LLM extraction, structural validation, OCI-driven fixup, deterministic compilation, and repair mechanisms. (Chapter 5, Chapter 6)
4. Demonstration	The framework is applied to 30 benchmark scenarios across three difficulty tiers, producing deployed container environments. (Chapter 7, Section 7.2)
5. Evaluation	Benchmark-based evaluation measuring deployability, scenario fit, repair effectiveness, and complexity sensitivity against machine-readable references with operationalised metrics. (Chapter 7)
6. Communication	This thesis document.

Table 2. Mapping of DSRM process steps [64] to thesis activities.

The thesis follows a problem-centred entry point (step 1), which Peffers et al. identify as the most common starting point for academic research projects.

4.1.3 Design Science Guidelines

Hevner et al. [65] define seven guidelines for design science research. All seven are addressed below.

Guideline 1 (Design as an Artefact). The thesis produces a concrete software artefact: the honeynet generation framework described in Chapters 5 and 6.

Guideline 2 (Problem Relevance). The artefact addresses the practical problem that manual honeynet construction does not scale across multiple network segments or evolving threat landscapes, while existing template-based tools cannot accept unstructured natural-language descriptions of the target environment. This relevance is established in Chapter 1.

Guideline 3 (Design Evaluation). Hevner et al. list five families of evaluation methods (observational, analytical, experimental, testing, descriptive). This thesis uses two of them. Under *Testing*, the framework is evaluated by *functional (black-box) testing*, that is, executing the artefact through its public interface on the 30-scenario benchmark corpus and measuring the resulting outputs against machine-readable references (Chapter 7). Under *Experimental*, the evaluation combines a *controlled experiment* (a fixed configuration applied to all scenarios) with *simulation* (synthetic benchmark scenarios that exercise the artefact in place of operational input). Hevner's *structural (white-box) testing* is not used in its strict sense (no execution-path coverage testing was performed), although a pytest suite of 338 cases validates individual components in isolation as a development-time safety net (Chapter 6, Section 6.7). Observational, analytical, and descriptive evaluation methods are not used.

Guideline 4 (Research Contributions). The thesis contributes at three levels: (a) the artefact itself as a situated implementation, (b) the benchmark corpus with formal references as a reusable evaluation instrument, and (c) design knowledge about where LLM-based infrastructure generation succeeds and where it fails. Following Gregor and Hevner [66], this positions the thesis between a Level 1 contribution (situated implementation) and a Level 2 contribution (nascent design theory in the form of extracted design principles).

Guideline 5 (Research Rigour). Rigour is pursued through operationalised metrics with explicit computation rules, rationale, and documented limitations (Chapter 7, Section 7.1).

Guideline 6 (Design as a Search Process). The framework was developed iteratively. Design choices such as multi-phase extraction, multi-level repair, pre-deploy probing, and OCI-driven fixup emerged from failure modes observed during preliminary development runs and were retained when they improved deployability or scenario fit. The retrospective rationale for each major design choice is discussed in Chapter 5, Section 5.10.

Guideline 7 (Communication of Research). The work is communicated through this thesis document, which combines a broader framing for non-specialist readers (Chapters 1, 2, and 8) with the technical detail required for reproduction (Chapters 5 and 6).

4.1.4 Evaluation Strategy

The evaluation follows the *Technical Risk and Efficacy* strategy from the FEDS framework [67]. This strategy is appropriate when technical performance risks dominate and when the artefact does not directly interact with end users during evaluation.

The evaluation is *ex post* (performed after the artefact is built) and *artificial* (performed in a controlled benchmark setting rather than in a production environment). Venable et al. [67] note that artificial evaluation is necessarily "unreal" along one or more dimensions (users, systems, problems). In this thesis, the evaluation uses synthetic benchmark scenarios rather than operational honeynet deployment requirements, and no human operators interact with the generated honeynets. These limitations are explicitly acknowledged in Chapter 8, Section 8.3.

The evaluation generates valid research knowledge despite these limitations: it answers the research questions rigorously against operationalised metrics, identifies failure modes, and produces measurable evidence about quality attributes of LLM-based infrastructure generation.

4.1.5 Knowledge Contribution

The design knowledge produced by this thesis takes three forms:

1. **Architectural knowledge:** Which pipeline stages (extraction, validation, fixup, compilation, deployment, verification) are necessary to bridge the gap between raw LLM output and deployable infrastructure, and how should they be composed?
2. **Evaluative knowledge:** Which quality dimensions (deployability, scenario fit, repair effectiveness, complexity sensitivity) are meaningful for assessing LLM-based infrastructure generation, and how can they be operationalised?
3. **Empirical knowledge:** Under what conditions does LLM-based honeynet generation succeed (well-known technology stacks, moderate complexity) and where does it fail (domain-specific services, complex dependency wiring, zone placement in multi-zone architectures)?

This knowledge is prescriptive: it provides guidance for future researchers and practitioners designing similar systems, independent of whether the specific framework artefact is reused.

4.2 Research Objectives

The research objectives are derived from the problem identification in Chapter 1 and operationalised through three research questions (RQ1 deployability, RQ2 fidelity, RQ3 scalability), stated in Chapter 1. Their operationalisation into measurable metrics is defined in Chapter 7, Section 7.1.

4.3 Functional Requirements

The following functional requirements define the core capabilities that the framework must provide to address the research questions:

FR1 (Prompt Processing): The framework must accept a natural-language description of a target infrastructure and produce a structured intermediate representation (WorldModel).

FR2 (Structural Validation): The framework must validate the extracted WorldModel for internal consistency: referential integrity of dependencies and zone assignments, DAG property of the dependency graph, and syntactic validity of image references.

FR3 (Image Resolution): The framework must validate container images against OCI registries and correct configurations (commands, environment variables, healthchecks) based on actual image metadata.

FR4 (Deterministic Compilation): The framework must transform the validated WorldModel into a deployment projection and render it as an OpenTofu artefact. The compilation must be a pure function: the same input must always produce the same output.

FR5 (Deployment): The framework must execute the rendered artefact through the OpenTofu workflow (plan, apply) and produce running Docker containers.

FR6 (Repair): The framework must detect and attempt to recover from failures at the validation, image resolution, and deployment stages through deterministic fixups and, where necessary, LLM re-invocation.

FR7 (Verification): The framework must verify the health of deployed containers through Docker status queries, TCP connectivity checks, and HTTP status checks.

FR8 (Evaluation): The framework must compare the generated WorldModel against machine-readable benchmark references and compute scenario-fit metrics (service recall, zone coverage, dependency satisfaction, placement violations).

Non-functional requirements are not enumerated as a separate list in this thesis. The quality attributes that would normally be expressed as non-functional requirements are satisfied implicitly through specific design choices: reproducibility is the purity property of FR4 (deterministic compilation), portability is provided by the Python and Docker runtime stack, openness is provided by the MPL-licensed OpenTofu and the open-source nature of the framework itself (Chapter 5, Section 5.6.1), and performance is measured (mean pipeline duration of 616 seconds per scenario, reported in Chapter 7) but is not specified as a hard target.

4.4 Benchmark Corpus Construction

The evaluation uses a benchmark corpus of 30 scenarios constructed by the author. The corpus size of 30 scenarios (10 per difficulty tier) was chosen as a pragmatic compromise between domain coverage and execution cost: a full corpus run requires approximately 5 hours of wall-clock time (mean 616 seconds per scenario including container creation and teardown). The equal distribution of 10 scenarios per tier ensures that per-tier statistics are based on comparable sample sizes, which simplifies cross-tier comparison.

Each scenario consists of two components: a natural-language prompt describing a target infrastructure, and a machine-readable reference specifying the minimum structural requirements that a correct generation result must satisfy.

4.4.1 Scenario Design

Scenarios were designed to cover three dimensions of variation:

1. **Topological complexity:** The number of required services (3–11), zones (2–6), and inter-service dependencies (2–6) increases across tiers.
2. **Domain diversity:** Scenarios span multiple infrastructure domains: web applications, identity management, monitoring, message queuing, object storage, DNS, VPN, file sharing, CI/CD, e-commerce, healthcare, DevOps, supply chain, streaming, SCADA, cryptocurrency, advertising, gaming, banking, telecom, machine learning, zero-trust, IoT, government, satellite, smart city, defence, and pharmaceutical research.
3. **Technology specificity:** Some prompts name concrete technologies (“PostgreSQL,” “Redis,” “Kafka”), while others use abstract descriptions (“a recursive DNS resolver that redirects suspicious domains”).

4.4.2 Difficulty Tiers

Scenarios are classified into three tiers based on the structural complexity of their benchmark reference:

- **Easy** (e01–e10): 3–4 required services, 2–3 zones, 2 dependencies.
- **Medium** (m01–m10): 5–7 required services, 3–4 zones, 3–4 dependencies.
- **Hard** (h01–h10): 8–11 required services, 4–6 zones, 5–6 dependencies.

The tier classification is based on the size of the benchmark reference (number of required services, zones, and dependencies), not on the expected difficulty for the LLM. As discussed in Chapter 8, LLM generation difficulty depends on domain familiarity rather than topological complexity alone.

4.4.3 Benchmark References

Each benchmark reference is a YAML document specifying:

- **Required services:** Each with matching criteria (`kinds_any`, `archetypes_any`, `names_any`, `roles_any`) that define how generated systems are matched to required services.
- **Required zones:** Each with exposure requirements (`exposure_any`, `internal` flag, `names_any`).
- **Required dependencies:** Directed edges between service identifiers specifying which services must declare dependencies on which other services.
- **Forbidden placements:** Pairs of (service, zone) specifying that a service must not be placed in a particular zone.

The references define *minimum* requirements, not exact specifications. A generated honeynet may contain additional services, zones, and dependencies beyond what the reference requires, and still pass the formal benchmark.

4.4.4 Benchmark Construction Process and Bias Mitigation

The benchmark scenarios and their references were constructed by the author. This introduces a potential bias risk: the same person who designs the evaluation criteria also designs the system being evaluated. To mitigate this risk, the following measures were taken:

- The benchmark references were defined based on standard infrastructure patterns documented in vendor documentation and reference architectures, not based on the framework's observed output.
- The matching criteria in the references use broad categories (`kinds_any`, `archetypes_any`) rather than exact system names, so that the framework is not rewarded for generating specific naming conventions.
- The 30 scenarios were assigned to difficulty tiers based on structural properties of the reference (service count, zone count, dependency count) before evaluation runs were conducted. The tier assignments were not revised after observing the results.

Despite these measures, the benchmark remains author-constructed and has not been independently validated. This limitation is discussed in Chapter 8, Section 8.3.

4.5 Evaluation Design

4.5.1 Evaluation Dimensions

The evaluation assesses the framework across four dimensions:

1. **Deployability** (RQ1): Does the generated infrastructure survive the pipeline stages from validation to runtime verification?
2. **Scenario fit** (RQ2): Does the generated result match the structural requirements of the input scenario?
3. **Repair effectiveness** (supports RQ1): Can the framework recover from generation and deployment failures?
4. **Complexity sensitivity** (RQ3): How do the above dimensions change across difficulty tiers?

The operationalisation of these dimensions into specific metrics (computation, rationale, limitations) is defined in Chapter 7, Section 7.1.

4.5.2 Execution Protocol

Each of the 30 scenarios is executed once through the full pipeline. The single-run design is a consequence of the deployment cost per scenario (mean duration: 616 seconds). The implications of this design choice for result interpretation are discussed in Chapter 8, Section 8.3.

4.5.3 Scope and Exclusions

The evaluation is limited to the following scope:

- **Structural evaluation only.** The evaluation measures whether the correct services, zones, and dependencies are present. It does not measure interactive deception fidelity (whether the honeynet would be convincing to an attacker).
- **Single LLM model.** All scenarios are evaluated with one LLM (GPT-4o). Multi-model comparison is outside the scope of this evaluation.
- **No ablation study.** The evaluation reports the full-pipeline configuration only. Comparison against reduced configurations (e.g., pipeline without repair) is not included.
- **Benchmark scenarios only.** The evaluation uses synthetic benchmark scenarios, not operational deployment requirements from real organisations.

These exclusions are deliberate constraints that keep the evaluation tractable within the scope of a master's thesis. They are discussed as limitations in Chapter 8.

5. Conceptual Design of the Honeynet Framework

This chapter presents the conceptual design of the honeynet generation framework. Section 5.1 presents the overall pipeline architecture. Section 5.2 defines the core data models. Section 5.3 describes the world model extraction from natural language. Section 5.4 covers validation and repair. Section 5.5 describes the deterministic compilation into infrastructure-as-code artefacts. Section 5.6 covers deployment and runtime verification. Section 5.7 describes the post-deployment quality assurance layer. The design decisions presented here are evaluated empirically in Chapter 7.

5.1 Overall Architecture and Pipeline

The framework generates a running container-based honeynet environment from a natural-language description of a target infrastructure. This transformation proceeds through a sequence of stages, each producing an inspectable intermediate artefact. The design enforces two principles: each stage must be independently testable, and all intermediate representations must be human-readable. The human-readability requirement is a debuggability and auditability choice rather than a functional one: when a downstream stage produces unexpected output, the operator must be able to inspect the upstream artefact (WorldModel YAML, DeployProjection, OpenTofu JSON) directly without running additional decoding tools. This is particularly important in a research context where understanding *why* the pipeline produced a particular result is as important as the result itself, and it simplifies manual review when LLM output causes downstream failures.

Ahmed et al. [38] present a similar separation of concerns for cybersecurity lab generation, where specialised agents handle scenario development, environment generation, and validation as separate stages. The framework presented here adopts this stage separation but replaces the agent-per-machine model with a pipeline of stages that construct shared intermediate representations.

The pipeline architecture is shown in Figure 2. All data processing stages are represented by rectangular nodes, and intermediate artefacts transmitted between stages are represented by rounded nodes. The repair feedback loop described in Section 5.4.5 is indicated by the dashed arrow on the right. When structural validation fails, the framework applies deterministic repair strategies (e.g., command-policy corrections); if these do not resolve the errors, the pipeline aborts. The steps are listed below; each step is explained in depth

in Sections 5.3 through 5.7.

The pipeline architecture was chosen over two alternatives: a rule-based or template-driven approach where either discovered services are mapped to predefined honeypot images through fixed configuration rules [8] or administrators manually instantiate configurations from a fixed catalogue [9], and an agent-per-machine model, where each system is generated by an independent LLM agent [38]. The pipeline strategy was chosen because it preserves determinism in the compilation and deployment stages, enables independent testing of each stage, and generates inspectable intermediate artefacts at every step. The trade-off is that while an agent architecture could enable runtime coordination between independent systems, the pipeline trades this flexibility for reproducibility and staged validation.

The pipeline consists of the following stages:

1. **Extraction and OCI fixup**: The LLM receives the prompt together with pre-fetched OCI image profiles and produces a WorldModel in a single call or across three focused phases. Immediately after the LLM call, deterministic post-extraction fixups correct the WorldModel: unnecessary commands are stripped from daemon images, missing required environment variables are added, fragile healthchecks are rewritten, missing dependency edges are inferred from environment variable values, and backend services placed in public zones are moved to internal zones (Section 5.3).
2. **Structural validation**: The WorldModel is checked for undefined `depends_on` targets, dependency cycles, and command-policy violations.
3. **Image resolution**: Each container image is validated against OCI registries via HTTP HEAD requests. Successfully resolved images are pinned to their content digest for reproducibility.
4. **Deterministic compilation**: The WorldModel is translated into a DeployProjection, resolving host port allocations and container dependency ordering.
5. **Pre-deploy probing**: Each container in the DeployProjection is briefly started to verify it remains running. Containers that exit are subjected to deterministic and LLM-assisted repair; those that remain unfixable are removed and the projection is recompiled.
6. **IaC rendering**: The DeployProjection is serialized as an OpenTofu JSON configuration.
7. **Deployment**: `tofu plan` and `tofu apply` instantiate the Docker networks and containers.
8. **Runtime verification and QA**: Container status, TCP reachability, and HTTP

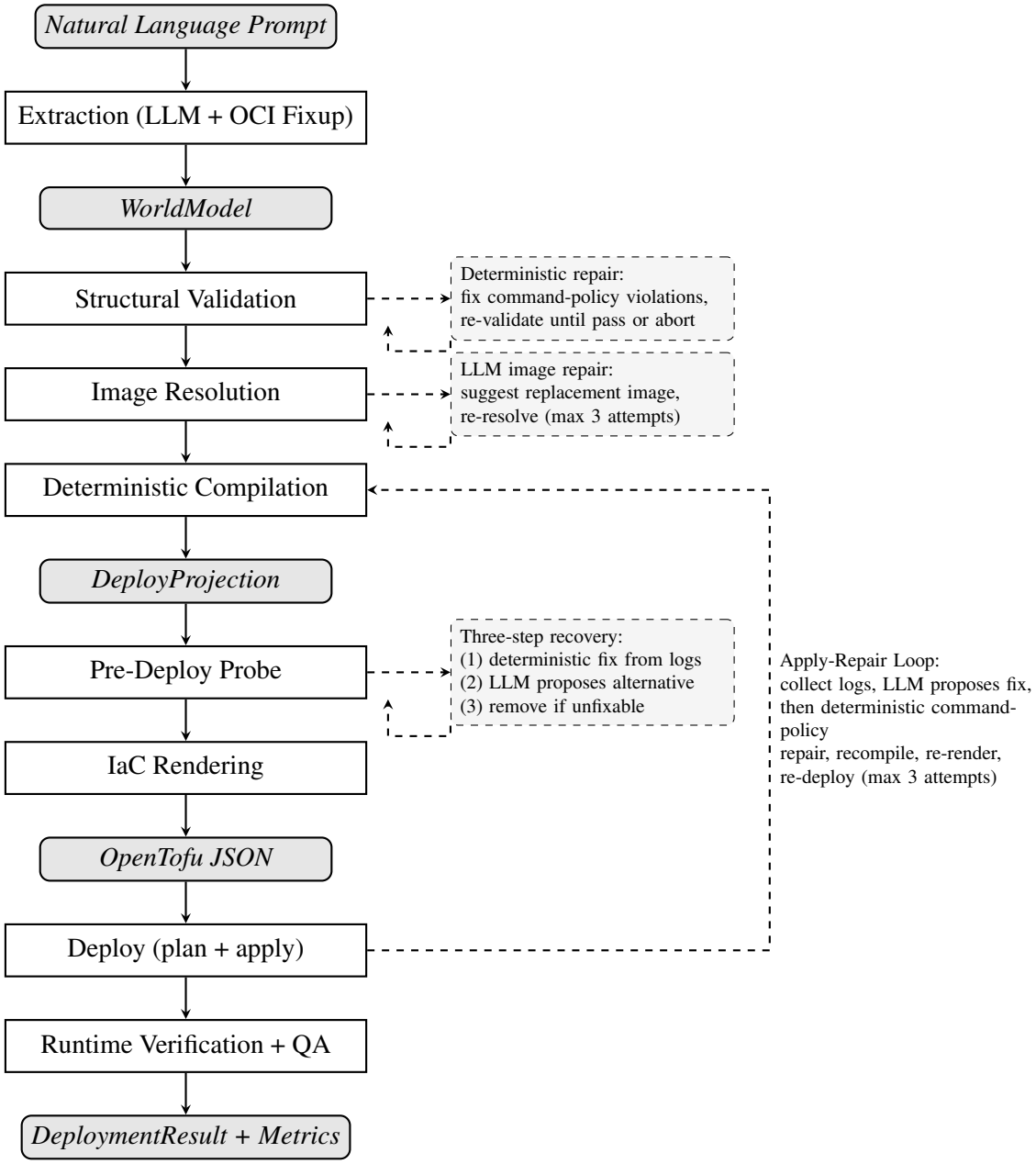

Figure 2. Pipeline architecture from natural language input to deployed honeynet. Rectangular nodes are processing stages; rounded nodes are intermediate artefacts. Dashed arrows on the right indicate repair loops at four pipeline stages, each with increasing cost: structural validation applies deterministic fixes; image resolution asks the LLM for replacement images; the pre-deploy probe applies a three-step recovery (deterministic, LLM, removal); and the apply-repair loop recompiles and redeploys after collecting container failure logs. Details are described in Section 5.4.5.

response codes are checked against the deployment projection.

This staged design is consistent with the observation by Beuter et al. [42] that LLM-generated container orchestration manifests require error-report-aware refinement processes, and with the finding by Palavalli and Santolucito [28] that feedback loops returning validation errors to the LLM can improve IaC generation quality, though the effectiveness of additional iterations decreases with diminishing returns.

5.2 Core Data Models

The pipeline stages described above operate on three primary data representations, each serving a distinct purpose in the transformation from natural language to deployed infrastructure.

5.2.1 WorldModel

The WorldModel is the central intermediate representation. It captures the full specification of the honeynet as extracted by the LLM, before any deployment-specific transformation. The use of a structured intermediate representation between natural language and deployment artefacts follows a design pattern that has been applied in compiler engineering [68] and in domain-specific languages for cyber range generation [36].

The WorldModel contains four components:

- **Organization**: Metadata about the simulated entity (name, type, regions, business units). This component is used for narrative coherence and artefact naming but has no effect on deployment.
- **Systems**: A dictionary of named systems, each representing a container to be deployed.
- **Zones**: A dictionary of named network zones, each representing a Docker network.
- **Secrets**: A dictionary of named secrets representing credentials, API keys, and tokens. The dictionary holds both real configuration credentials that the deployed services need to start (such as database passwords) and simulated honeytokens planted as bait for attackers to discover. The framework distinguishes the two through metadata in each secret's simulate block.

5.2.2 System Model

Each system in the WorldModel has two distinct sections that serve different purposes:

Deploy section. Contains all information required for deterministic infrastructure generation: container image, network zone assignment, exposed ports, environment variables, volume mounts, startup command, dependency declarations, and healthcheck specification.

Simulate section. Contains behavioural metadata for deception purposes: realistic hostname, role description, simulated vulnerabilities, known secrets, and behavioural patterns. This section is never read by the compilation or deployment stages.

The separation of deployment mechanics from simulation semantics ensures that the deterministic compilation stage reads only from the deploy section. This invariant guarantees that the same WorldModel always produces the same deployment artefact, regardless of changes to the simulate section.

The deploy section contains the following fields:

- **image**: A Docker image reference (e.g., `postgres:16-alpine`).
- **zone**: The name of the zone this system is assigned to.
- **ports**: A list of internal container ports to expose.
- **env**: A list of environment variable assignments in `KEY=value` format.
- **volumes**: A list of volume mount specifications.
- **command**: An optional command override for the container entrypoint.
- **depends_on**: A list of system names that this system depends on, determining startup ordering and connectivity.
- **healthcheck**: A Docker healthcheck specification defining the test command, interval, timeout, retries, and start period.

5.2.3 Zone Model

Each zone represents a Docker network with deployment properties (network name, driver, internal flag, optional subnet) and optional simulation properties (exposure level, sensitivity, trust level).

Zones enforce network segmentation. A zone with `internal=true` creates a Docker

network that is not reachable from the host network, isolating backend services from external access. The framework enforces placement constraints: databases, message brokers, and identity services must be assigned to internal zones. These constraints are enforced as a post-extraction fixup that automatically moves misplaced backend services into internal zones (Section 5.3).

This zone-based segmentation follows the architectural patterns described by Javadpour et al. [60], who identify network isolation as a key property of effective honeynet deployments, and by Kim et al. [69], who use graph-based topology generation for network segmentation in cyber exercise systems.

5.2.4 DeployProjection

The DeployProjection is the output of the deterministic compilation stage. It contains:

- **Networks**: Docker network definitions derived from zones.
- **Containers**: Container definitions derived from systems, with allocated host ports, resolved image references, and explicit dependency edges.
- **Constraints**: Terraform-specific constraints (provider source, provider version, Docker host endpoint).

The compilation from WorldModel to DeployProjection is a pure function: given the same WorldModel and the same set of already-occupied host ports, it always produces the same DeployProjection. This determinism is a design choice intended to enable reproducibility.

5.3 World Model Extraction

The extraction stage translates a natural-language prompt into a WorldModel. The framework supports two extraction modes: single-call and multi-phase.

5.3.1 Single-Call Extraction

In single-call mode, the LLM receives the full prompt, a system message containing generation rules, and pre-fetched OCI image configurations, and produces a complete WorldModel as a YAML document in a single invocation.

The extraction stage provides the LLM with two tools: `validate_docker_image` (checks whether an image reference exists in a registry) and `fetch_image_config` (re-

trieves the OCI configuration blob for an image, including entrypoint, command, exposed ports, and environment variables).

The framework injects pre-fetched image configurations into the system prompt to reduce hallucination. This approach is consistent with the finding by Béchard and Marquez Ayala [70] that retrieval-augmented generation reduces hallucination in structured output tasks. The image selection uses a hybrid BM25 and lexical ranking scheme over a curated registry of known-good images, with keyword annotations mapping technology terms (e.g., "cache", "database") to image categories.

The selection procedure works as follows. The curated registry (64 image categories) stores, for each category, one or more validated image tags and a list of domain keyword annotations: the `redis` category is annotated with "cache", "session", "kv", "queue", "pubsub"; the `postgres` category with "database", "sql", "transaction"; and so on. At retrieval time, the user prompt is tokenized and each category is scored in two independent channels: (1) a BM25 score over the category name, tag tokens, and keyword annotations treated as a single document, and (2) a lexical overlap score counting direct token matches between the prompt and the category's annotations. The two ranked lists are combined through reciprocal rank fusion, and the top-K categories are selected (adaptively between 20 and 50, scaled by prompt complexity). For each selected category, the framework fetches the OCI configuration (entrypoint, default command, exposed ports, required environment variables) from the image registry using concurrent HTTP HEAD requests, and injects this information directly into the prompt so the LLM can reference real image metadata rather than generating plausible-sounding but incorrect values. Images not covered by the registry can still be used: the LLM may request validation of an unknown image through the `validate_docker_image` tool, and the post-extraction image resolver validates every image regardless of whether it was pre-fetched.

The generation is guided by a set of rules embedded in the system prompt. These rules constrain the LLM output to produce deployable configurations:

- Base runtime images (Python, Node.js, Ruby) must not be used as application services unless they include a built-in server module.
- Every container must run a long-lived daemon as PID 1; one-shot commands (identified from a maintained list of known one-shot binaries such as `pg_dump` and `mysqldump`) are rejected.
- Every system must declare a healthcheck.
- All systems referenced in environment variables must appear in the `depends_on` list.

- Databases, message brokers, and identity services must be placed in internal zones.

After the LLM call completes (in either extraction mode), the framework applies three deterministic post-extraction fixups: (1) the OCI fixup described in Section 5.4.3 strips unnecessary commands, adds required environment variables, and rewrites fragile healthchecks; (2) dependency inference scans environment variable values for references to other system names and adds missing `depends_on` edges; and (3) placement correction moves backend services (databases, message brokers, identity services) from public to internal zones based on OCI port analysis. These fixups require no LLM call and produce deterministic corrections.

5.3.2 Multi-Phase Extraction

For complex scenarios, the framework offers a three-phase extraction mode that splits generation into focused sub-tasks. This decomposition addresses the problem that a single LLM call must simultaneously handle architecture design, image selection, environment configuration, healthchecks, dependencies, and zone placement, which can lead to token budget exhaustion and truncated output for scenarios with many systems.

The multi-agent decomposition pattern is well-established in recent code generation research, where task decomposition into specialised agents consistently improves quality (see Chapter 3, Section 3.4).

The three phases are:

Phase 1: Architecture

The first phase generates the topology skeleton: organisation metadata, zone definitions, and system stubs with kind, zone assignment, and dependency edges. No deployment configurations (images, ports, environment variables) are generated in this phase. The output token budget is approximately 80 tokens per system plus a 500-token overhead, keeping the output well below provider limits even for large scenarios.

Phase 2: Configuration

The second phase adds deployment configurations to each system: image selection, port declarations, environment variables, healthchecks, and volume mounts. Systems are processed in batches of up to 8, grouped by zone affinity to maintain contextual coherence. Each batch receives the pre-fetched OCI configurations for scenario-relevant images. The token budget per batch is approximately 500 tokens per system plus a 1000-token overhead.

Phase 3: Enrichment

The third phase adds deception metadata: simulation blocks (hostnames, roles, behavioural patterns), secrets, and optional companion systems (admin interfaces, log collectors, backup agents). The token budget is approximately 150 tokens per system plus a 2000-token overhead. Phase 3 failure is non-fatal: if the enrichment call fails, the WorldModel from Phases 1 and 2 is used as-is. This design ensures that a Phase 3 failure does not prevent deployment of the infrastructure generated in Phases 1 and 2.

Merging

The outputs of the three phases are merged sequentially. Phase 2 deployment configurations are overlaid onto the Phase 1 skeleton. Phase 3 simulation blocks and companion systems are added to the merged result. Companion systems that reference non-existent zones are assigned to the first available internal zone as a fallback.

Figure 3 illustrates the three-phase extraction process and the data flow between phases.

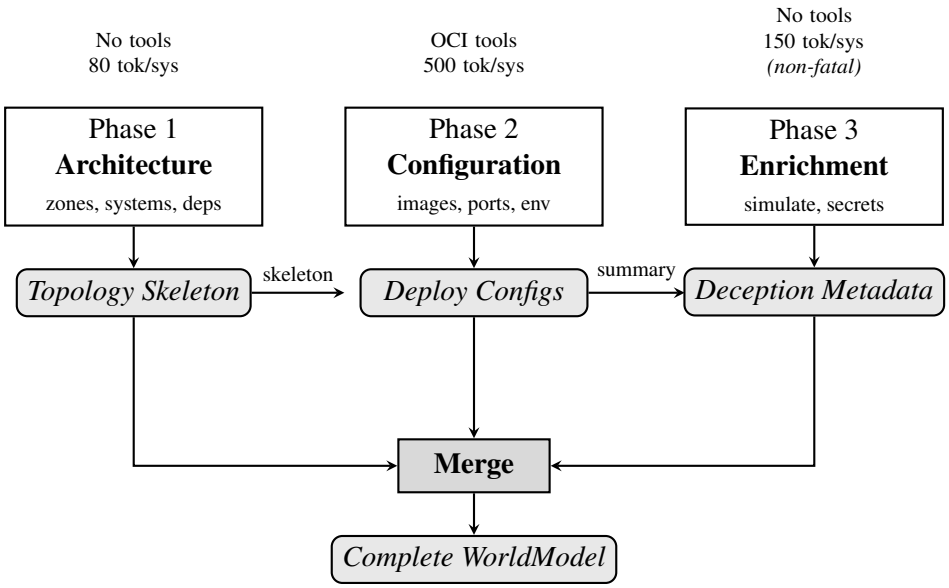

Figure 3. Multi-phase extraction process. Phase 1 generates the topology skeleton (zones, systems, dependencies). Phase 2 adds deployment configurations (images, ports, environment variables) in batches with OCI tool access. Phase 3 adds deception metadata (simulate blocks, secrets, companion systems). Phase 3 failure is non-fatal. The merge step overlays the outputs sequentially.

5.4 Validation and Repair

The framework applies multiple layers of validation and repair to compensate for the inherent unreliability of LLM-generated outputs. Srivatsa et al. [12] report that LLMs achieve 50–60% success rates on IaC generation tasks, and Kon et al. [11] find that GPT-4

achieves only 19.36% pass@1 on the IaC-Eval benchmark, confirming that raw LLM output is insufficient for deployment without post-processing.

5.4.1 Structural Validation

After extraction, the WorldModel is checked against a registry of structural rules:

- All `depends_on` references must point to existing systems (referential integrity).
- All system zone assignments must reference existing zones.
- Every system must declare a Docker image reference.
- Dependencies must not form circular chains.
- No self-dependencies.
- All declared ports must be in valid range (1–65535).
- Network names derived from zones must be unique.
- At least one zone and one deployable system must be present.
- Container commands must not violate the command policy.

Hard failures that prevent compilation are classified as validation errors; the command-policy rule is an error that triggers automatic repair. Validation warnings, such as one-shot commands used as container entrypoints or references to non-existent script files in interpreter images, are captured but do not block the pipeline by default.

If structural validation fails, the framework applies deterministic repair strategies (e.g., correcting command-policy violations detected by the validator). If repair succeeds, validation is re-run to confirm. If repair does not resolve the errors, the pipeline aborts. The framework does not re-invoke the LLM for structural repair; this design follows the finding by Palavalli and Santolucito [28] that the effectiveness of feedback loops exhibits diminishing returns per iteration.

5.4.2 Consistency Fixup

After structural validation passes, a deterministic consistency fixup step applies safe mutations to the WorldModel. This step auto-creates missing zone definitions when a system references a zone name that does not yet exist (assigning default internal properties), and detects bare secret values embedded in environment variables that should instead reference named secrets. These fixes are logged but do not require LLM re-invocation.

5.4.3 OCI Image Fixup

The framework fetches the OCI configuration for each container image from the registry. The OCI configuration contains the image's entrypoint, default command, exposed ports, and required environment variables. Based on this data, the framework applies deterministic corrections:

1. **Command stripping**: If the image has a daemon entrypoint (e.g., `nginx -g "daemon off;"`) but the LLM specified a command that would override it, the command is removed.
2. **Required environment variables**: If the OCI configuration declares required variables that are not set in the WorldModel, placeholder values are added (e.g., `POSTGRES_PASSWORD=changeme`).
3. **Healthcheck rewriting**: Fragile healthchecks (e.g., relying on `curl` alone) are rewritten to use service-native CLIs where available. A lookup table maps image families to preferred healthcheck commands (e.g., `postgres` → `pg_isready`, `redis` → `redis-cli ping`). For images known to lack common shell utilities (e.g., distroless images, Traefik, CoreDNS), healthchecks are rewritten to use TCP probes.

This OCI-driven correction layer addresses a category of errors that Liu et al. [10] classify as "resource hallucinations" in their taxonomy of LLM code hallucinations: the LLM generates references to APIs, parameters, or configurations that do not exist in the target system.

5.4.4 Image Resolution

The framework validates each container image against the Docker registry using the OCI Distribution Specification.[1] The framework performs HTTP HEAD requests against `/v2/{repo}/manifests/{ref}` to verify image existence and retrieve content digests. This approach does not require a running Docker daemon and does not count towards pull rate limits.

Images that resolve successfully are pinned to their digest (`image@sha256:...`) to ensure reproducible deployments. Images that cannot be resolved are flagged; the framework attempts namespace alias fallbacks (e.g., `bitnami/kafka` → `kafka`, where the namespaced image is replaced by the official Docker Hub library image) and tag fallbacks

[1] `https://github.com/opencontainers/distribution-spec`

(`:latest` as last resort) before classifying an image as unresolvable.

5.4.5 Repair Architecture

The framework implements repair mechanisms at five levels:

1. **Structural repair**: When structural validation fails, deterministic repair strategies correct detected violations (e.g., command-policy violations are automatically fixed). If repair succeeds, validation is re-run to confirm; if not, the pipeline aborts. The framework does not re-invoke the LLM for structural repair.
2. **Image-level repair**: The OCI fixup strips unnecessary commands, adds required environment variables, and rewrites fragile healthchecks based on image metadata. A daemon-entrypoint guard prevents the LLM-specified command from overriding a known daemon entrypoint (e.g., an Nginx image whose entrypoint already starts the server does not need an explicit command).
3. **Pre-deploy probe repair**: Before deployment, each container is briefly started. If the container exits, the framework applies a three-step recovery: (a) deterministic repair based on container logs (e.g., missing environment variables, missing subcommands inferred from OCI profiles), (b) LLM-assisted repair if deterministic repair fails, and (c) container removal if neither repair strategy succeeds.
4. **Apply-repair loop**: When `tofu apply` fails due to container crashes, the framework collects container logs, applies deterministic fixes, recompiles, and retries. If deterministic repair is insufficient, the LLM may be invoked to propose image or command changes.
5. **Runtime repair**: After deployment, containers that are not running or are unhealthy are candidates for repair. The framework first attempts deterministic correction based on container logs, then LLM-assisted repair if deterministic repair fails. Containers that remain unfixable after both repair paths are removed, and the surviving systems are redeployed.

Repair proposals from the LLM are scored by a deterministic scorer that assigns a confidence value between 0.0 and 1.0 based on the failure type and whether the proposed image is in the known-good registry. The loop guard stops the repair process from cycling through the same failing images. Each system tracks the images that have already failed in earlier attempts. When the LLM proposes a new image, the guard checks the proposal against this list and rejects it if it has already been tried. The default "hybrid" mode splits the check in two: the most recent failure is compared directly against the proposal, while older failures are compared as a separate set. For example, suppose a web container has already failed with the images `nginx:1.20`, `nginx:1.21`, and most recently `nginx:1.22`.

If the LLM re-proposes `nginx:1.22`, the first part of the hybrid check (the most-recent comparison) flags the loop. If the LLM re-proposes `nginx:1.20` or `nginx:1.21`, the second part (the older-failures set) flags it. Splitting the check this way ensures that the most recent failure is never compared twice. A daemon-entrypoint guard additionally prevents repair proposals that would override a known daemon entrypoint with a conflicting command. Repair incidents are logged with content hashes for deduplication and post-hoc analysis.

When the framework removes a container, it does not also remove the containers that depended on it. It only removes the deleted system from the others' `depends_on` lists, so they can start without complaining about a missing dependency. Their environment variables are left as they were, so if a dependent container has `DB_HOST=postgres_db` and `postgres_db` is gone, the container will start but fail when it actually tries to connect. That runtime failure usually triggers another repair pass.

This multi-level repair design is informed by the finding of Wang et al. [71] that two-agent architectures with compiler feedback for iterative repair achieve substantial improvements over single-pass generation, and by the observation of Bi et al. [47] that compiler feedback used to iteratively align and fix errors can improve LLM-generated code by over 80%.

Figure 4 summarises the five repair levels, ordered by increasing cost from cheap deterministic fixes to expensive LLM-assisted repair with container removal.

5.5 Deterministic Compilation

The compilation stage transforms the validated WorldModel into a DeployProjection. The compilation is a pure function: the same WorldModel and the same set of occupied host ports always produce the same DeployProjection.

The compilation proceeds in five steps:

1. **Network extraction**: Each zone is mapped to a Docker network definition with the appropriate driver and internal flag. A backbone network is created as a fallback for containers that do not have a zone assignment.
2. **Container extraction**: Each system with a deploy section is mapped to a container definition with image, environment variables, volumes, command, and healthcheck.
3. **Cross-zone networking**: Containers with dependencies in other zones are joined to the dependency's zone network in addition to their own, enabling direct cross-zone communication. The backbone network serves as a fallback for containers without a

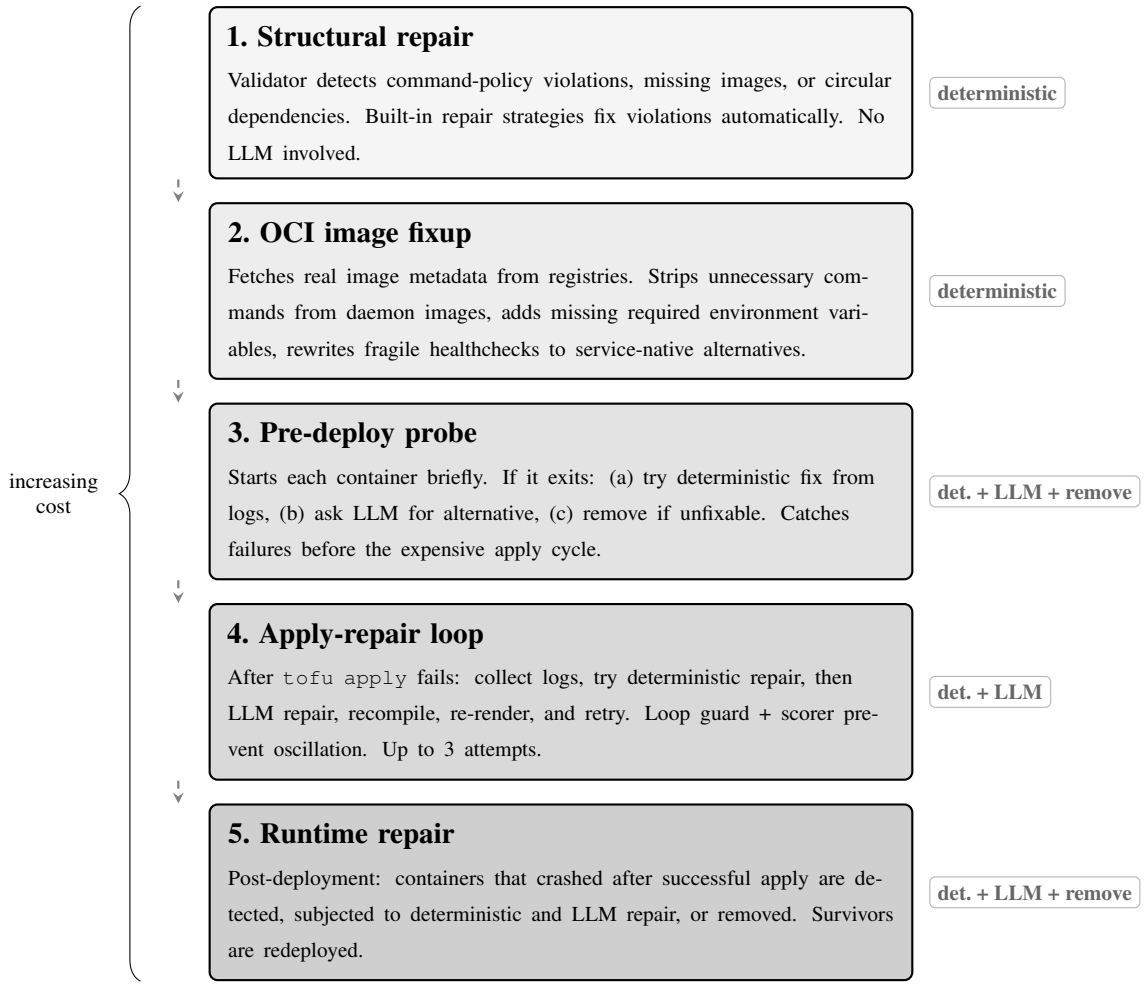

Figure 4. Five-level repair architecture, ordered by increasing cost. Levels 1 and 2 run during extraction and validation (before deployment) and are fully deterministic. Levels 3, 4, and 5 run during or after deployment and use the LLM as a fallback after deterministic repair fails. Levels 3 and 5 remove unfixable containers as a last resort.

zone assignment.

4. **Port allocation**: Each declared container port is assigned a unique host port, starting from the container's internal port (with a minimum of 1024) and skipping ports that are already in use. The set of occupied ports is seeded from the Docker daemon at compilation time.
5. **Dependency resolution**: The `depends_on` declarations are resolved to container references and emitted as explicit `depends_on` edges in the OpenTofu configuration; OpenTofu determines the startup ordering from these edges at apply time.

The DeployProjection is then rendered as an OpenTofu JSON configuration file using the `kreuzwerker/docker` provider.[2] Each zone becomes a `docker_network` resource; each system becomes a `docker_container` resource with explicit dependency edges.

The use of a deterministic compilation step that translates a higher-level model into infrastructure code follows the approach described by Costa et al. [36], where VSDL specifications are converted to deployment scripts via an intermediate model, and by Caturano et al. [37], who use automated extraction to generate Docker-based vulnerable environments.

5.6 Deployment and Runtime Verification

5.6.1 Choice of OpenTofu

This thesis is not about comparing infrastructure-as-code tools. With the right engineering, several alternatives could be made to work for honeynet deployment, but shell scripts and similar manual approaches would require substantially more code than a ready-to-use solution that already covers container and network lifecycle. OpenTofu was chosen for three practical reasons. It provides the `kreuzwerker/docker` provider, which handles container and network lifecycle out of the box. It is published under the Mozilla Public License, while Terraform since version 1.6 uses the more restrictive Business Source License. Being open source, it can be reused and extended freely by anyone who wants to build on this work.

[2] `https://registry.terraform.io/providers/kreuzwerker/docker`. OpenTofu maintains full compatibility with Terraform providers and HCL syntax; references to "Terraform providers" in this thesis refer to providers used through OpenTofu.

5.6.2 Deployment

The rendered JSON configuration is deployed through the standard OpenTofu workflow: `fmt`, `init`, `validate`, `plan`, and `apply`. The `fmt` step canonicalizes the generated JSON configuration before validation. The `apply` step is executed with auto-approval (the `-auto-approve` flag bypasses the interactive confirmation prompt that OpenTofu normally requires before modifying infrastructure); the plan is captured for audit purposes.

If the initial apply fails, the framework attempts recovery through three paths:

1. **Import recovery**: If Docker resources from a previous run still exist, the framework imports them into the Terraform state and retries the apply.
2. **Apply-repair loop**: If containers exit immediately after creation, the framework collects container logs and queries the LLM with the failure context to propose an alternative image or command. After the LLM repair, a deterministic command-policy repair pass corrects any remaining violations. The modified WorldModel is then recompiled, re-rendered, and re-applied. This cycle repeats up to a configurable maximum number of attempts.
3. **Partial apply**: If the full apply continues to fail after repair attempts, the framework identifies the failing resources and retries with a targeted `-target` apply that deploys only the surviving subset, preserving the health of the non-failing containers.

The iterative refinement pattern in the apply-repair loop follows the approach described by Tang et al. [72], where error messages from execution are fed back to guide correction. Tang et al. [49] formalize code repair as an exploration-exploitation tradeoff, where each attempt either refines the current solution or explores a fundamentally different approach.

Chouliaras et al. [32] describe a similar container-based deployment pipeline for cybersecurity training platforms, combining Docker technology with IaC methodology to automate environment provisioning. Kokolakis et al. [9] demonstrate automated honeypot deployment via Helm chart generation from high-level service specifications on Kubernetes.

5.6.3 Pre-Deploy Probing

Before deployment, the framework optionally probes each container image by starting it briefly and observing whether it remains running. Containers that exit within the probe window undergo a three-step recovery process: (1) deterministic repair based on container logs (e.g., adding missing environment variables, inferring missing subcommands from

usage output), (2) LLM-assisted repair if deterministic repair fails (the LLM proposes an alternative image or command), and (3) container removal if neither repair strategy succeeds. This pre-deploy probe serves as an early filter that prevents containers with known startup failures from entering the deployment.

The probe skips two categories of images that are expected to exit immediately and would produce false failures. Base runtime images (e.g., `python`, `node`) start an interactive interpreter that exits without a TTY, so probing them would always report a failure even though the image itself is valid. Config-dependent images (e.g., HAProxy, Envoy) require a mounted configuration file to start; without one, they exit with a file-not-found error that is not indicative of a generation defect.

5.6.4 Runtime Verification

After deployment, the framework queries the Docker daemon for the status of each expected container. This verification is entirely rule-based and does not involve any LLM queries. Containers that are not running or that are reported as unhealthy are first subjected to rule-based deterministic repair (correcting commands or environment variables based on container log analysis) and, if that fails, LLM-assisted repair (proposing an alternative image or command). Containers that remain unfixable after both repair paths are removed from the deployment, and the surviving systems are re-deployed. The framework records which containers were dropped and tracks the ratio of running containers to originally planned containers as a deployability metric.

5.7 Quality Assurance

After runtime verification, the framework executes two layers of QA checks against the deployed environment.

5.7.1 Connectivity and Health Checks

The first layer executes deterministic connectivity checks against each deployed container. Four check types are supported:

1. **Container running**: Queries the Docker daemon for container status.
2. **Dependency running**: Verifies that all declared upstream dependencies are in running state.
3. **TCP connect**: Attempts a TCP socket connection to each exposed port with a

configurable timeout (default: 5 seconds).

4. **HTTP status**: Sends an HTTP GET request to each exposed HTTP port and checks for a response with status code below 500.

Each check produces a binary pass/fail result. The aggregate QA pass rate is the ratio of passed checks to total checks executed. The QA results are recorded per check type.

5.7.2 Deception Surface Checks

The second layer assesses structural properties of the deployment's deception surface through heuristic checks: presence of honeytoken artefacts (weak credentials, bait files), narrative consistency of system hostnames and roles, variety of deployed service types, and absence of suspicious uniformity in configurations. These checks produce warning-level results and do not block the deployment pipeline.

The connectivity and deception checks do not assess interactive deception fidelity, specifically whether a deployed service would be convincing to an attacker during active engagement. Javadpour et al. [60] identify this dimension as a key factor in honeynet effectiveness. Assessing interactive fidelity would require adversarial testing, which is outside the scope of the current framework.

5.8 Scenario-Fit Evaluation

The framework supports two modes of scenario-fit evaluation that assess whether the generated infrastructure matches the structural requirements of the input scenario.

5.8.1 Prompt-Fit Evaluation

Prompt-fit evaluation uses rule-based keyword extraction (no LLM queries) to analyse the original natural-language prompt, identify required service types and network tiers, and then check whether the generated WorldModel contains matching systems and zones. This mode requires no external reference file and is always available.

5.8.2 Formal Benchmark Evaluation

When a machine-readable benchmark reference is provided (the references used in this thesis were constructed by the author; see Chapter 4, Section 4.4), the framework performs formal evaluation by comparing the generated WorldModel against the reference using

deterministic matching rules (no LLM queries). The reference specifies:

- **Required services**, described by one or more matching criteria. A generated system matches a required service if it satisfies any of the values in any of the criteria below.
  - **kind** — the broad type of system, like `web`, `database`, `identity`, `proxy`, `monitor`, or `logging`. Matches any system of that type regardless of which image was used.
  - **archetype** — the image family, like `postgres`, `nginx`, `fluentd`, or `filebeat`. Matches when the benchmark wants a specific implementation, not just a type.
  - **name pattern** — a piece of the system's name in the WorldModel. Matches when the benchmark wants a specifically named system.
  - **functional role** — a short role label from the simulate block, like `ldap`, `auth`, `admin`, or `management`. Matches systems by what they do, not what technology they use.
- **Required zones** with matching criteria: exposure level (internal vs. public) and name patterns.
- **Required dependencies** as directed edges between service identifiers.
- **Forbidden placements** specifying which services must not be deployed in which zones.

The formal evaluation computes service recall, zone coverage, dependency satisfaction, and placement violation counts. A composite scenario-fit score is computed as defined in Section 7.1.2 of Chapter 7.

5.9 Metrics and Observability

The framework records structured metrics at every pipeline stage. The metrics are grouped into three categories:

1. **Deployability metrics**: Boolean success flags and ratios for each pipeline stage (validation, plan, apply, runtime verification, QA).
2. **Scenario-fit metrics**: Coverage scores from both prompt-fit and formal benchmark evaluation, kept as separate fields so that running a formal benchmark evaluation does not overwrite the prompt-fit results (or vice versa), which would cause data loss if only one mode is used in a given run.
3. **Observability**: Run identifiers, stage durations, error classifications, repair incident counts, and repair action logs.

All metrics are serialized as a versioned JSON document and persisted alongside the deployment artefacts. The metric schema distinguishes "not measured" (stage never reached) from "measured as zero" to avoid conflation in downstream aggregation.

The metrics schema is designed to support the evaluation dimensions defined in Chapter 7. Each metric field is traceable to a specific pipeline stage or execution artefact.

5.10 Ablation Analysis

The pipeline architecture described in the preceding sections consists of multiple interacting stages, each contributing to the overall deployment quality. A full empirical ablation study, in which individual components would be systematically disabled and the benchmark corpus re-run, was not conducted due to the execution cost (each full corpus run requires approximately 5 hours; see Chapter 7). Instead, this section presents a *predicted* ablation analysis that derives the expected impact of removing each component from the component's role in the pipeline and from failure patterns observed during development. This follows the ablation study methodology common in machine learning evaluation [73].

Table 3 summarises the predicted effects of removing each major pipeline component. The predictions are derived from the component's role in the pipeline and from failure patterns observed during development.

5.10.1 Multi-phase vs. Single-Agent Extraction

The multi-phase extraction (Section 5.3.2) decomposes world model generation into three specialised LLM calls (architecture, configuration, enrichment) rather than producing the entire world model in a single call. This decomposition is motivated by two hypotheses:

1. **Token budget hypothesis**: Complex scenarios with 8–11 services may exceed the effective output capacity of a single LLM call, leading to truncated or incomplete configurations.
2. **Task focus hypothesis**: Separating topology decisions from configuration decisions allows each LLM call to focus on a narrower sub-task, reducing interference between design concerns.

The trade-off is increased LLM cost (three calls instead of one) and the need for a merging strategy that can handle conflicts between phases. Section 5.3.2 describes the merging strategy; the empirical comparison between single-call and multi-phase extraction requires

Component removed	Primary effect	Expected metric impact
Pre-deploy probing	Containers with startup failures enter deployment, causing OpenTofu apply errors	Apply success rate decreases; runtime failures increase proportionally to current pre-deploy drops (156 containers in the full corpus)
OCI image fixup	Missing environment variables, incorrect commands, and fragile healthchecks persist	Runtime failures increase; QA pass rate decreases due to healthcheck failures
Deterministic repair	Common failure patterns (missing env vars, wrong subcommands) are not auto-corrected during apply-repair loop	Deploy retry success rate decreases; container realisation ratio decreases
Multi-phase extraction	Single LLM call must handle architecture, configuration, and enrichment simultaneously	Service recall may decrease for complex scenarios due to token budget constraints; output truncation risk increases for hard-tier scenarios (8–11 services)
Consistency fixup	Missing zone definitions and embedded secrets not auto-corrected	Validation failure rate increases; repair loop is triggered more frequently
Structural validation	Invalid world models (undefined deps, cycles) reach compilation	Compilation errors increase; deployment of inconsistent configurations

Table 3. Predicted effects of removing individual pipeline components (ablation analysis). Each row describes the expected impact on system metrics if the component were disabled.

benchmark data from both modes, which is planned as future work (see Chapter 9).

5.10.2 Agent Tool Access

Each extraction phase has access to a defined set of tools that constrain the LLM's capabilities. Table 4 summarises the tool access per phase.

Phase	Tools available	Purpose
Phase 1 (Architecture)	None	Topology generation is a purely creative task; tool access would not improve zone/system design
Phase 2 (Configuration)	`validate_docker_image`, `fetch_image_config`	Image validation reduces hallucinated image references; OCI config retrieval enables correct port, env var, and healthcheck generation
Phase 3 (Enrichment)	None	Simulation metadata (hostnames, roles, secrets) does not require registry interaction

Table 4. Tool access per extraction phase. Only Phase 2 has tool access, as it is the only phase that must interact with external registries to produce correct deployment configurations.

The restriction of tool access to Phase 2 is a design choice: providing tools in Phase 1 would not improve topology generation (which is a creative task independent of image details), and Phase 3 enrichment data does not require registry validation. This selective tool access reduces the total number of tool calls and LLM latency while preserving the tool-assisted verification where it has the highest impact, namely during image selection and configuration.

The design described in this chapter defines what the framework should do; Chapter 6 describes how it is technically realised.

6. Implementation

This chapter describes the technical realisation of the framework design presented in Chapter 5. Section 6.1 provides a project overview. Section 6.2 describes the command-line interface and orchestration layer. Section 6.3 covers the LLM provider abstraction. Section 6.4 describes the extraction implementation. Section 6.5 covers validation and repair. Section 6.6 describes the compilation and deployment pipeline. Section 6.7 describes the testing infrastructure.

6.1 Project Overview

The framework is implemented in Python 3.10+ as an installable package (`honeynet_framework`) with a command-line entry point. The implementation comprises approximately 20,000 lines of Python across 69 modules, organised into six functional groups: extraction (6 modules), models (10 modules), catalogue (4 modules), detection (5 modules), plugins (3 modules), and core pipeline modules (41 modules).

Table 5 lists the principal modules and their sizes.

Module	Purpose	Lines
`orchestrator.py`	Pipeline orchestration	3,073
`llm.py`	LLM provider abstraction	1,252
`deployer.py`	OpenTofu execution	1,219
`extraction/prompts.py`	Prompt construction	953
`benchmark_runner.py`	Benchmark execution	851
`extraction/extractor.py`	WorldModel extraction	841
`image_resolver.py`	OCI registry validation	822
`extraction/multi_phase.py`	Multi-phase extraction	502
`cli.py`	Command-line interface	499
`retrieval.py`	BM25 image selection	432
`validator.py`	Structural validation	393
`qa.py`	Post-deployment QA	365
`deploy_compiler.py`	WorldModel → DeployProjection	304
`tofu_renderer.py`	DeployProjection → HCL	303
`metrics.py`	Metric tracking (schema v4)	277

Table 5. Principal modules by size. The remaining 54 modules (models, catalogue, detection, plugins, utilities) account for approximately 8,100 lines.

External dependencies are minimised. The framework requires five runtime packages: `httpx` (async HTTP client for LLM APIs and OCI registries), `PyYAML` (YAML

parsing), `prompt_toolkit` (interactive CLI), `rich` (terminal formatting), and `typing-extensions`. All three LLM provider implementations communicate directly via `httpx`; no vendor-specific SDKs are required.

6.2 Command-Line Interface and Orchestration

The CLI provides four commands: `deploy` (generate and deploy a honeynet from a prompt), `generate` (produce a WorldModel without deploying), `destroy` (tear down a deployed honeynet), and `benchmark run` (execute the benchmark corpus). All commands accept common arguments for LLM provider selection, model name, API key, work directory, and logging verbosity.

The `deploy` command instantiates an `HoneynetOrchestrator` with the provided configuration and calls its `deploy()` method, which executes the full pipeline. The orchestrator is the central coordination point: it sequences the pipeline stages, manages timing and metrics, handles error routing, and persists intermediate artefacts. The orchestrator's configuration (`OrchestratorConfig`) exposes 41 parameters controlling extraction mode, repair limits, validation strictness, plugin activation, artefact management, pre-deploy probe settings, QA timeout values, and deployment recovery options. All parameters have defaults that produce a functional configuration without user intervention.

6.3 LLM Provider Abstraction

The framework abstracts LLM interactions behind a provider interface (`LLMProvider`) with two methods: `generate()` for text-only generation and `generate_with_tools()` for tool-augmented generation. The framework provides three concrete implementations: `OllamaProvider` for locally hosted models via the Ollama API, `OpenAIProvider` for OpenAI's API, and `AnthropicProvider` for Anthropic's API. A factory function (`create_llm_provider`) returns the appropriate implementation based on a configuration object.

The provider interface returns a `TokenUsage` record (prompt tokens, completion tokens) with each response, enabling token-level cost tracking across pipeline stages. The multi-phase extraction pipeline accumulates token usage across all three phases.

Tool-augmented generation follows the function-calling protocol of each provider: the framework defines tool schemas (JSON Schema for parameters), passes them to the LLM alongside the prompt, and executes tool calls as they are returned. In the extraction stage,

two tools are registered: `validate_docker_image` (checks image existence via OCI HEAD requests) and `fetch_image_config` (retrieves OCI configuration blobs).

The `LLMConfig` dataclass centralises provider selection, model name, API key, API endpoint, temperature (default: 0.3), maximum output tokens (default: 8,192), and timeout (default: 1,200 seconds).

6.4 World Model Extraction

6.4.1 Prompt Construction

For every extraction call, the system message and user message are constructed by the prompt creation module (`extraction/prompts.py`, 953 lines). Generation rules (Section 5.3 of Chapter 5) are embedded as numbered constraints in the system message. The natural-language question, pre-fetched OCI picture settings, and scale recommendations based on a dynamic estimation function are all included in the user message.

The scale estimation function analyses the user prompt to infer an appropriate system count range. It tokenises the prompt, counts matches against technology-term patterns (databases, proxies, message brokers, monitoring, CI/CD), detects scale multipliers ("distributed," "cluster," "federated"), and applies complexity-based scaling factors. The estimated range is injected into the prompt as guidance (e.g., "generate between 12 and 18 systems").

6.4.2 Image Pre-Fetching

Before extraction, the framework selects scenario-relevant images from a curated registry (`data/known_good_images.json`, 64 service categories). The selection uses a hybrid BM25 and lexical ranking scheme: each image category is annotated with keywords (e.g., `redis` $\rightarrow$ "cache, session, kv, pubsub"), and the user prompt is scored against these annotations using reciprocal rank fusion.

The top-K images are then fetched concurrently from OCI registries (semaphore of 5 to limit concurrent connections), and their configurations (entrypoint, command, exposed ports, required environment variables) are injected into the system prompt. K is chosen based on two things: how many of the known image categories match words in the prompt, and how long the prompt is. K starts at 20 (so even a short prompt gets a useful set of images), grows by one for each matching category plus a small extra, and grows by another 5 or 10 if the prompt is longer than 40 or 80 words. K never goes above 50, which keeps

the number of registry requests and the size of the prompt within practical limits. The numbers 20 and 50, and the small step values, were chosen during development as sensible defaults and were not tuned through a parameter sweep.

This pre-fetching step provides the LLM with ground-truth image metadata, reducing the probability of hallucinated image references or incorrect environment variable names. Images requested in the prompt but absent from the curated registry are handled at the validation stage rather than at pre-fetch: the LLM generates an image reference from its own training knowledge, and that reference is then validated and, if necessary, repaired by the mechanisms described in Chapter 5, Sections 5.4.4 and 5.4.5.

6.4.3 Single-Call and Multi-Phase Extraction

Both extraction modes produce a raw YAML dictionary that is parsed and coerced into a WorldModel. The YAML parsing module (`extraction/yaml_parsing.py`) handles common LLM output artefacts: markdown fences around YAML blocks, trailing prose after the YAML document, and malformed quoting. The coercion module (`extraction/coercion.py`) normalises types: booleans represented as strings, integers as floats, and single values where lists are expected.

The multi-phase extraction (Chapter 5, Section 5.3.2) is implemented in `extraction/multi_phase.py` (502 lines). The `MultiPhaseExtractor` class coordinates three sequential LLM calls and maintains accumulated token usage. Phase 2 batching groups systems by zone affinity (systems sharing a zone are assigned to the same batch) with a maximum batch size of 8. The merge logic overlays Phase 2 deploy configurations onto Phase 1 skeleton systems and adds Phase 3 enrichment data (simulate blocks, secrets, companion systems).

Post-extraction, the orchestrator applies deterministic fixups: dependency inference from environment variable values (scanning for system names in `KEY=value` pairs), placement correction (moving backend services from public to internal zones), and OCI-driven command and healthcheck corrections.

6.5 Validation and Repair

6.5.1 Structural Validation

The validator (`validator.py`, 393 lines) implements a registry of named rules. Each rule is a function that receives the WorldModel and returns a list of errors or warnings. The rule registry includes: referential integrity (`depends_on` targets exist, zone assignments reference existing zones), DAG property (no dependency cycles, detected via iterative depth-first search), image presence validation (every system must declare an image), one-shot command detection (rejecting `pg_dump`, `mysqldump`, etc. as container entrypoints), fictional script detection (rejecting references to non-existent files in interpreter images), port range checks (all declared ports must be in the range 1–65535), and command-policy violations (container commands must not violate the command policy).

Errors block pipeline progression; warnings are logged but do not block. If validation fails, the framework applies deterministic repair strategies (e.g., correcting command-policy violations detected by the validator). If repair succeeds, validation is re-run to confirm; if not, the pipeline aborts. The framework does not re-invoke the LLM for structural repair.

6.5.2 Image Resolution

The image resolver (`image_resolver.py`, 822 lines) validates each container image against OCI registries using HTTP HEAD requests on `/v2/{repo}/manifests/{ref}`. Resolution proceeds through a five-stage fallback chain: (1) local Docker image cache, (2) remote OCI API with exponential-backoff retry for transient errors, (3) alternate registries (`ghcr.io`, `quay.io`) for namespaced images, (4) namespace alias fallback (e.g., `bitnami/kafka` → `kafka`, where the namespaced image is replaced by the official Docker Hub library image), and (5) `:latest` tag as last resort. Images that cannot be resolved after all fallbacks are flagged; the image-level repair mechanism (Section 5.4.5) then asks the LLM to suggest a replacement. Successfully resolved images are pinned to their content digest (`image@sha256:...`).

6.5.3 Healthcheck Rewriting

The healthcheck fixup module (`healthcheck_fixup.py`, 272 lines) rewrites fragile healthchecks based on image profiles. A lookup table maps image families to preferred healthcheck commands (e.g., `postgres` → `pg_isready`). Images known to lack common shell utilities (Traefik, CoreDNS, distroless images) receive TCP-based probes.

Single-tool probes without fallbacks (e.g., `curl` alone) are augmented with alternatives (`curl || wget || true`).

6.5.4 Repair Mechanisms

The repair architecture operates at three levels:

1. **Deterministic repair** (`orchestrator.py`): Pattern-matching on container startup logs identifies common failures. Missing environment variables detected from error messages (e.g., "did you forget to add `-e SLAPD_PASSWORD`") are added automatically. Usage text indicating a missing subcommand triggers command inference from the OCI profile (e.g., MinIO requires `server /data`).
2. **LLM-assisted repair** (`orchestrator.py`): When deterministic repair is insufficient, the LLM is invoked with the failure context (container logs, image profile, error classification) to propose an alternative image or command. Proposals are scored by the repair scorer (`repair_scorer.py`, 189 lines) based on failure type and known-good registry membership.
3. **Container removal**: Containers that cannot be repaired are removed from the WorldModel, and the deployment projection is recompiled and re-deployed with the surviving systems.

The loop guard (`loop_guard.py`, 159 lines) stops the LLM from proposing images that have already failed for the same system. It keeps a per-system list of failed images and, on each new proposal, checks whether the proposal is already in that list. The default "hybrid" mode runs the check in two parts: the most recent failure is compared directly (the "no-change" rule), and the earlier failures are compared as a separate set (the "image repeat" rule). An illustrated example is given in Chapter 5, Section 5.4.5. Separately, the orchestrator applies a daemon-entrypoint guard that rejects repair proposals suggesting a non-daemon image (e.g., a bare `python` runtime that would exit immediately) by checking the proposed image's OCI profile before applying the change.

6.6 Compilation and Deployment

6.6.1 Deterministic Compilation

The WorldModel is converted into a DeployProjection via the deploy compiler (`deploy_compiler.py`, 304 lines): systems become container definitions with assigned host ports, and zones become Docker network definitions. The Docker daemon is queried during

compilation to determine whether ports are already in use, and port allocation begins with each container's internal port (with a minimum of 1,024).

The compilation is implemented as a pure function: the same WorldModel and the same set of occupied ports always produce the same DeployProjection. This is achieved by not keeping any data between calls, so each call starts fresh. The thesis does not include a test that runs the compilation twice on the same input and compares the results, so the property is guaranteed by the way the code is written rather than checked by a test. Adding such a test is left as future work.

This holds only when the WorldModel does not change. If a container is removed during the repair loop, the surviving containers are recompiled. The compiler then assigns host ports again from scratch, so a survivor may end up on a different host port than before, including the port the removed container was using. The operator should not rely on host ports staying the same across the repair cycle.

6.6.2 HCL Rendering

The renderer (`tofu_renderer.py`, 303 lines) serialises the DeployProjection as an OpenTofu JSON configuration file (`main.tofu.json`). Each zone maps to a `docker_network` resource; each system maps to a `docker_container` resource with explicit dependency edges (`depends_on`). Container names are prefixed with `hn_` to avoid conflicts with existing containers.

6.6.3 Deployment Execution

The deployer (`deployer.py`, 1,219 lines) wraps the OpenTofu CLI. It executes `fmt`, `init`, `validate`, `plan`, and `apply` as subprocess calls with configurable timeouts. The deployer supports two execution modes: native (OpenTofu binary on the host) and containerized (OpenTofu running inside a Docker container, useful when the host does not have OpenTofu installed).

After `apply`, the deployer queries the Docker daemon for the status of each expected container. Containers that are not running or are reported as unhealthy are classified as runtime failures and may trigger the repair mechanisms described above.

6.7 Testing and Quality Assurance

6.7.1 Post-Deployment QA

The QA module (`qa.py`, 365 lines) executes four types of checks against each deployed container: `container_running` (Docker status query), `depends_on_running` (dependency target status), `tcp_connect` (TCP socket connection with configurable timeout and retry count), and `http_status` (HTTP GET request expecting status < 500).

The check set is generated automatically from the deployment projection: each container receives a `container_running` check; containers with declared ports receive `tcp_connect` checks; containers with HTTP-capable ports (80, 443, 8080, etc.) receive `http_status` checks; and containers with `depends_on` declarations receive `depends_on_running` checks.

6.7.2 Automated Test Suite

The test suite comprises 338 test cases across five test modules, executed with `pytest` in async mode (`asyncio_mode = "auto"`). Tests are organised into three categories:

1. **Unit tests**: Validate individual components in isolation (YAML parsing, type coercion, validator rules, metric computation, repair scoring). These tests use synthetic WorldModel fixtures and do not require LLM or Docker access.
2. **Integration tests**: Validate the interaction between pipeline stages (extraction $\rightarrow$ validation $\rightarrow$ compilation $\rightarrow$ rendering). These tests use mock LLM providers that return predefined YAML responses.
3. **Metrics schema tests**: Validate that the metrics output (schema version 4) produces the expected structure, ensuring backward compatibility when fields are added.

End-to-end tests that deploy to a live Docker daemon are not included in the automated test suite; the benchmark runner (`benchmark_runner.py`) serves this function during evaluation.

The implementation described in this chapter realises the conceptual design from Chapter 5. The evaluation in Chapter 7 measures the quality of the outputs produced by this implementation across the 30-scenario benchmark corpus.

7. Evaluation Framework

This chapter defines the evaluation metrics (Section 7.1), describes the experimental setup (Section 7.2), and reports the results (Section 7.3). The chapter presents measured data only; interpretation and discussion of the results follow in Chapter 8.

7.1 Evaluation Metrics

Each metric used in this evaluation is defined in terms of three properties: (1) what it measures and how it is computed, (2) why it is included as an evaluation dimension, and (3) what it does not capture.

7.1.1 Deployability Metrics

Deployability metrics track whether the generated infrastructure survives the concrete technical stages of the pipeline. Each metric corresponds to one pipeline stage and is recorded as a boolean or ratio.

World Model Validity Rate

Computation. After the LLM extraction phase, the generated world model is passed to the structural validator. The validator checks for: (a) all declared `depends_on` references pointing to existing systems, (b) all system zone assignments referencing existing zones, (c) all images being syntactically valid Docker references, and (d) no duplicate system or zone names. A world model that passes all checks is recorded as valid. The rate is: $\frac{\text{valid runs}}{\text{total runs}}$.

Justification. A structurally invalid world model cannot be compiled into a deployment artefact. This metric captures whether the LLM produces internally consistent output.

Limitations. Structural validity does not imply semantic correctness. A world model may pass validation while containing services that do not match the requested scenario or images that do not exist in any registry.

Infrastructure Validation Rate

Computation. After the world model is compiled into an OpenTofu JSON artefact, the framework executes `tofu validate`. This checks the generated HCL for syntax errors,

undefined resource references, and type mismatches. The metric records whether this command exits with code 0.

Justification. Infrastructure validation catches compilation errors before the more expensive plan and apply stages.

Limitations. `tofu validate` performs only static analysis. It does not check whether referenced Docker images exist, whether port mappings conflict at runtime, or whether environment variables contain valid values.

Plan Success Rate

Computation. The framework executes `tofu plan` on the validated artefact. Plan success is recorded when the command exits with code 0 and produces a non-empty execution plan. The rate is: $\frac{\text{successful plans}}{\text{total runs that reached this stage}}$.

Justification. A successful plan indicates that the infrastructure tool can resolve all resource dependencies and compute a valid execution graph.

Limitations. Plan success does not guarantee apply success. Resource conflicts (e.g., Docker network address exhaustion) may only manifest during apply.

Apply Success Rate

Computation. The framework executes `tofu apply` on the plan. Apply success is recorded when the command exits with code 0 and the deployment was a full apply (not a partial `-target` apply). The rate is: $\frac{\text{successful full applies}}{\text{total runs that reached this stage}}$.

Justification. Apply success is the first point at which containers are actually instantiated.

Limitations. Apply success means OpenTofu reported no errors. Individual containers may still exit immediately after creation. The apply success rate does not distinguish between scenarios where all containers remain running and scenarios where some exit within seconds.

Runtime Verification Rate

Computation. After apply, the framework queries the Docker daemon for the status of each expected container. A container is classified as running if its Docker status is `running` and it is not reported as *unhealthy*. In Docker, a container is marked unhealthy when its declared `HEALTHCHECK` probe (typically a command such as `pg_isready` or an HTTP

request) has failed its configured retry budget; a container without a healthcheck cannot be unhealthy by definition. Runtime verification succeeds when all expected containers are running and none are unhealthy. The rate is: $\frac{\text{runs with all containers healthy}}{\text{total deployed runs}}$.

Justification. Runtime verification captures whether the deployed containers actually function, not just whether OpenTofu reported success.

Limitations. A container reported as "running" by Docker may still be in a crash loop, returning errors on every request, or serving incorrect content. The metric does not assess application-level correctness; this is partially addressed by the QA pass rate below, which probes connectivity and HTTP responses.

QA Pass Rate

Computation. The QA module executes deterministic checks against each deployed container. The set of checks is generated automatically based on each container's declared ports and dependencies. Check types include: `container_running` (Docker status query), `tcp_connect` (TCP socket connection with 5-second timeout), `http_status` (HTTP request expecting a status code below 500), and `depends_on_running` (verifying that dependency targets are running). Each check produces a pass/fail result. The QA pass rate for a run is: $\frac{\text{checks passed}}{\text{checks executed}}$. The reported aggregate is the mean across runs.

Justification. QA checks provide a more granular assessment of deployment health than the binary runtime verification metric. A run with 49 of 52 checks passing (94%) conveys different information than a binary "runtime verification failed."

Limitations. QA checks are shallow. A TCP connection success does not verify that the service behind the port behaves correctly. An HTTP 200 response does not verify that the response body contains meaningful content. QA alone cannot tell whether the *right* services are present — that is captured by the scenario-fit metrics (Section 7.1.2).

Container Realization Ratio

Computation. At compilation time, the framework records the number of containers in the deploy projection as `planned_containers`. After deployment and runtime verification, the number of containers confirmed as running is recorded as `running_containers`. The ratio is: $\frac{\text{running}}{\text{planned}}$. Containers may be lost through two mechanisms: pre-deploy probe removal (the framework detects during probing that a container cannot start and removes it before deployment) and runtime failure (the container is deployed but exits or becomes unhealthy after apply).

Justification. The container realisation ratio captures the fraction of the intended deployment that actually materialised.

Limitations. The denominator (`planned_containers`) reflects the LLM's generation output, which may include non-essential companion systems. A realisation ratio below 1.0 does not necessarily mean that required services are missing. Whether the missing containers correspond to required services is captured by the running-phase scenario-fit metrics (Section 7.3.3).

Relationship Between Runtime Verification, QA Pass Rate, and Container Realization Ratio

These three metrics assess deployment health at different levels of granularity and answer distinct questions. Table 6 summarises their differences.

Metric	Question answered	Unit of observation	Scale	Aggregation
Runtime Verification Rate	Is the *entire* deployment healthy?	Whole deployment (all-or-nothing)	Binary per run	Fraction of runs with all containers healthy
QA Pass Rate	How many *individual* health checks pass?	Single check (TCP, HTTP, dependency)	Ratio per run	Mean ratio across runs
Container Realization Ratio	How much of the *planned infrastructure* materialized?	Single container	Ratio per run	Running / planned

Table 6. Comparison of the three deployment health metrics. Each metric operates at a different level of granularity: the deployment as a whole, individual health checks, or individual containers.

The three metrics are complementary, not redundant. A deployment can exhibit high QA pass rates (e.g., 98%) while having a low runtime verification rate (e.g., 10%) because runtime verification requires *all* containers to pass, whereas the QA pass rate tolerates partial failures. Similarly, a low container realisation ratio (e.g., 38% in the hard tier) can coexist with a high QA pass rate (98.7%) because the QA module only checks containers that survived deployment; containers removed during pre-deploy probing are excluded from the QA denominator.

In formal terms, the three metrics form a hierarchy of strictness:

1. **Container realisation ratio** measures how much of the planned infrastructure exists

at all (regardless of health).

2. **QA pass rate** measures how many health checks pass among the *surviving* containers.
3. **Runtime verification rate** requires *all* containers to exist *and* be healthy, that is, the conjunction of full realisation and full QA pass.

A scenario can therefore have a realisation ratio of 0.60 (40% of containers lost), a QA pass rate of 1.00 (all surviving containers are healthy), and a runtime verification of 0 (failed, because not all planned containers are present). This pattern is common in the hard tier and is discussed further in Section 7.3.1.

7.1.2 Scenario-Fit Metrics

Scenario-fit metrics evaluate whether the generated infrastructure matches the structural requirements of the input scenario. Each benchmark scenario is paired with a machine-readable reference.

Required-Service Recall

Computation. The benchmark reference defines a set of required services, each identified by matching criteria: `kinds_any` (system kind), `archetypes_any` (image patterns), `names_any` (system name patterns), and `roles_any` (functional role). A required service is considered matched if at least one system in the generated world model satisfies any of these criteria. The recall is: $\frac{\text{matched services}}{\text{total required services}}$.

Justification. Service recall measures whether the LLM produced the core services that the scenario describes.

Limitations. The matching is based on system metadata (kind, image name, system name), not on functional behaviour. A system may match the `postgres` archetype but be configured incorrectly.

Required-Zone Coverage

Computation. The benchmark reference defines required zones with matching criteria: `exposure_any` and `names_any`. Coverage is: $\frac{\text{matched zones}}{\text{total required zones}}$.

Justification. Zone coverage measures whether the LLM produced the correct network segmentation.

Limitations. A matched zone may contain the wrong services. Zone coverage does not assess whether services are assigned to the correct zones; that is captured by the placement violation metric.

Required-Dependency Satisfaction

Computation. The benchmark reference defines required dependencies as directed edges between service IDs. For each required dependency (s, t), the evaluator checks whether any system matched to service s has a `depends_on` entry pointing to a system matched to service t. Satisfaction is: $\frac{\text{satisfied deps}}{\text{total required deps}}$.

Justification. Dependency satisfaction measures whether the LLM correctly wired inter-service relationships.

Limitations. The metric only checks explicit `depends_on` declarations. Implicit dependencies (e.g., a service connecting to a database via environment variables without declaring `depends_on`) are not captured.

Forbidden-Placement Violations

Computation. The benchmark reference may define forbidden placements: pairs of (service ID, zone ID) indicating that a service must not be deployed in a specific zone. The metric is the count of such violations.

Justification. Placement violations capture security-relevant misconfigurations, such as placing an internal database in a public-facing zone.

Limitations. Forbidden placements are defined in 29 of 30 scenarios in this corpus. The metric captures only explicitly prohibited placements; zone correctness beyond the defined constraints is assessed through zone coverage.

Composite Scenario-Fit Score

Computation. The composite score combines the three coverage dimensions with a penalty for placement violations:

$$\text{score} = \bar{c} \cdot \max(0.5,\ 1 - 0.1 \cdot v) \tag{7.1}$$

where $\bar{c} = \frac{1}{3}(\text{svc\_recall} + \text{zone\_cov} + \text{dep\_sat})$ and v is the number of placement violations.

The formula consists of two components: a *coverage component* $\bar{c}$ that measures structural completeness, and a *violation penalty* that reduces the score based on security-relevant misplacements.

Design Justification.

Why a simple average for the coverage component. The three sub-dimensions (service recall, zone coverage, and dependency satisfaction) are all fractions between 0 and 1 that express "how much of what was required is present." Because they share the same scale and interpretation, they can be combined into a single coverage value using an arithmetic mean, as recommended by the OECD Handbook on Constructing Composite Indicators [74]. Equal weights (one third each) are used because no empirical evidence exists for ranking one dimension above another in this novel domain; equal weighting is the most transparent and reproducible choice in such cases [75].

Why the violation penalty is multiplicative rather than additive. Placement violations are not a fourth type of missing coverage — they indicate that a required service is present but placed where it should not be (e.g., a database deployed in a public zone). A multiplicative penalty keeps these two aspects conceptually separate: the coverage component answers "is it there?", and the penalty modifies the score if "is it in the right place?" fails. The same pattern is used by the Common Vulnerability Scoring System (CVSS) [76], where a base score is multiplied by modifiers rather than having them added as further dimensions.

Why the penalty stops at a floor of 0.5. Without a lower bound, ten or more violations would drive the penalty factor to zero, collapsing the composite score to zero regardless of how well the coverage dimensions were satisfied. This would discard useful information: a honeynet with 90% coverage and many violations would be indistinguishable from a honeynet with 10% coverage and the same violations. The 0.5 floor ensures that the coverage component always retains at least half of its signal, so scenarios with different coverage levels remain comparable even when violation counts are high.

Limitations. The equal weighting implies full compensability [75]: high service recall can compensate for low zone coverage. In domains where any single dimension is non-negotiable, a geometric mean $\bar{c}_{\text{geo}} = (\text{svc} \cdot \text{zone} \cdot \text{dep})^{1/3}$ would be more appropriate, as it forces the composite towards zero when any dimension approaches zero [77]. The 10%-per-violation penalty is a heuristic design parameter; its robustness is discussed in Section 8.3 of Chapter 8.

Overall Scenario Fit (Formal Benchmark Pass)

Computation. A scenario achieves full scenario fit if and only if: service recall $= 1.0$, zone coverage $= 1.0$, dependency satisfaction $= 1.0$, and placement violations $= 0$.

Justification. The overall scenario fit provides a binary answer to the question: "Did the framework produce exactly what was required?"

Limitations. The strict requirement means that a single missing service or a single placement violation causes failure. A scenario with a composite fit score of 0.90 does not achieve full scenario fit.

7.1.3 Repair and Robustness Metrics

Validation Repair Trigger Rate

Computation. If the world model fails structural validation, the framework enters a repair loop. The trigger rate is: $\frac{\text{runs where repair was triggered}}{\text{total runs}}$. This measures how often repair was needed, not how often it succeeded.

Justification. This metric captures how often the LLM's initial output is structurally defective.

Limitations. The metric counts repair *attempts*, not successes.

Deploy Retry Rate

Computation. After a failed `tofu apply`, the framework may attempt recovery through import of existing Docker resources or an apply-repair loop. The deploy retry rate is: $\frac{\text{runs with retry count}>0}{\text{total deployed runs}}$.

Justification. Deploy retries indicate infrastructure-level failures that the framework attempted to recover from.

Limitations. Deploy retries are often caused by transient Docker resource conflicts rather than by configuration errors in the generated output.

Container Drop Rate

Computation. The total container loss for a run is: planned $-$ running. This total consists of two components: containers removed during pre-deploy probing (recorded as

`dropped_containers`) and containers that failed at runtime after deployment. The drop rate is: $\frac{\text{planned}-\text{running}}{\text{planned}}$.

Justification. The drop rate measures the fraction of the planned deployment that the framework was unable to realise.

Limitations. Dropped containers may be non-essential companions rather than core services. The relationship between dropped containers and scenario fit requires cross-referencing with the scenario-fit metrics.

7.1.4 Complexity Sensitivity

All metrics defined above are reported separately for each difficulty tier.

Computation. Scenarios are assigned to tiers based on the benchmark reference: easy (e01–e10), medium (m01–m10), hard (h01–h10).

Justification. Aggregate metrics across all scenarios can mask poor performance on complex scenarios. Per-tier reporting reveals whether quality degrades gradually or at a specific complexity threshold.

Limitations. Each tier contains exactly 10 scenarios. With a single run per scenario, per-tier statistics are based on small samples, and individual outliers can substantially affect tier-level aggregates. Min–max ranges are reported alongside means to indicate spread.

Having defined the evaluation metrics, the next section describes the experimental setup under which they were collected.

7.2 Experimental Setup

7.2.1 Benchmark Corpus

The evaluation uses the full benchmark corpus of 30 scenarios across three difficulty tiers (10 easy, 10 medium, 10 hard). Each scenario consists of a natural-language prompt and a machine-readable reference. Table 7 summarises the structural complexity.

The number of forbidden placements increases with tier complexity: easy scenarios define

Tier	N	Mean Svc	Mean Zones	Mean Deps	Mean Forbid.
Easy	10	3.3	2.1	2.0	1.5
Medium	10	6.0	3.1	3.7	2.5
Hard	10	9.3	5.2	5.6	4.7
Total	30	6.2	3.5	3.8	2.9

Table 7. Benchmark corpus complexity by tier. Svc = required services, Deps = required dependencies, Forbid. = mean forbidden placements defined in the reference.

1.5 placements on average (15 pairs total across 9 scenarios; e10 is the only scenario without any forbidden-placement constraints), medium scenarios define 2.5 on average (25 pairs across all 10), and hard scenarios define 4.7 on average (47 pairs across all 10). This pattern reflects the increasing number of security-relevant zone boundaries as topological complexity grows: easy-tier scenarios contain 2–3 zones with at most one or two sensitive components, whereas hard-tier scenarios contain 4–6 zones with multiple trust boundaries (e.g., a banking core separating DMZ, core, data, identity, compliance, and ops zones), each contributing additional service-to-zone combinations that must be prohibited.

7.2.2 System Configuration

The framework was configured with multi-phase extraction (three LLM calls: architecture skeleton, batched configuration, enrichment). The LLM was provided with tool access for Docker image validation and OCI configuration retrieval. Post-extraction fixups (OCI command correction, dependency inference from environment variables, placement correction for backend services) were enabled. Deployment used the Docker provider via OpenTofu. Table 8 lists the environment specifications.

Component	Specification
Operating system	Windows 11 (WSL2)
CPU	AMD Ryzen 7 5800X (8 cores)
RAM	32 GB DDR4
Container runtime	Docker Desktop 4.37
Infrastructure tool	OpenTofu 1.11.5
LLM provider	OpenAI API
LLM model	GPT-4o (gpt-4o-2024-11-20)
LLM temperature	0.3
Extraction mode	Multi-phase (3-call)
Max image repair attempts	3
Max apply repair attempts	3

Table 8. Environment and configuration for all benchmark runs.

7.2.3 Execution Protocol

Each scenario was deployed into an isolated Docker network namespace and destroyed after metric collection. Scenarios were executed sequentially on a single workstation; no parallelism was used between scenarios. Each scenario was run exactly once.

The single-run design was chosen due to the deployment cost of each scenario (mean duration: 616 seconds, including container creation and teardown). A full 30-scenario corpus run requires approximately 5 hours. Multi-run evaluation with 5 repetitions would require approximately 25 hours per configuration, making exhaustive repetition impractical within the available time budget. As a consequence, the reported results do not include variance estimates across runs, and individual outlier scenarios may disproportionately affect tier-level aggregates. This limitation is acknowledged; the evaluation focuses on structural patterns across scenarios rather than on statistical significance of individual measurements.

7.3 Results

This section reports the measured data. All values are reported as observed; no corrections or adjustments were applied.

7.3.1 Deployability

Table 9 reports the pipeline stage success counts by tier.

Pipeline Stage	**Easy**	**Medium**	**Hard**
World model valid	10/10	10/10	9/10
Infrastructure valid	10/10	10/10	9/10
Plan success	10/10	10/10	9/10
Apply success	10/10	10/10	9/10
Runtime verification	3/10	1/10	5/10

Table 9. Pipeline stage success counts by difficulty tier. One hard-tier scenario (h10) failed at world model validation; the remaining stages were not reached for that scenario.

The five stages reported here correspond to the pipeline stages described in Chapter 5: *world model valid* follows the extraction and structural validation steps (Section 5.3); *infrastructure valid* corresponds to `tofu validate` on the rendered artifact; *plan success* and *apply success* correspond to the OpenTofu `plan` and `apply` commands executed by the deployer; and *runtime verification* corresponds to the post-deployment

Docker status check (Section 5.6.4). The compilation, image resolution, and rendering stages are not listed separately because they are deterministic and succeeded for every scenario that reached them.

All 10 easy-tier and all 10 medium-tier scenarios completed every pipeline stage through apply. In the hard tier, 9 of 10 scenarios completed all stages; scenario h10 failed at world model validation with an `llm_schema_error` (a generated system referenced an undefined dependency).

Table 9 reveals a sharp discontinuity between the deterministic pipeline stages and runtime verification. The first four stages (world model validation, infrastructure validation, plan, and apply) achieve near-perfect success rates across all tiers (100% for easy and medium, 90% for hard due to a single extraction failure). This indicates that the framework's validation and repair mechanisms reliably produce deployable infrastructure artefacts regardless of scenario complexity.

The discontinuity occurs at runtime verification, which drops to 30% (easy), 10% (medium), and 50% (hard). This gap between apply success and runtime verification reflects the distinction between OpenTofu reporting a successful apply (all resource creation API calls succeeded) and all containers remaining healthy after creation. Containers may exit seconds after OpenTofu considers them created.

The counterintuitive result that hard-tier runtime verification (5/10) exceeds medium-tier verification (1/10) is not evidence that hard scenarios are easier to deploy. Rather, it is a consequence of pre-deploy probing (Section 5.6.3): the hard tier loses 110 containers to pre-deploy probing (Table 10), leaving fewer containers that must all be healthy for runtime verification to pass. With fewer containers to check, the probability that all pass increases. This effect is an instance of *survivorship bias* in the verification metric: by removing likely failures before deployment, the framework increases the fraction of scenarios where the remaining containers are all healthy, while the overall deployment is substantially degraded. Whether this is the right behaviour or should be discussed as a methodological question is addressed in Chapter 8.

Table 10 reports aggregate container counts.

Table 10 reveals two patterns that require explanation.

Pattern 1: Realization ratio decreases with complexity. The container realisation ratio drops from 91% (easy) to 85% (medium) to 38% (hard). This degradation is driven

Metric	Easy	Medium	Hard	All
Containers planned	160	228	248	636
Containers running	146	193	94	433
Total lost	14	35	154	203
of which: pre-deploy drops	12	34	110	156
of which: runtime failures	2	1	44	47
Realization ratio	91%	85%	38%	68%
Mean QA pass rate	97.5%	99.6%	98.7%	98.6%
Mean QA checks per run	33.6	36.7	17.6	29.3

Table 10. Aggregate container metrics. "Total lost" is planned − running. "Pre-deploy drops" are containers removed before deployment because the pre-deploy probe detected they could not start. "Runtime failures" are containers that were deployed but exited or became unhealthy after apply. The hard tier has fewer QA checks per run because fewer containers survived to be checked.

primarily by pre-deploy drops, not runtime failures: 110 of 154 containers lost in the hard tier (71%) were removed before deployment because they could not start during probing. Hard-tier scenarios require more specialised services (e.g., satellite ground stations, SCADA controllers, ML training nodes) that depend on configuration files, specific hardware, or complex multi-service coordination that cannot be satisfied by the framework's default environment variable injection. The pre-deploy probe is a conservative design choice: it trades realisation ratio for deployment stability by removing containers that would otherwise cause OpenTofu apply failures.

Pattern 2: QA pass rate remains high despite low realisation. The mean QA pass rate is 97.5% (easy), 99.6% (medium), and 98.7% (hard), showing no meaningful degradation across tiers even as the realisation ratio drops by 53 percentage points. This apparent paradox is explained by the QA denominator: QA checks are only generated for containers that survived deployment. Containers removed during pre-deploy probing are excluded from the QA check set entirely. Consequently, the QA pass rate answers the question "of the containers that are running, how many are healthy?" rather than "how much of the intended deployment is healthy?" The latter question is answered by the container realisation ratio. The distinction is critical for interpretation: a QA pass rate of 98.7% in the hard tier does not mean the deployment is 98.7% successful; it means that the 38% of containers that survived are individually healthy. This relationship between the metrics is formalized in Section 7.1.1.

Figure 5 shows the per-scenario container counts across all 30 scenarios. The three tiers are arranged left-to-right: easy (e01–e10), medium (m01–m10), hard (h01–h10). For the easy tier, the gap between planned and running bars is small, with e06 (object storage)

showing the largest loss (21 planned, 16 running). For the medium tier, the gap is generally small except for m10 (game backend), which drops from 28 planned to 13 running. For the hard tier, the gap is consistently large: h03 (ML platform) drops from 36 to 15, h08 (smart city) from 31 to 13, and h10 (pharma research) has 35 planned and 0 running due to the extraction failure.

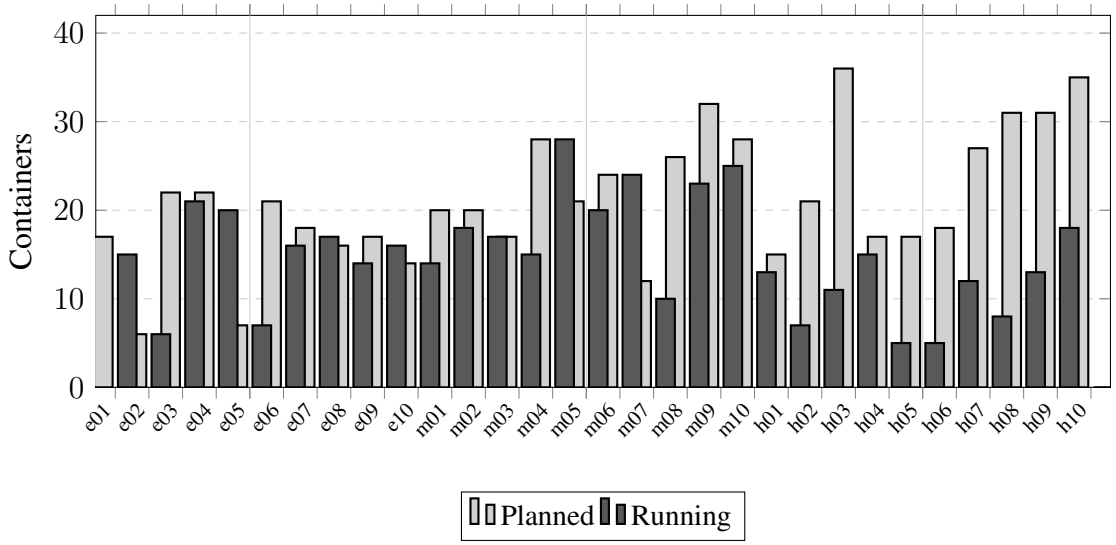

Figure 5. Planned vs. running containers for all 30 scenarios. Scenarios are ordered by tier: e01–e10 (easy), m01–m10 (medium), h01–h10 (hard). The gap between the light bar (planned) and the dark bar (running) represents containers lost through pre-deploy probing or runtime failure.

7.3.2 Scenario Fit

Table 11 reports the mean scenario-fit metrics by tier, with min–max ranges to indicate spread. All values in this section refer to the *planned* evaluation (against the world model before deployment).

Metric	Easy	Medium	Hard	All
Service recall	0.77 [0.33–1.00]	0.98 [0.83–1.00]	0.82 [0.00–1.00]	0.86
Zone coverage	0.85 [0.50–1.00]	0.93 [0.67–1.00]	0.90 [0.00–1.00]	0.89
Dep. satisfaction	0.65 [0.00–1.00]	0.98 [0.75–1.00]	0.55 [0.00–0.83]	0.73
Mean violations	0.3 [0–2]	1.7 [0–12]	2.7 [0–10]	1.6
Composite score	0.73 [0.44–1.00]	0.87 [0.50–1.00]	0.60 [0.00–0.94]	0.73
Overall scenario fit	2/10	5/10	0/10	7/30

Table 11. Mean planned scenario-fit metrics by tier. Values in square brackets indicate the range [min–max] across the 10 scenarios in each tier. The composite score is computed per Equation 7.1.

Figure 6 shows the composite fit score for each of the 30 scenarios. In the easy tier, scores range from 0.44 (e07 DNS sinkhole, e08 VPN gateway) to 1.00 (e03 web+cache+DB,

e10 CI runner). In the medium tier, all scores are at or above 0.50, with five scenarios reaching 1.00 (m03, m05, m07, m08, m10). In the hard tier, no scenario reaches 1.00; the highest is h03 (ML platform) at 0.94, and h10 (pharma research) scores 0.00 due to the extraction failure.

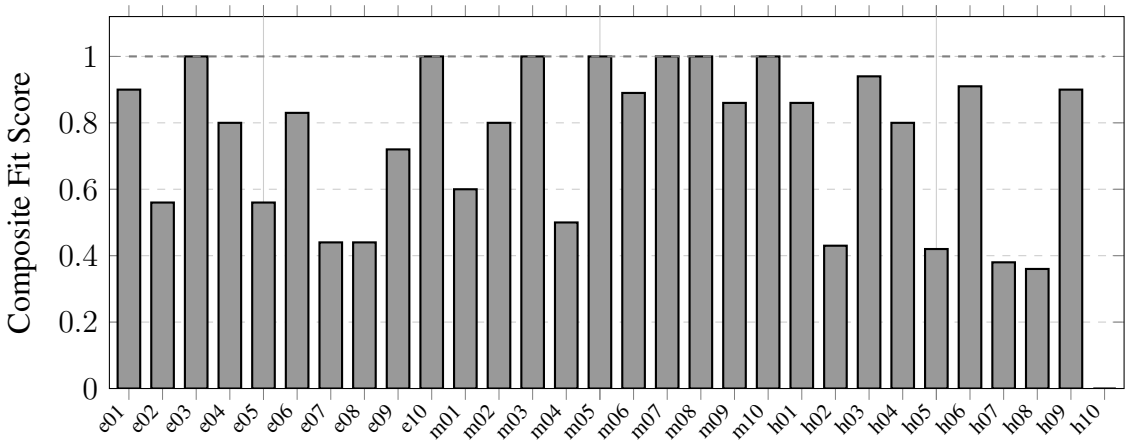

Figure 6. Composite scenario-fit score (planned) for all 30 scenarios. Scores are computed per Equation 7.1. A score of 1.0 corresponds to full overall scenario fit. Scenarios are ordered by tier: e01–e10 (easy), m01–m10 (medium), h01–h10 (hard). The dashed line marks the maximum score of 1.0.

Table 12 reports the per-scenario breakdown.

ID	Scenario	Svc	Zone	Dep	Viol.	Pass	Score
Easy tier							
e01	Web + Database	1.00	1.00	1.00	1	No	0.90
e02	Identity (LDAP)	0.67	0.50	0.50	0	No	0.56
e03	Web + Cache + DB	1.00	1.00	1.00	0	Yes	1.00
e04	Monitoring Stack	1.00	1.00	1.00	2	No	0.80
e05	Message Queue	0.67	0.50	0.50	0	No	0.56
e06	Object Storage	1.00	0.50	1.00	0	No	0.83
e07	DNS Sinkhole	0.33	1.00	0.00	0	No	0.44
e08	VPN Gateway	0.33	1.00	0.00	0	No	0.44
e09	File Share	0.67	1.00	0.50	0	No	0.72
e10	CI Runner	1.00	1.00	1.00	0	Yes	1.00
Medium tier							
m01	E-Commerce	1.00	1.00	1.00	4	No	0.60
m02	Corporate Email	1.00	0.67	1.00	1	No	0.80
m03	Healthcare	1.00	1.00	1.00	0	Yes	1.00
m04	DevOps Platform	1.00	1.00	1.00	12	No	0.50
m05	Supply Chain	1.00	1.00	1.00	0	Yes	1.00

ID	Scenario	Svc	Zone	Dep	Viol.	Pass	Score
m06	Streaming Platform	1.00	0.67	1.00	0	No	0.89
m07	SCADA Honeypot	1.00	1.00	1.00	0	Yes	1.00
m08	Crypto Exchange	1.00	1.00	1.00	0	Yes	1.00
m09	Ad Network	0.83	1.00	0.75	0	No	0.86
m10	Game Backend	1.00	1.00	1.00	0	Yes	1.00
Hard tier							
h01	Banking Core	0.91	1.00	0.67	0	No	0.86
h02	Telecom Mediation	1.00	1.00	0.60	10	No	0.43
h03	ML Platform	1.00	1.00	0.83	0	No	0.94
h04	Zero-Trust Enterprise	0.89	1.00	0.50	0	No	0.80
h05	IoT Platform	1.00	1.00	0.50	6	No	0.42
h06	Government Portal	0.91	1.00	0.83	0	No	0.91
h07	Satellite Ground	0.88	1.00	0.40	5	No	0.38
h08	Smart City	0.75	1.00	0.40	6	No	0.36
h09	Defense C2	0.89	1.00	0.80	0	No	0.90
h10	Pharma Research	0.00	0.00	0.00	0	No	0.00

Table 12. Per-scenario planned fit results. Svc = service recall, Zone = zone coverage, Dep = dependency satisfaction, Viol. = placement violations, Pass = overall scenario fit, Score = planned composite per Equation 7.1. Scenario h10 failed at validation; all metrics are zero.

7.3.3 Planned vs. Running Scenario Fit

The preceding section evaluates scenario fit against the generated world model (planned). This section compares those results with the fit evaluated against only the containers confirmed running after deployment.

Table 13 shows the mean planned and running coverage by tier. Service coverage decreases from planned to running in all tiers, with the largest drop in the hard tier (0.82 to 0.51). Zone coverage is unchanged because zone definitions are structural properties of the deployment and are not affected by individual container failures. Dependency coverage shows the largest degradation: in the hard tier, it drops from 0.55 (planned) to 0.07 (running), indicating that container failures break dependency chains even when the world model declared the correct relationships.

	Service		Zone		Dependency	
Tier	Plan	Run	Plan	Run	Plan	Run
Easy	0.77	0.73	0.85	0.85	0.65	0.60
Medium	0.98	0.90	0.93	0.93	0.98	0.63
Hard	0.82	0.51	0.90	0.90	0.55	0.07

Table 13. Mean planned vs. running coverage by tier.

7.3.4 Repair and Robustness

Table 14 reports the repair metrics by tier.

Metric	**Easy**	**Medium**	**Hard**	**All**
Validation repair triggered	0/10	0/10	1/10	1/30
Runs with deploy retries	7/10	9/10	7/10	23/30
Runtime repair incidents	7/10	9/10	4/10	20/30
Pre-deploy containers dropped	12	34	110	156
Runtime containers failed	2	1	44	47
Total container loss rate	8.8%	15.4%	62.1%	31.9%
Container realisation ratio	91.2%	84.6%	37.9%	68.1%
Image validation pass rate	100%	100%	100%	100%

Table 14. Repair and robustness metrics by tier. The single validation repair (h10) did not succeed. Image validation passed for all images across all 30 scenarios.

Validation repair was triggered in one scenario (h10) and did not succeed. All other 29 world models passed structural validation on the first attempt. Image validation achieved 100% across all tiers.

Deploy retries occurred in 23 of 30 runs (77%). Runtime repair incidents occurred in 20 of 30 runs (67%). The total container loss rate increases from 8.8% (easy) to 62.1% (hard). In the hard tier, 110 containers were dropped during pre-deploy probing and an additional 44 failed at runtime, for a combined loss of 154 out of 248 planned containers.

7.3.5 Complexity Sensitivity

Figure 7 provides a side-by-side comparison of seven key metrics across the three tiers. The deploy rate is at or near 1.0 for all tiers. Service recall, zone coverage, and dependency satisfaction are highest in the medium tier and lowest in the hard tier. Container realisation drops from 0.91 (easy) to 0.38 (hard). The composite fit score follows a similar pattern: 0.73 (easy), 0.87 (medium), 0.60 (hard). The overall scenario fit rate is 0.20 (easy), 0.50 (medium), 0.00 (hard).

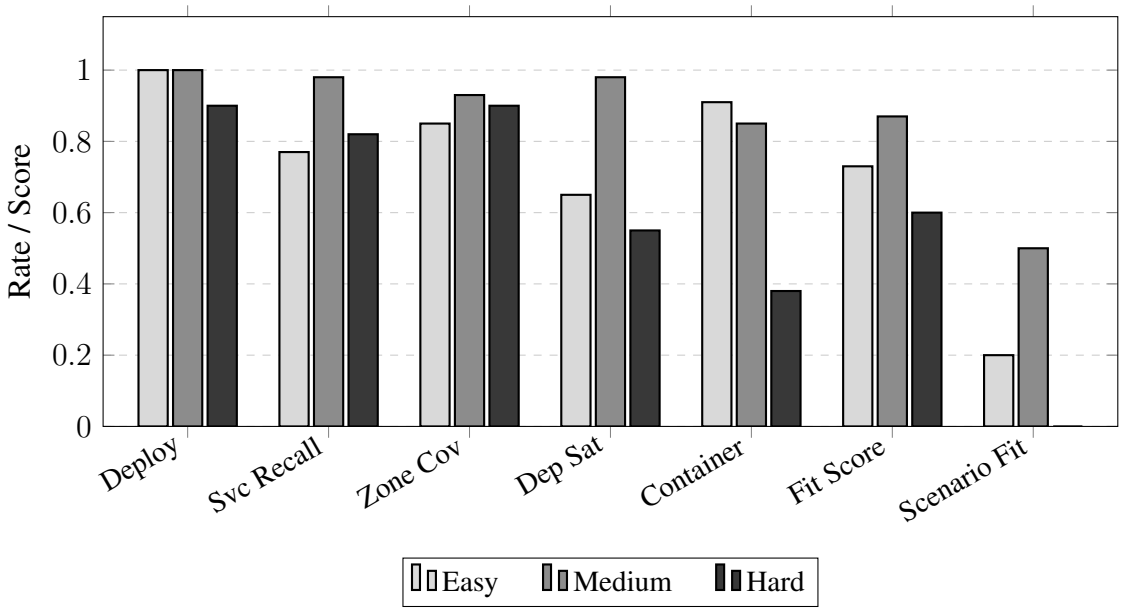

Figure 7. Key metrics by difficulty tier. Deploy = apply success rate, Svc Recall = mean service recall, Zone Cov = mean zone coverage, Dep Sat = mean dependency satisfaction, Container = realisation ratio, Fit Score = mean composite score (Eq. 7.1), Scenario Fit = overall scenario fit rate (count/10). All values are in the range $[0, 1]$.

7.3.6 Timing

Table 15 reports the mean pipeline stage durations by tier. The "Other" row captures time spent in pre-deploy probing, Docker image pulls during probing, inter-stage overhead, and cleanup operations that are not attributed to a named stage.

Stage	Easy	Medium	Hard	All
Extraction (LLM)	38s	54s	60s	51s
Deploy (OpenTofu)	73s	65s	46s	61s
Runtime verification	184s	217s	155s	185s
Other (probing, overhead)	232s	367s	355s	318s
Total pipeline	527s	703s	616s	616s

Table 15. Mean per-stage timing by tier (seconds, rounded). Image resolution and compilation are each consistently below 1 second and are included in "Other." The "Other" category also includes pre-deploy container probing and Docker image pull times during probing.

Figure 8 visualises the time breakdown as a stacked bar chart. Across all tiers, the "Other" category (which includes pre-deploy probing) consumes the largest share of total pipeline time. Extraction time increases from 38s (easy) to 60s (hard) as the LLM generates more systems. Deploy time is highest for the easy tier (73s) due to one outlier scenario (e03, 364s) that required extensive Docker image pulls for 22 containers.

Hard-tier scenarios complete faster than medium-tier scenarios (616s vs. 703s), despite generating more systems. This is a consequence of pre-deploy probing: the hard tier loses 110 containers during probing (Table 10), so fewer containers reach the runtime verification stage. Runtime verification is sequential per container, and with roughly half as many surviving containers on average in the hard tier, this stage takes 155s rather than the 217s observed in medium.

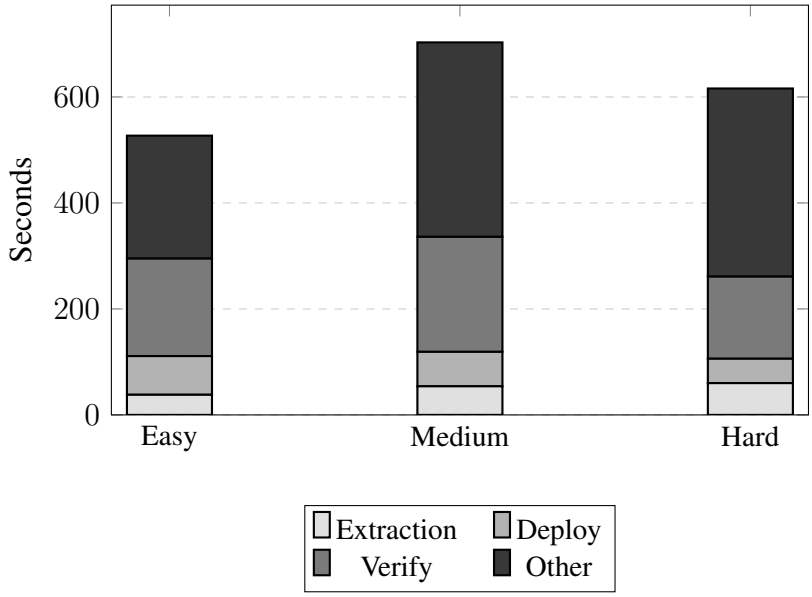

Figure 8. Mean pipeline time breakdown by tier (stacked bar chart). Each bar represents the total mean pipeline duration for a tier, decomposed into four components. "Other" includes pre-deploy probing, image resolution, compilation, rendering, and inter-stage overhead.

7.4 Summary of Measured Results

Table 16 provides the aggregate results across all 30 scenarios.

The per-tier overall scenario fit distribution is: easy 2/10, medium 5/10, hard 0/10. The per-tier mean composite fit scores are: easy 0.73, medium 0.87, hard 0.60. The per-tier container realisation ratios are: easy 91%, medium 85%, hard 38%.

Three patterns dominate the aggregate numbers. First, the deployability metrics (world model validity, apply success, image validation) all exceed 97%, while the overall scenario fit rate is 23%; the gap between these two groups is the largest in the table. Second, among the scenario-fit dimensions, zone coverage (0.89) is the highest and dependency satisfaction (0.73) is the lowest, with service recall (0.86) between them. Third, the per-tier breakdown of overall scenario fit (2/10, 5/10, 0/10) is non-monotonic in difficulty: the medium tier passes more scenarios than the easy tier. The interpretation of these patterns

Metric	Value
Scenarios evaluated	30
World model validity rate	97% (29/30)
Apply success rate	97% (29/30)
Overall scenario fit rate	23% (7/30)
Mean composite scenario-fit score	0.73
Mean required-service recall	0.86
Mean required-zone coverage	0.89
Mean required-dependency satisfaction	0.73
Total placement violations	47
Mean QA pass rate	98.6%
Container realisation ratio	68% (433/636)
Total container loss rate	32% (203/636)
Validation repairs triggered	1/30
Image validation pass rate	100%
Mean pipeline duration	616s

Table 16. Aggregate results across all 30 benchmark scenarios.

is deferred to Chapter 8.

8. Discussion

This chapter interprets the evaluation results presented in Chapter 7. Section 8.1 answers each research question. Section 8.2 positions the findings with respect to prior work. Section 8.3 discusses limitations. Section 8.4 derives implications for research and practice.

8.1 Answering the Research Questions

8.1.1 RQ1: Deployability

Can an LLM-based pipeline reliably generate deployable container-based honeynets from natural-language prompts?

The pipeline completed the full deployment path (validate, plan, apply) in 29 of 30 scenarios (97%). However, the container realisation ratio (the fraction of originally planned containers that remained running after deployment) was 68% overall (433 of 636 planned containers) and dropped to 38% in the hard tier. The 97% figure reflects pipeline completion, not deployment completeness. The single pipeline failure (h10) was caused by an LLM schema error in which a generated system referenced an undefined dependency.

Runtime verification, which requires all expected containers to be running and healthy, succeeded in 9 of 29 deployed scenarios (31%). The remaining 20 scenarios were classified as runtime-degraded: the deployment completed, but individual containers exited or failed health checks. The mean QA pass rate across all deployed scenarios was 98.6%, indicating that the large majority of individual health checks passed even in degraded deployments.

The data show a separation between two aspects of pipeline viability. The pipeline reliably translates a natural-language prompt into a deployable infrastructure artefact: the path from prompt to running containers succeeds in 97% of cases. However, the pipeline does not reliably produce deployments in which *all* planned containers remain healthy. The container realisation ratio was 91% for the easy tier, 85% for the medium tier, and 38% for the hard tier, with an overall ratio of 68% (433 of 636 planned containers).

The 32% container loss is attributable to two mechanisms: pre-deploy probing removed 156 containers that could not start, and 47 additional containers failed at runtime after deployment. Inspecting the repair-history logs across the 30 scenarios, four recurring root

causes account for the large majority of these losses:

1. **Config-dependent images without mounted configuration.** Images such as HAProxy, Envoy, Squid, or Varnish require a configuration file at a specific path (e.g., `/etc/haproxy/haproxy.cfg`) to start. When the LLM generated such a service without providing a volume mount, the container exited with a file-not-found error and was removed during probing. This accounts for a substantial share of hard-tier drops, where such gateway and proxy services appear more frequently.
2. **Base-runtime images without an application command.** Images such as `python:3.12` or `node:20` start an interactive interpreter that exits when run without a TTY and without a command. When the LLM selected a base runtime image for what was intended to be an application service, the container exited immediately.
3. **Missing required environment variables.** Some images (e.g., `postgres`, `mysql`) refuse to start when required variables are unset. Post-extraction OCI fixup injects placeholder values for common cases, but image-specific requirements outside the fixup table cause runtime exit.
4. **Command/argument mismatch.** Some images require a subcommand to run as a server (e.g., `minio server /data`). When the LLM provided only the image name without the subcommand, the container exited after printing usage text. Deterministic repair catches a subset of these cases by pattern-matching on "USAGE:" in logs, but not all variants.

The pre-deploy probe is a design choice that trades container loss for deployment stability; without it, the apply success rate would be lower because OpenTofu apply fails when containers exit immediately.

Role of repair mechanisms. Validation-level repair was triggered in 1 of 30 runs (h10) and did not succeed. The remaining 29 world models passed structural validation on the first attempt without requiring repair. Image validation achieved 100% across all scenarios after removing systems whose images could not be resolved through the five-stage fallback chain, indicating that the catalogue-guided image selection combined with OCI introspection reliably produces valid image references for the surviving systems.

Deploy retries occurred in 23 of 30 runs (77%), and runtime repair incidents occurred in 20 of 30 runs (67%). However, the primary action of the runtime repair mechanism was container removal rather than configuration correction: 156 containers were removed during pre-deploy probing, while 47 additional containers failed at runtime.

The repair architecture is effective at two levels:

- **Structural repair** (OCI fixup, healthcheck rewriting, dependency inference from environment variables, placement correction) operates before deployment and produces deterministic corrections. The 100% image validation rate and the 97% apply success rate are partially attributable to these fixups.
- **Runtime repair** (pre-deploy probing, apply-repair loop, container removal) first attempts deterministic and LLM-assisted repair for failing containers. Containers that cannot be repaired are removed to preserve the health of the surviving deployment.

The absence of ablation data means that the contribution of individual repair mechanisms cannot be isolated from the evaluation data. Palavalli and Santolucito [28] report that feedback loop effectiveness for LLM IaC generation decreases with diminishing returns per iteration. The framework's design of limiting repair to one iteration at the validation level is consistent with this finding.

8.1.2 RQ2: Fidelity

While deployability addresses whether the framework *can* deploy, fidelity addresses whether what it deploys *matches* the user's intent.

How well do the generated honeynets match the structural intent of the input scenario?

The formal overall scenario fit rate was 7 of 30 scenarios (23%). The mean planned composite scenario-fit score was 0.73 (evaluated against the generated WorldModel before deployment; running-phase scores are lower due to container loss, as discussed in Section 7.3.3).

The three coverage dimensions show different levels of achievement. Zone coverage was the strongest dimension (mean 0.89), indicating that the LLM reliably creates the required network segmentation. Service recall was 0.86 overall, with the medium tier achieving 0.98 and the easy tier achieving only 0.77. Dependency satisfaction was the weakest dimension (mean 0.73 overall, 0.55 in the hard tier), indicating that the LLM generates required services reliably but struggles with inter-service dependency declaration.

Placement violations totaled 47 across all 30 scenarios, with concentrations in m04 (DevOps platform, 12 violations) and h02 (telecom mediation, 10 violations). In both cases, the LLM generated the required services but distributed them across zones differently than the benchmark reference specified.

The data reveal three categories of scenario-fit failure:

1. **Domain-specific service gaps.** Scenarios e07 (DNS sinkhole, service recall 0.33) and e08 (VPN gateway, service recall 0.33) require niche services (DNS resolver, VPN concentrator) that the LLM did not generate. Instead, the LLM produced generic web proxies and admin panels. Hashemi Doulabi et al. [78] identify "contextual reasoning failure" as a category of LLM IaC errors where the model generates syntactically valid but semantically misaligned configurations. Bhusal et al. [79] show that LLMs exhibit notable weaknesses on specialised cybersecurity tasks, particularly those requiring contextual reasoning over emerging threats, and Alam et al. [80] find that existing general-purpose benchmarks fail to capture the practical aspects of CTI-specific reasoning, both consistent with the thesis finding that LLM domain familiarity with niche infrastructure types is a primary quality predictor.
2. **Dependency wiring failures.** In the hard tier, mean dependency satisfaction was 0.55. The LLM generated most required services but failed to declare the correct `depends_on` relationships between them. For example, h07 (satellite ground station) achieved service recall of 0.88 but dependency satisfaction of only 0.40. This pattern is consistent with the observation that LLMs process individual entities more reliably than they reason about relationships between entities in multi-constraint architectures.
3. **Zone assignment disagreements.** Placement violations arise when the LLM assigns services to zones that do not match the benchmark reference. In m04 (DevOps platform), the LLM distributed services across `internal_zone` and `data_zone` rather than concentrating them in the expected `build_zone`. In some cases (e.g., h02), the assignment chosen by the LLM represents a defensible alternative topology rather than an error.

8.1.3 RQ3: Scalability

RQ1 and RQ2 report aggregate results across all 30 scenarios. RQ3 examines how these results vary with scenario complexity, revealing whether the framework's quality degrades predictably as the required architecture grows.

How does generation quality change as scenario complexity grows?

The data show a non-monotonic relationship between difficulty tier and quality. The medium tier (5–7 required services, 3–4 zones, 3–4 dependencies) outperformed the easy tier (3–4 required services, 2–3 zones, 2 dependencies) on service recall (0.98 vs. 0.77), dependency satisfaction (0.98 vs. 0.65), and composite fit score (0.87 vs. 0.73). The overall

scenario fit rate was 50% for the medium tier and 20% for the easy tier.

The hard tier (8–11 required services, 4–6 zones, 5–6 dependencies) showed the expected degradation: composite fit score 0.60, benchmark pass rate 0/10, and container realisation ratio 38%.

The non-monotonic easy–medium pattern can be attributed to the tier classification method. Difficulty tiers were assigned based on the number of required services, zones, and dependencies in the benchmark reference. This measures *topological complexity*, that is, the size of the required architecture. However, the LLM's generation difficulty depends on a different factor: the *familiarity* of the requested service types in the model's training data.

The five passing medium scenarios (m03 healthcare, m05 supply chain, m07 SCADA, m08 crypto exchange, m10 game backend) describe technology stacks built from well-known components (PostgreSQL, Redis, Kafka, NATS, MinIO). The failing easy scenarios (e07 DNS sinkhole, e08 VPN gateway) require niche services (DNS resolvers configured for sinkhole behaviour, VPN concentrators) that map poorly to standard Docker images. Abhyankar et al. [40] report a similar pattern: LLMs achieve higher accuracy on well-known infrastructure patterns than on niche configurations, independent of the number of resources involved.

The scalability analysis reveals that the framework's quality is bounded by two independent factors: (1) the LLM's domain knowledge for the requested service types, and (2) the number of inter-service dependencies and zone constraints that must be satisfied simultaneously. The hard tier is affected by both: it requires more dependencies (mean 5.6) and often involves specialised domains (satellite, defence, telecom).

8.2 Positioning with Respect to Prior Work

The framework occupies a position between two categories of existing systems. Template-based honeynet generators such as HoneyFactory [8] and HoneyChart [9] produce reliable deployments from predefined configurations but cannot generate novel architectures from natural language. LLM-based IaC generation systems such as those surveyed by Srivatsa et al. [12] can interpret natural language but produce configurations that are frequently syntactically valid yet semantically incorrect.

The framework's 97% deploy success rate and 98.6% QA pass rate exceed the raw LLM pass rates reported in the literature. Kon et al. [11] report 19.36% pass@1 for GPT-4 on the IaC-Eval Terraform benchmark. Hashemi Doulabi et al. [78] report that knowledge

injection raises IaC generation success from 27.1% to 62.6%. The framework's higher deploy rate is attributable to the multi-stage validation and repair pipeline that corrects common LLM errors before deployment.

However, the framework's 23% formal overall scenario fit rate reveals that deployability and scenario fidelity are distinct dimensions. Abhyankar et al. [40] observe the same gap in their Multi-IaC-Eval benchmark: LLMs achieve over 95% syntactic validity but substantially lower semantic alignment. The 74-percentage-point gap between deploy success (97%) and benchmark pass (23%) in this evaluation is consistent with their finding.

Ahmed et al. [38] describe a multi-agent system for cybersecurity lab generation that uses specialised agents for scenario planning, environment generation, and validation. Their system achieves high accuracy on 14 attack scenarios. The framework presented here addresses a broader scope (30 scenarios across three difficulty tiers) but achieves lower fidelity on complex scenarios, which may reflect the difference between targeted attack-scenario generation and open-ended infrastructure generation from unconstrained prompts.

The multi-phase extraction design (architecture, configuration, enrichment) follows the decomposition pattern surveyed by He et al. [55] and empirically validated in the IaC domain by Hussain Khan et al. [56], whose MACOG framework raises GPT-5 from 54.90 to 74.02 on IaC-Eval over a RAG baseline (a relative gain of approximately 35%) through specialised agent decomposition with deployment-feedback loops. Without ablation data comparing multi-phase to single-call extraction, the contribution of the decomposition in this framework cannot be quantified from the available results.

8.3 Limitations

8.3.1 Single LLM Model

All evaluation results are based on a single LLM (GPT-4o). The generalizability of the results to other models (Claude, Gemini, Llama, Mistral) is unknown. The domain-specific service gaps observed in scenarios e07 and e08 may be specific to GPT-4o's training data distribution. A model with stronger coverage of networking and security domains might produce different results on these scenarios.

8.3.2 Single Run per Scenario

Each scenario was executed once. LLM outputs are inherently non-deterministic; Atil et al. [81] report that even with deterministic settings (temperature 0), LLM accuracy can vary by up to 15% across runs, with best-to-worst performance gaps of up to 70% on some tasks. Ouyang et al. [82] further demonstrate that up to 75.76% of code generation tasks produce no identical outputs across repeated requests, and that temperature 0 does not guarantee determinism. Song et al. [83] additionally argue that LLM evaluations reporting only single outputs systematically ignore non-determinism. The reported metrics are therefore point estimates without confidence intervals. A scenario that narrowly failed (e.g., e01 with a fit score of 0.90 due to one placement violation) might pass on a subsequent run, and a passing scenario might fail. The 23% overall scenario fit rate has an unknown confidence interval.

8.3.3 No Ablation of Component Contributions

The evaluation runs the full pipeline only. Disabling each component in turn and re-running the corpus would have multiplied the five-hour corpus run by the number of components, which did not fit within the time budget of the thesis. The trade-off is that we cannot say from the data how much each component (the OCI fixup, the repair loop, or the multi-phase extraction) contributes to the 97% deploy rate and the 0.73 mean fit score. Any claim about a single component's contribution in this thesis is therefore qualitative. A full ablation study is listed as future work in Chapter 9.

8.3.4 Structural Evaluation Only

The evaluation measures structural properties of the generated honeynet: whether the correct services are present, whether they are in the correct zones, and whether they are connected by the correct dependencies. It does not measure interactive deception fidelity, specifically whether the deployed services would be convincing to an attacker during active engagement.

Srinivasa et al. [3] demonstrate that honeypots with default configurations are trivially detectable through automated fingerprinting scans, identifying over 21,000 honeypot instances across IPv4 space. Aradi et al. [59] propose standardised metrics for honeypot deception quality including fingerprinting resistance, protocol fidelity, and engagement analysis. The framework's evaluation does not systematically measure these dimensions, although the QA layer includes preliminary deception surface checks (honeytoken presence,

hostname consistency, service variety) that provide initial indicators. Bridges et al. [13] argue that honeypot evaluation requires a triad of human user studies, statistical fidelity tests, and real-world deployment; the framework's benchmark-based evaluation falls into the "artificial, ex post" category in the FEDS evaluation framework [67].

8.3.5 Pre-Deploy Probing as Problem Masking

The pre-deploy probing mechanism attempts to repair containers that cannot start before removing them: it first applies deterministic repair based on container logs (e.g., adding missing environment variables), then attempts LLM-assisted repair if deterministic repair fails, and only removes the container if neither repair strategy succeeds. This three-step process increases the deploy success rate (97%) at the cost of reducing the container realisation ratio (68%). This design choice prioritizes deployment stability over completeness. Without probing, scenarios with many failing containers would report deploy failures rather than degraded deployments, producing a lower but potentially more informative deploy success rate.

8.3.6 Benchmark Design

The benchmark scenarios and their references were constructed by the author. The tier classification (easy, medium, hard) is based on service count and does not account for LLM familiarity with the requested domain, which the results show to be a stronger predictor of generation quality. The non-monotonic easy–medium pattern (20% vs. 50% pass rate) is a consequence of this classification approach.

The composite scenario-fit score (Equation 7.1) uses equal weighting of the three coverage dimensions and a heuristic violation penalty. No sensitivity analysis was conducted to assess how alternative weightings would affect the rankings.

8.3.7 Repair Loop Effectiveness

The runtime repair mechanism primarily removes failing containers rather than correcting their configurations. Tang et al. [49] formalize code repair as an exploration-exploitation tradeoff and demonstrate that repair strategies which explore fundamentally different solutions outperform those that repeatedly refine the same failing approach. The framework includes several mechanisms to prevent unproductive repair cycles (a loop guard that detects repeated image proposals and no-change proposals, a repair scorer that filters low-confidence proposals, and a daemon-entrypoint guard that prevents overriding known

daemon entrypoints), but it does not explore fundamentally different repair strategies when the initial approach fails. The repair loop remains an exploitation-only process; broadening it to include strategy exploration is an open direction.

8.3.8 Manual-Baseline Comparison

The thesis does not include an empirical study comparing the framework's automated pipeline (mean duration 616 seconds per scenario) against the time a professional developer would take to construct the same honeynet by hand. As discussed in Chapter 1, a default single-honeypot deployment can be done in roughly 30 minutes [1] but is trivially fingerprinted by Internet-wide scanners [2, 3], while a realistic multi-service honeynet has no quantified person-hour baseline in the literature; the cyber range literature offers only a qualitative "weeks to months" bound [6, 7]. A productivity study with developers timed on a subset of the benchmark scenarios is a candidate for future work.

8.4 Implications for Research and Practice

8.4.1 Implications for Research

Domain-grounded generation. The service recall gap between well-known and niche domains (medium 0.98 vs. easy 0.77) indicates that LLM-based infrastructure generation benefits from domain-specific grounding. The framework already includes a BM25-based retrieval module with 64 service categories that injects scenario-relevant image profiles into the extraction prompt; however, this retrieval operates at the image level, not at the reference-architecture level. Extending retrieval to inject domain-specific architecture templates (e.g., what a DNS sinkhole deployment requires) into the generation prompt is a candidate for addressing this gap. Béchard and Marquez Ayala [70] demonstrate that retrieval-augmented generation reduces hallucination in structured output tasks.

Dependency-aware generation. Dependency satisfaction is the weakest evaluation dimension (mean 0.55 in the hard tier). Future work on dependency-aware extraction, where the LLM explicitly reasons about data flow paths rather than generating services independently, could address this gap.

Evaluation methodology. The single-run, single-model evaluation design limits the conclusions that can be drawn from the results. Future evaluations should include multi-

run protocols with variance reporting, multi-model comparisons, and, where possible, naturalistic evaluation through deployment in operational environments [67].

Deception quality. The evaluation measures structural correctness but not deception quality. Aradi et al. [59] propose standardised metrics for honeypot evaluation. Integrating such metrics into the framework's QA layer would extend the evaluation from deployability and scenario fit to deception effectiveness.

8.4.2 Implications for Practice

The framework's 97% deploy success rate and mean pipeline duration of 616 seconds (approximately 10 minutes) indicate that LLM-based honeynet generation is operationally feasible for scenarios where the requested architecture falls within the LLM's domain knowledge. The medium-tier results (50% overall scenario fit, 0.87 composite score) show that well-known technology stacks (web application + database + cache + monitoring) can be generated with reasonable fidelity.

For practitioners deploying the framework, the results suggest two operational guidelines: (1) prompts should name concrete technologies rather than abstract service categories, as this correlates with higher service recall; and (2) generated deployments should be reviewed for zone placement correctness, as placement violations are the primary source of failure for scenarios where all required services are present.

The framework's repair mechanisms are effective at producing deployable infrastructure but do not guarantee scenario fidelity. In operational use, the framework is better suited as a generation accelerator that produces a first draft for a human operator to review and refine, rather than as a fully autonomous deployment tool.

9. Conclusion and Future Work

9.1 Summary

This thesis presented the design, implementation, and evaluation of a framework that generates container-based honeynet environments from natural-language descriptions using large language models. The framework implements a multi-stage pipeline: LLM-based extraction of a structured intermediate representation (WorldModel), structural validation, OCI-driven image fixup, deterministic compilation into OpenTofu artefacts, deployment, and post-deployment verification.

The framework was evaluated against a benchmark corpus of 30 scenarios across three difficulty tiers (easy, medium, hard), using operationalised metrics for deployability, scenario fit, repair effectiveness, and complexity sensitivity. The evaluation followed a Design Science Research methodology, positioning the framework as a research artefact that generates prescriptive knowledge about LLM-based infrastructure generation.

9.2 Key Findings

The evaluation produced the following principal findings, corresponding to the three research questions:

RQ1 (Deployability). The pipeline completed the full deployment path in 29 of 30 scenarios (97%). The container realisation ratio (the fraction of planned containers that remained running) was 68% overall and 38% in the hard tier. The mean QA pass rate across all deployed scenarios was 98.6%. The pipeline reliably produces deployable infrastructure, but a substantial fraction of planned containers are lost during pre-deploy probing and runtime repair. Validation repair was needed in only 1 of 30 scenarios, image validation passed at 100% across all scenarios, and deploy retries occurred in 77% of runs. The total container loss rate increased from 8.8% (easy) to 62.1% (hard).

RQ2 (Fidelity). The overall scenario fit rate was 23% (7 of 30 scenarios). The mean planned composite scenario-fit score was 0.73. Service recall (0.86), zone coverage (0.89), and dependency satisfaction (0.73) were the three contributing dimensions. The medium tier achieved the highest scenario fidelity (mean score 0.87, 5/10 pass),

outperforming the easy tier (0.73, 2/10) on service recall and dependency satisfaction. The hard tier achieved the lowest fidelity (0.60, 0/10).

RQ3 (Scalability). Quality degradation with complexity was non-monotonic. The medium tier outperformed the easy tier on scenario fidelity because its scenarios describe well-known technology stacks that the LLM generates reliably, while several easy-tier scenarios use specialised concepts (DNS sinkhole, VPN gateway) outside the LLM's reliable domain knowledge. The hard tier showed the steepest declines in container realisation (38%) and dependency satisfaction (0.55 planned, 0.07 running).

The central insight is that *deployability and scenario fidelity are distinct quality dimensions*: the framework reliably produces infrastructure that deploys, but the deployed infrastructure does not always match the requested scenario. The bottleneck is not the pipeline mechanics but the LLM's ability to generate the correct services, dependencies, and zone assignments for the input prompt.

9.3 Contributions

This thesis makes the following contributions:

1. An end-to-end pipeline architecture for LLM-based honeynet generation, combining structured extraction, multi-stage validation, deterministic compilation, and automated deployment.
2. A benchmark corpus of 30 scenarios with machine-readable references enabling reproducible, automated evaluation of honeynet generation quality across four dimensions.
3. Empirical evidence on the capabilities and limitations of LLM-based infrastructure generation, including the finding that LLM domain familiarity is a stronger predictor of generation quality than topological complexity.
4. Design knowledge about how to architect validation and repair mechanisms for LLM-generated infrastructure, including the observation that front-loading quality through multi-phase extraction is preferable to relying on post-hoc repair loops with diminishing returns.

9.4 Future Work

The findings of this thesis suggest several directions for future research:

Multi-run evaluation. Repeating each scenario multiple times would enable variance estimation and statistical significance testing. LLM output is stochastic; single-run results are point estimates.

Ablation study. Evaluating reduced pipeline configurations (e.g., without repair, without OCI fixup, single-call instead of multi-phase extraction) would isolate the contribution of individual components.

Domain-grounded retrieval. The framework already includes a BM25-based retrieval module with 64 service categories for image-level retrieval. Extending this to inject scenario-specific service architecture templates (e.g., what a DNS sinkhole deployment requires) into the extraction prompt could address the domain familiarity gap identified in the easy and hard tiers.

Multi-model comparison. The framework already supports three LLM providers (OpenAI, Anthropic, Ollama). Evaluating the pipeline with different LLMs would establish whether the findings generalise beyond GPT-4o.

Runtime deception assessment. Extending the evaluation beyond structural correctness to interactive deception fidelity, specifically whether the honeynet would deceive an attacker, would require adversarial testing and builds on the fingerprinting and deception quality literature reviewed in Chapter 3.

Feedback from deployed honeynets. Connecting the framework to actual attack telemetry would enable a closed loop: deploy a honeynet, observe attacker behaviour, and use the observations to inform the generation of improved honeynets.

References

[1] HackerTarget. *Cowrie Honeypot Analysis — 24hrs of Attacks*. https://hackertarget.com/cowrie-honeypot-analysis-24hrs/. Practitioner blog; deployment-time reference of 30 minutes for default Cowrie instance. 2017.

[2] Alexander Vetterl and Richard Clayton. "Bitter Harvest: Systematically Fingerprinting Low- and Medium-Interaction Honeypots at Internet Scale". In: *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. USENIX Association, 2018. URL: https://www.usenix.org/conference/woot18/presentation/vetterl.

[3] Shreyas Srinivasa, Jens Myrup Pedersen, and Emmanouil Vasilomanolakis. "Gotta Catch 'em All: A Multistage Framework for Honeypot Fingerprinting". In: *ACM Digital Threats: Research and Practice* 4.3 (2023).

[4] Daniel Fraunholz et al. "An Adaptive Honeypot Configuration, Deployment and Maintenance Strategy". In: *19th International Conference on Advanced Communication Technology (ICACT)*. IEEE, 2017, pp. 53–57. DOI: 10.23919/ICACT.2017.7890056.

[5] Warren Z. Cabral et al. "Advanced Cowrie Configuration to Increase Honeypot Deceptiveness". In: *ICT Systems Security and Privacy Protection (IFIP SEC 2021)*. Vol. 625. IFIP Advances in Information and Communication Technology. Springer, 2021, pp. 317–331. DOI: 10.1007/978-3-030-78120-0_21.

[6] Enrico Russo, Gabriele Costa, and Alessandro Armando. "Building Next Generation Cyber Ranges with CRACK". In: *Computers & Security* 95 (2020), p. 101837. DOI: 10.1016/j.cose.2020.101837.

[7] Nestoras Chouliaras et al. "Cyber Ranges and TestBeds for Education, Training, and Research". In: *Applied Sciences* 11.4 (2021), p. 1809. DOI: 10.3390/app11041809.

[8] T. Yu, Y. Xin, and C. Zhang. "HoneyFactory: Container-Based Comprehensive Cyber Deception Honeynet Architecture". In: *Electronics* 13.2 (2024), p. 361. DOI: 10.3390/electronics13020361.

[9] George Kokolakis et al. "HoneyChart: Automated Honeypot Management over Kubernetes". In: *Computer Security. ESORICS 2022 International Workshops*. Vol. 13785. Lecture Notes in Computer Science. Springer, 2022, pp. 321–328.

[10] Fang Liu et al. “Beyond Functional Correctness: Exploring Hallucinations in LLM-Generated Code”. In: *arXiv preprint arXiv:2404.00971* (2024). DOI: 10.48550/arXiv.2404.00971.

[11] Patrick Tser Jern Kon et al. “IaC-Eval: A Code Generation Benchmark for Cloud Infrastructure-as-Code Programs”. In: *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*. 2024.

[12] Kalahasti Ganesh Srivatsa et al. “A Survey of using Large Language Models for Generating Infrastructure as Code”. In: *Proceedings of the 20th International Conference on Natural Language Processing (ICON)*. Goa, India, 2023.

[13] Robert A. Bridges et al. “SoK: Honeypots and LLMs, More Than the Sum of Their Parts?” In: *arXiv preprint arXiv:2510.25939* (2025).

[14] Lance Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley, 2003. ISBN: 0321108957.

[15] Niels Provos and Thorsten Holz. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*. Addison-Wesley Professional, 2007. ISBN: 9780321336323.

[16] Ashish Vaswani et al. “Attention Is All You Need”. In: *Advances in Neural Information Processing Systems 30 (NeurIPS)*. 2017, pp. 5998–6008.

[17] OpenAI. “GPT-4 Technical Report”. In: *arXiv preprint arXiv:2303.08774* (2023).

[18] Ziwei Ji et al. “Survey of Hallucination in Natural Language Generation”. In: *ACM Computing Surveys* 55.12 (2023).

[19] Brandon T. Willard and Rémi Louf. “Efficient Guided Generation for Large Language Models”. In: *arXiv preprint arXiv:2307.09702* (2023).

[20] Saibo Geng et al. “JSONSchemaBench: A Rigorous Benchmark of Structured Outputs for Language Models”. In: *arXiv preprint arXiv:2501.10868* (2025).

[21] Kief Morris. *Infrastructure as Code: Managing Servers in the Cloud*. O’Reilly Media, 2016. ISBN: 9781491924358.

[22] OpenTofu Contributors. *OpenTofu: A Fork of Terraform*. https://opentofu.org/. Linux Foundation project, MPL-2.0 license. 2023.

[23] HashiCorp. *Terraform Documentation*. https://developer.hashicorp.com/terraform. 2024.

[24] Docker, Inc. *Docker Networking Overview*. https://docs.docker.com/engine/network/. 2024.

[25] Open Container Initiative. *OCI Distribution Specification*. https://github.com/opencontainers/distribution-spec. 2021.

[26] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. “Automated Program Repair in the Era of Large Pre-trained Language Models”. In: *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. 2023, pp. 1482–1494.

[27] Aman Madaan et al. “Self-Refine: Iterative Refinement with Self-Feedback”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2023.

[28] Mayur Amarnath Palavalli and Mark Santolucito. “Using a Feedback Loop for LLM-based Infrastructure as Code Generation”. In: *arXiv preprint arXiv:2411.19043* (2024).

[29] Mark Chen et al. “Evaluating Large Language Models Trained on Code”. In: *arXiv preprint arXiv:2107.03374* (2021).

[30] Tianyu Zhang et al. “Deployability-Centric Infrastructure-as-Code Generation: Fail, Learn, Refine, and Succeed through LLM-Empowered DevOps Simulation”. In: *arXiv preprint arXiv:2506.05623* (2025).

[31] Manel Begoug, Mouna Chouchen, and Ali Ouni. “TerraMetrics: An Open Source Tool for Infrastructure-as-Code (IaC) Quality Metrics in Terraform”. In: *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension (ICPC)*. ACM, 2024. DOI: 10.1145/3643916.3644439.

[32] Nestoras Chouliaras et al. “A novel autonomous container-based platform for cyber-security training and research”. In: *PeerJ Computer Science* 9 (2023), e1574.

[33] Chetan Gupta, Tom van Ede, and Andrea Continella. “HoneyKube: Designing and Deploying a Microservices-based Web Honeypot”. In: *44th IEEE Symposium on Security and Privacy Workshops (SPW)*. IEEE, 2023.

[34] Uphar Bartwal et al. “Security Orchestration, Automation, and Response Engine for Deployment of Behavioural Honeypots”. In: *2022 IEEE Conference on Dependable and Secure Computing (DSC)*. IEEE, 2022.

[35] L. J. Moeller. “An Adaptive Multi-Layered Honeynet Architecture for Threat Behavior Analysis via Deep Learning”. In: *arXiv preprint arXiv:2512.07827* (2025).

[36] Gabriele Costa, Enrico Russo, and Alessandro Armando. “Automating the Generation of Cyber Range Virtual Scenarios with VSDL”. In: *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications* 13.4 (2022), pp. 61–80. DOI: 10.48550/arXiv.2001.06681.

[37] Francesco Caturano et al. “ExploitWP2Docker: a Platform for Automating the Generation of Vulnerable WordPress Environments for Cyber Ranges”. In: *IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2022.

[38] Yaseen Ahmed et al. “From Abstract Prompts to Cybersecurity Labs: Automating Virtual Environment Design and Deployment with Multi-Agent Systems and LLM-Driven Orchestration”. In: *Proceedings of the 5th Intelligent Cybersecurity Conference (ICSC)*. IEEE, 2025. DOI: 10.1109/ICSC65596.2025.11140357.

[39] Jiawei Liu et al. “Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2023.

[40] Viren Abhyankar et al. “Multi-IaC-Eval: Benchmarking Cloud Infrastructure as Code Across Multiple Formats”. In: *arXiv preprint arXiv:2509.05303* (2025).

[41] Prithwish Jana et al. “TerraFormer: Automated Infrastructure-as-Code with LLMs Fine-Tuned via Policy-Guided Verifier Feedback”. In: *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2026.

[42] Niklas Beuter, Andre Drews, and Nane Kratzke. “Prompt-Driven and Kubernetes Error Report-Aware Container Orchestration”. In: *Future Internet* 17.416 (2025). DOI: 10.3390/fi17100416.

[43] Deep Mehta et al. “Automated DevOps Pipeline Generation for Code Repositories using Large Language Models”. In: 2023.

[44] Changjie Wang et al. “NetConfEval: Can LLMs Facilitate Network Configuration?” In: *Proceedings of the ACM on Networking* 2.CoNEXT2 (2024). DOI: 10.1145/3656296.

[45] Noah Shinn et al. “Reflexion: Language Agents with Verbal Reinforcement Learning”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2023.

[46] Xinyun Chen et al. “Teaching Large Language Models to Self-Debug”. In: *Proceedings of the International Conference on Learning Representations (ICLR)*. 2024.

[47] Zhangqian Bi et al. “Iterative Refinement of Project-Level Code Context for Precise Code Generation with Compiler Feedback”. In: *arXiv preprint arXiv:2403.16792* (2024).

[48] Antonio Mastropaolo et al. “Towards the Self-Healing of Infrastructure as Code Projects Using Constrained LLM Technologies”. In: *5th ACM/IEEE International Workshop on Automated Program Repair (APR)*. ACM, 2024.

[49] Hao Tang et al. “Code Repair with LLMs gives an Exploration-Exploitation Tradeoff”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2024.

[50] Mojtaba Eshghie. “Repairing Language Model Pipelines by Meta-Self-Refining Competing Constraints at Runtime”. In: *arXiv preprint arXiv:2507.10590* (2025).

[51] Sirui Hong et al. “MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework”. In: *Proceedings of the Twelfth International Conference on Learning Representations (ICLR)*. 2024.

[52] Chen Qian et al. “ChatDev: Communicative Agents for Software Development”. In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Bangkok, Thailand, 2024, pp. 15174–15186.

[53] Dong Huang et al. “AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation”. In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*. Bangkok, Thailand, 2024.

[54] Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. “MapCoder: Multi-Agent Code Generation for Competitive Problem Solving”. In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Bangkok, Thailand, 2024.

[55] Junda He, Christoph Treude, and David Lo. “LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead”. In: *ACM Transactions on Software Engineering and Methodology* 34 (2025).

[56] Rana Nameer Hussain Khan et al. “Multi-Agent Code-Orchestrated Generation for Reliable Infrastructure-as-Code”. In: *arXiv preprint arXiv:2510.03902* (2025).

[57] Qingyun Wu et al. “AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation”. In: *arXiv preprint arXiv:2308.08155* (2023).

[58] Haolin Jin et al. “Towards Advancing Code Generation with Large Language Models: A Research Roadmap”. In: *arXiv preprint arXiv:2501.11354* (2025).

[59] Zoltán Aradi et al. “Metrics-Driven Evaluation and Optimization of Honeypots: Toward Standardized Measures of Deception Effectiveness”. In: *Acta Polytechnica Hungarica* 22.12 (2025), pp. 295–313.

[60] A. Javadpour et al. “A Comprehensive Survey on Cyber Deception Techniques to Improve Honeypot Performance”. In: *Computers & Security* 140 (2024), p. 103792.

[61] Nils Spahn et al. “Container Orchestration Honeypot: Observing Attacks in the Wild”. In: *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. ACM, 2023.

[62] Maria Sladic et al. “LLM in the Shell: Generative Honeypots”. In: *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2024.

[63] Shihab Ahmed et al. “SPADE: Enhancing Adaptive Cyber Deception Strategies with Generative AI and Structured Prompt Engineering”. In: *arXiv preprint arXiv:2501.00940* (2025).

[64] Ken Peffers et al. “A Design Science Research Methodology for Information Systems Research”. In: *Journal of Management Information Systems* 24.3 (2007), pp. 45–77.

[65] Alan R. Hevner et al. “Design Science in Information Systems Research”. In: *MIS Quarterly* 28.1 (2004), pp. 75–105.

[66] Shirley Gregor and Alan R. Hevner. “Positioning and Presenting Design Science Research for Maximum Impact”. In: *MIS Quarterly* 37.2 (2013), pp. 337–355.

[67] John Venable, Jan Pries-Heje, and Richard Baskerville. “FEDS: A Framework for Evaluation in Design Science Research”. In: *European Journal of Information Systems* 25.1 (2016), pp. 77–89.

[68] Indraneil Paul et al. “IRCoder: Intermediate Representations Make Language Models Robust Multilingual Code Generators”. In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Bangkok, Thailand, 2024, pp. 15023–15041.

[69] Dong-Wook Kim et al. “Framework for Network Topology Generation and Traffic Prediction Analytics for Cyber Exercises”. In: *IEEE Access* 12 (2024).

[70] Patrice Béchard and Orlando Marquez Ayala. “Reducing Hallucination in Structured Outputs via Retrieval-Augmented Generation”. In: *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Industry Track*. 2024.

[71] Hanbin Wang et al. “INTERVENOR: Prompting the Coding Ability of Large Language Models with the Interactive Chain of Repair”. In: *Findings of the Association for Computational Linguistics: ACL 2024*. 2024.

[72] Thai Tang Quoc et al. “An Empirical Study on Self-correcting Large Language Models for Data Science Code Generation”. In: *arXiv preprint arXiv:2408.15658* (2024).

[73] Richard Meyes et al. “Ablation Studies in Artificial Neural Networks”. In: *arXiv preprint arXiv:1901.08644* (2019).

[74] Michela Nardo et al. *Handbook on Constructing Composite Indicators: Methodology and User Guide*. OECD Publishing, 2008. DOI: 10.1787/9789264043466-en.

[75] Salvatore Greco et al. “On the Methodological Framework of Composite Indices: A Review of the Issues of Weighting, Aggregation, and Robustness”. In: *Social Indicators Research* 141.1 (2019), pp. 61–94. DOI: 10.1007/s11205-017-1832-9.

[76] FIRST. *Common Vulnerability Scoring System v4.0: Specification Document*. https://www.first.org/cvss/specification-document. 2023.

[77] Enrico Casadio Tarabusi and Giulio Guarini. "Aggregating Composite Indicators through the Geometric Mean: A Penalization Approach". In: *Computation* 10.4 (2022), p. 64. DOI: 10.3390/computation10040064.

[78] Soroush Hashemi Doulabi, Nicolas Bettenburg, and Foutse Khomh. "IaC Generation with LLMs: An Error Taxonomy and A Study on Configuration Knowledge Injection". In: *arXiv preprint arXiv:2512.14792* (2025).

[79] Dipkamal Bhusal et al. "SECURE: Benchmarking Large Language Models for Cybersecurity". In: *arXiv preprint arXiv:2405.20441* (2024).

[80] Md Tanvirul Alam et al. "CTIBench: A Benchmark for Evaluating LLMs in Cyber Threat Intelligence". In: *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*. 2024.

[81] Berk Atil et al. "LLM Stability: A Detailed Analysis with Some Surprises". In: *arXiv preprint arXiv:2408.04667* (2024).

[82] Shuyin Ouyang et al. "An Empirical Study of the Non-determinism of ChatGPT in Code Generation". In: *ACM Transactions on Software Engineering and Methodology* (2024). DOI: 10.1145/3697010.

[83] Yifan Song et al. "The Good, The Bad, and The Greedy: Evaluation of LLMs Should Not Ignore Non-Determinism". In: *Proceedings of the 2025 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*. 2025.

Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis[1]

I Nikolaus Riedl

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Automated Honeynet Generation from Natural Language: An LLM-Based Pipeline with Multi-Stage Validation and Empirical Evaluation", supervised by Hayretdin Bahsi
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

27.05.2026

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – Benchmark Corpus Overview

Table 17 lists all 30 benchmark scenarios with their difficulty tier, prompt summary, and reference complexity.

ID	Tier	Prompt Summary	Svc	Zones	Deps
e01	Easy	Web frontend + PostgreSQL database	4	3	2
e02	Easy	Corporate LDAP directory	3	2	2
e03	Easy	Web + Redis cache + PostgreSQL	4	2	2
e04	Easy	Prometheus + Grafana monitoring	4	2	2
e05	Easy	RabbitMQ message broker	3	2	2
e06	Easy	MinIO object storage data lake	3	2	2
e07	Easy	DNS sinkhole with recursive resolver	3	2	2
e08	Easy	VPN concentrator + auth proxy + wiki	3	2	2
e09	Easy	Nextcloud file share + LDAP + antivirus	3	2	2
e10	Easy	CI runner + MinIO + PostgreSQL	3	2	2
m01	Med.	E-commerce (storefront + catalogue + payment)	7	4	4
m02	Med.	Corporate email (webmail + SMTP + LDAP)	6	3	4
m03	Med.	HIPAA healthcare (portal + EHR + FHIR)	6	3	4
m04	Med.	DevOps (Gitea + Jenkins + registry + monitoring)	6	3	4
m05	Med.	Supply chain (vendor portal + inventory + docs)	6	3	3
m06	Med.	Kafka streaming (Zookeeper + ClickHouse + API)	6	3	4

ID	Tier	Prompt Summary	Svc	Zones	Deps
m07	Med.	SCADA (HMI + historian + OPC-UA + syslog)	5	3	3
m08	Med.	Crypto exchange (trading + wallet + KYC)	6	3	3
m09	Med.	Ad platform (serving + click tracker + fraud)	6	3	4
m10	Med.	Game backend (API + matchmaking + NATS chat)	6	3	4
h01	Hard	Core banking (6 zones, 11 services)	11	6	6
h02	Hard	Telecom mediation (signaling + CDR + compliance)	9	5	5
h03	Hard	ML platform (inference + feature store + MLflow)	10	4	6
h04	Hard	Zero-trust enterprise (mTLS + Vault + Keycloak)	9	6	6
h05	Hard	Industrial IoT (MQTT + Kafka + InfluxDB + fleet)	9	5	6
h06	Hard	Government citizen portal (7 zones, 11 services)	11	6	6
h07	Hard	Satellite ground station (telemetry + HSM + audit)	8	5	5
h08	Hard	Smart city (MQTT + traffic + dispatch + Kafka)	8	5	5
h09	Hard	Military C2 (command + SA + tactical DB + NATS)	9	5	5
h10	Hard	Pharma research (LIMS + compound DB + regulatory)	9	5	6

Table 17. Complete benchmark corpus. Svc = required services, Zones = required zones, Deps = required dependencies. Med. = Medium.

Appendix 3 – Benchmark Reference Example

The following listing shows an abbreviated benchmark reference for scenario e01 (web frontend + PostgreSQL database). Each benchmark reference is a YAML document that defines the minimum structural requirements a generated honeynet must satisfy.

```
scenario_id: e01_web_db
difficulty: easy
prompt_variant: detailed
prompt_text: >
  Model a simple two-tier corporate application:
  a web frontend serving HTTP traffic and a
  PostgreSQL database holding application records.

reference:
  required_services:
    - id: web_frontend
      kinds_any: [web]
      archetypes_any: [nginx, caddy, httpd, traefik]
    - id: postgres_db
      kinds_any: [database]
      archetypes_any: [postgres]
    - id: admin_panel
      kinds_any: [web]
      roles_any: [admin, management]
    - id: log_collector
      kinds_any: [monitor, infra]
      archetypes_any: [fluentd, filebeat, vector]

  required_zones:
    - id: public_zone
      internal: false
    - id: data_zone
      internal: true
    - id: mgmt_zone
      internal: true

  required_dependencies:
    - source_service_id: web_frontend
```

```
    target_service_id: postgres_db
  - source_service_id: admin_panel
    target_service_id: postgres_db

forbidden_placements:
  - service_id: postgres_db
    zone_id: public_zone
  - service_id: admin_panel
    zone_id: public_zone
```

The matching criteria use broad categories (`kinds_any`, `archetypes_any`) rather than exact system names. A generated honeynet passes the benchmark when all required services, zones, and dependencies are matched and no forbidden placements are violated. The generated result may contain additional systems, zones, and dependencies beyond what the reference requires.