UNIVERSITY OF SALERNO

Department of Computer Science

Master Degree in Computer Science
Major in Software Engineering & IT Management

THESIS

Infrastructure Intent Discovery via TOSCA-based Reverse Engineering

SUPERVISOR
Prof. **Fabio Palomba**
University of Salerno

Prof. **Damian A. Tamburri**
Dott. **Stefano Fossati**
Jheronimus Academy of Data Science

CANDIDATE
Dario Mazza
Student ID: 0522501553

Academic Year 2024-2025

This thesis was carried out in

Jheronimus Academy of Data Science

SOFTWARE ENGINEERING SALERNO

"I tell you what, this has been really fun. And I got to help make something pretty cool, so I've got no complaints. I mean, not me, exactly, but close enough. It's the kind of thing that makes you glad you stopped and smelled the pine trees along the way, you know?"

— Gabbro, *Outer Wilds*

Abstract

Modern infrastructures are increasingly defined using heterogeneous Infrastructure-as-Code (IaC) frameworks, resulting in fragmented, vendor-specific definitions that hinder portability and interoperability. This thesis addresses this challenge by introducing a methodology for reverse engineering IaC specifications into standardized, vendor-neutral TOSCA 2.0 templates, effectively discovering architectural intent.

Following an Action Design Research methodology, the study develops and contrasts two pipelines: a deterministic, rule-based translator and an exploratory AI-augmented system using Large Language Models. The deterministic framework, implemented as a modular proof-of-concept, successfully translates complex Terraform configurations by preserving semantic intent through multi-cue relationship inference and ensuring verifiability. In contrast, the AI-driven approach, while feasible for single resources, proved unreliable for topology reconstruction and suffered from non-determinism.

A key practical contribution is the extension of the open-source Puccini compiler to support the TOSCA 2.0 standard. Ultimately, this work delivers both a validated translation framework and an empirical analysis of its underlying design principles, concluding that a deterministic core is essential for reliable reverse engineering, while positioning AI as a promising assistive tool for future development.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

The widespread adoption of cloud computing has fundamentally reshaped the landscape of information technology, offering unprecedented scalability, flexibility, and efficiency. However, managing the inherent complexity of distributed cloud infrastructure has become a primary challenge for modern enterprises. This chapter introduces the evolution of infrastructure management that led to the central problem addressed in this research: the lack of interoperability in cloud automation. We outline the specific challenges posed by proprietary Infrastructure-as-Code (IaC) tools and present our proposed solution for bridging this gap through automated reverse engineering.

1.1 The Evolution of Infrastructure Management

The way that businesses supply, oversee, and run their computer infrastructure has undergone a significant change over the past 20 years. Elasticity, cost optimization, and quick deployment cycles have been the main factors driving the move from conventional on-premises data centers to cloud-based settings. The complexity of managing infrastructure has increased dramatically as more businesses use cloud computing. This includes not only computing and storage resources but also networking elements, security regulations, monitoring systems, and workflow orchestration across various cloud providers and geographical locations [1, 2].

According to recent industry analyses, the global multi-cloud management market was valued at USD 12.52 billion in 2024 and is projected to expand from USD 16.02 billion in 2025 to approximately USD 147.12 billion by 2034, registering a CAGR of 27.94% during 2025–2034 [3]. This growth is driven by enterprises seeking flexibility, cost optimization, and risk mitigation through diversified cloud strategies. North America currently holds the largest revenue share (36

To address this complexity, the IaC paradigm has become a fundamental component of contemporary DevOps workflows. Practitioners can specify infrastructure configurations as declarative or imperative code that can be version-controlled, reviewed, tested, and automatically deployed with the help of IaC tools like Terraform, AWS CloudFormation, Azure Resource Manager, and Ansible. This method has made it possible to treat infrastructure with the same level of rigor as application code, greatly increasing reproducibility and decreasing human error [1].

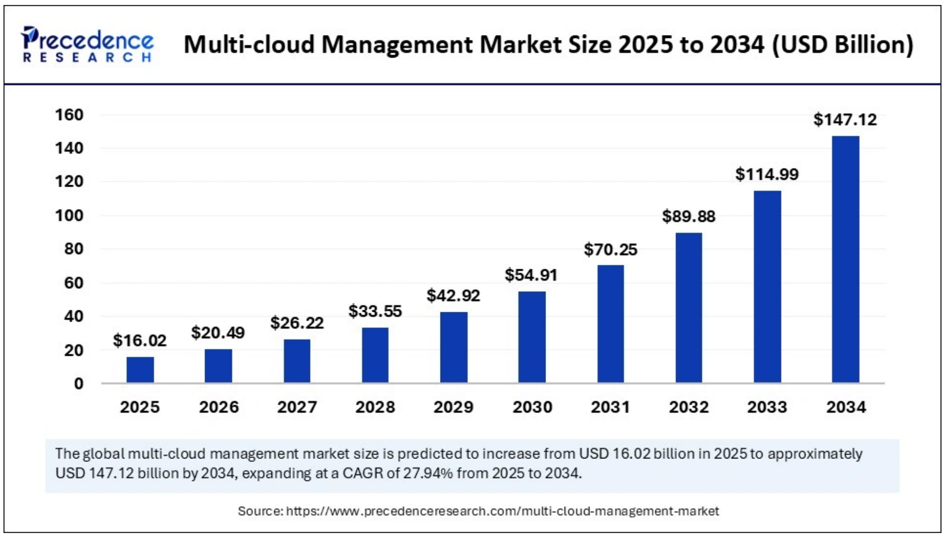


Figure 1.1: Projected growth of the multi-cloud management market (2025–2034) [3].

1.2 Problem Statement

While IaC has solved many challenges, its widespread adoption has unintentionally created a new class of issues centered on interoperability and vendor lock-in. The majority of IaC tools use proprietary domain-specific languages that are not portable or are closely linked to particular cloud providers. A Terraform configuration created for AWS infrastructure, for example, cannot be easily moved to Azure or Google Cloud Platform without requiring significant rewriting. Because workload migration is challenging, this vendor-specific coupling limits the negotiating power with cloud providers, raises the total cost of ownership, and poses strategic risks for enterprises pursuing multi-cloud or hybrid cloud strategies [4].

Furthermore, cooperation and knowledge exchange between organizations are hampered by the fragmentation of the IaC ecosystems. The introduction of distinct syntaxes, execution models, and resource abstractions by each IaC tool results in challenging learning curves for practitioners and disperses the community's endeavors to create reusable infrastructure patterns. Standardized methods for infrastructure management, orchestration, and lifecycle automation cannot be developed because cloud application topologies lack a single, vendor-neutral representation.

Industry consortia and standardization bodies have proposed several cloud application modeling standards to address interoperability issues. Among these, the Topology and Orchestration Specification for Cloud Applications (TOSCA) has become a comprehensive standard that describes the dependencies, lifecycle management, and structure of cloud applications in a vendor-agnostic and portable manner [5]. TOSCA, which was initially created as part of the OASIS consortium[1], offers a rich type system, relationship modeling tools, and workflow definitions that let applications be described without reference to the underlying infrastructure provider.

Moreover, the industry is increasingly moving toward intent-based orchestration paradigms, wherein infrastructure management shifts from imperative, step-by-step instructions to declarative and high-level intent specifications [6]. Intel's recent work on intent-driven orchestration highlights this trend [7], emphasizing that future cloud systems should enable practitioners to specify the outcomes they desire rather than how to achieve them. However, the vast majority of existing infrastructure is defined by the *how* that these new paradigms seek to abstract. This is precisely where this study's contribution lies. The proposed reverse engineering methodology serves as a form of intent discovery, a systematic process for extracting high-level

[1]`https://www.oasis-open.org/`

architectural goals (*what*) from low-level, provider-specific implementations (*how*). TOSCA 2.0, with its declarative nature and rich semantic modeling capabilities, is inherently aligned with this vision, offering a standardized foundation for expressing this recovered infrastructure intent so that it can be interpreted and executed across diverse orchestration engines [8].

The latest version of TOSCA 2.0[2], which introduces groundbreaking changes that modernize the specification with better syntax, increased expressiveness, and better alignment with modern cloud-native practices, marks a significant evolution of the standard. A chicken-and-egg problem has arisen because TOSCA adoption has lagged behind that of proprietary IaC tools despite these benefits. Organizations have made significant investments in existing IaC codebases, especially Terraform, which leads the multi-cloud IaC market, whereas TOSCA-based orchestrators lack sufficient real-world workloads to prove their worth.

By examining automated reverse engineering methods that can convert current IaC configurations into TOSCA 2.0 representations, this study addresses this adoption barrier. Given Terraform's market dominance and AWS's vast service catalog, we specifically concentrated on Terraform configurations aimed at AWS infrastructure as our first proof of concept. The main argument of this study is that by offering automated migration pathways from well-known IaC tools to standardized representations, we can:

- Lower the entrance barrier for businesses looking to implement TOSCA-based strategies without sacrificing current infrastructure definition.
- Allow phased migration plans in which TOSCA-planned deployments and legacy IaC coexist.
- Transform vendor-specific configurations into vendor-neutral TOSCA templates that can be modified for various cloud targets to enable multi-cloud portability.
- Make current infrastructure knowledge immediately available in TOSCA format to hasten the development of the TOSCA ecosystem.

[2]`https://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.html`

We created a dual-strategy framework to accomplish these objectives, fusing the flexibility of AI-enhanced methods with the accuracy of deterministic rule-based translation.

1. **Rule-Based Translation:** In order to convert Terraform resource types, attributes, and dependencies into comparable TOSCA node types, properties, and relationships, we methodically define mapping rules. This method offers explicit, verifiable transformations that can be audited and maintained by subject-matter experts and ensures accuracy for the covered resource types.
2. **AI-Assisted Translation:** We performed initial tests utilizing retrieval-based methods in conjunction with large language models (LLMs) to investigate the possibilities of automation in the translation process. The aim of this study was to assess whether these models could support translation beyond manually defined rules and aid in the identification of semantic correspondences. These experiments indicate that AI may supplement deterministic methods and offer more flexibility for future extensions, even though it is still in the exploratory stage.

With rule-based methods offering guarantees but requiring manual labor to maintain, and AI-based methods offering broad coverage but introducing uncertainty that needs to be managed through validation and human review, this dual approach clearly recognizes the inherent trade-offs between automation and determinism.

Beyond the methodological contributions, our work on the Puccini project[3], a TOSCA processor and compiler, contributes in a tangible way to the open-source TOSCA ecosystem. To enable the community to parse, validate, and compile the new specification, we expanded Puccini to support the TOSCA 2.0 Standard. This contribution was examined and accepted into the official repository, guaranteeing that our study will be useful immediately and confirming that using TOSCA 2.0 in production settings is feasible.

[3] `https://github.com/tliron/go-puccini`

1.3 Research Relevance

The relevance of this study is described from two different angles. The scientific and practical significance of this study are explained in the following sections.

1.3.1 Scientific Relevance

The study has important scientific implications for model-driven development, software engineering, and cloud computing. By reversing IaC into vendor-neutral standards, it fills a significant gap in the methodical approaches to cloud infrastructure portability.

Few studies have examined the methodical reverse engineering of current IaC codebases, whereas most previous research has concentrated on forward engineering techniques that involve designing cloud applications directly in TOSCA or other vendor-neutral languages [9].

In addition to addressing this neglected issue, this study offers comparative empirical data on two radically different translation paradigms: AI-assisted generation and deterministic rule-based mapping.

The dual-strategy study sheds light on the trade-off between automation and precision in many software engineering transformation tasks. Through a systematic comparison of rule-based and AI-assisted methods in terms of accuracy, coverage, and maintainability, this study adds empirical support to the larger debate on the potential replacement or complement of conventional program transformation methods with large language models [10].

Additionally, this study improves the knowledge of Retrieval-Augmented Generation applications in contexts involving domain-specific code generation. Although RAG has demonstrated potential in general-purpose programming tasks, little is known about its use for infrastructure modeling with formal specifications such as TOSCA. This thesis validates specific architectural patterns for knowledge base construction from technical standards and offers verifiable proof of RAG's efficacy in capturing semantic mappings between infrastructure abstractions.

The methodological contribution of Action Design Research shows how design science can interact with open-source communities as organizational settings. The

Puccini contribution, which was examined and approved by community maintainers, serves as an example of how scholarly research can produce validated design knowledge and directly enhance practitioner tools. This closes the gap between research and practice, which frequently reduces the influence of scholarly work in the field of software engineering.

1.3.2 Industrial Relevance

For enterprises with significant IaC investments, the design principles and translation framework of this study directly reduce cloud vendor lock-in. The proof-of-concept offers a tangible route to infrastructure portability without requiring total rewrites by practitioners overseeing Terraform codebases aimed at AWS.

Organizations can modify their approach to fit their unique IaC technologies and cloud providers, thanks to the framework's extensible architecture, which is documented through design principles and meta-requirements. Given their limitations regarding accuracy requirements, automation needs, and available resources, practitioners can use a comparative analysis of rule-based versus AI-assisted strategies to make informed decisions about how to approach translation projects.

This study addresses a significant adoption obstacle for the TOSCA community and organizations considering implementing TOSCA: the work required to move current infrastructure definitions. This study reduces the obstacles to TOSCA adoption and fortifies the ecosystem, enabling vendor-neutral cloud orchestration.

Furthermore, this research contributes to the emerging paradigm of intent-based orchestration. By reverse engineering provider-specific IaC into TOSCA 2.0's declarative specifications, the framework enables a shift from low-level, imperative infrastructure definitions to high-level intent-focused models. This aligns with industry directions, positioning TOSCA 2.0 as a foundation for intent-driven cloud management, where infrastructure behavior is specified through semantic goals rather than procedural steps.

For the open-source community, the Puccini extension offers instant utility. This contribution guarantees that the generated templates can be validated, compiled, and possibly orchestrated using community-standard tools by turning on TOSCA

2.0 Simple Profile processing. The most recent TOSCA standard, which has not yet established any open-source processor implementations, can now be adopted more widely owing to this effort.

Finally, the study offers a methodical approach to comprehending and recording the architectures of their infrastructures for enterprises with little experience with cloud computing. Making implicit dependencies and relationships explicit is a necessary step in the translation process, which may yield architectural insights that are useful outside the portability use case.

1.4 Research Questions

In order to enable infrastructure portability and interoperability across heterogeneous cloud environments, the main objective of this work is to create a methodical approach for reverse engineering IaC artifacts into standardized TOSCA 2.0 representations. The following research questions were developed to organize this study:

Main Research Question

MRQ. *How can provider-specific IaC files be systematically and accurately reverse-engineered into the vendor-neutral TOSCA 2.0 standard?*

Sub-questions

To decompose the primary question into tractable components, we identify three complementary sub-questions:

RQ$_1$. *What are the design principles for a deterministic, rule-based mapping from Terraform to TOSCA 2.0?*

The focus of this question is the creation of a methodical, verifiable translation technique that converts Terraform resource definitions into semantically equivalent TOSCA node types, relationships, and topology templates using explicit mapping rules. The main goal is to find guidelines that guarantee semantic preservation, rule-set maintainability, and extensibility to new resource types [11].

🔍 **RQ$_2$.** *To what extent can Large Language Models augmented with RAG support automated IaC-to-TOSCA translation? A preliminary investigation of feasibility and limitations.*

Motivated by recent surveys on LLM applications in infrastructure code generation [12], this preliminary investigation explored the potential of RAG-augmented translation methods. This study examines the RAG architecture design for TOSCA domain knowledge integration, prompt engineering techniques for IaC-to-TOSCA mapping, and structured output validation to constrain LLM generation. While contemporary AI techniques show promise for capturing semantic correspondences beyond explicit rules, this exploratory work focuses on identifying feasibility boundaries and technical challenges rather than delivering production-ready automation.

🔍 **RQ$_3$.** *What are the observed trade-offs between deterministic rule-based mapping and AI-augmented approaches in the context of IaC-to-TOSCA translation?*

This comparative question examines both approaches along multiple dimensions, including translation accuracy characteristics, resource type coverage patterns, maintenance and extension effort requirements, and edge-case handling strategies. Rather than providing quantitative performance metrics, this study identifies qualitative trade-offs between the two paradigms, observes scenarios in which each approach demonstrates relative strengths, and establishes an evaluation framework for future rigorous comparative analyses.

1.5 Research Scope

This thesis establishes precise parameters for its investigation in order to successfully address the research questions while upholding methodological rigor and practical viability. Technical, methodological, and contextual aspects that frame the contributions and recognize intentional limitations are included in the scope.

1.5.1 Technical Scope

- **IaC Technology Focus:** Terraform is the primary source of IaC technology in this study. Terraform was chosen because of its broad use in businesses, declarative syntax that makes methodical analysis easier, and provider-agnostic architecture that supports various cloud platforms [13]. Although the framework's extensibility principles allow for future adaptation to other IaC technologies (such as Azure Resource Manager or AWS CloudFormation), only Terraform configurations were used for the proof-of-concept implementation and evaluation.
- **Cloud Provider Focus:** The research focuses on Amazon Web Services (AWS) resources within the Terraform ecosystem. AWS is the leading public cloud platform with a wide range of infrastructure services covering computing, storage, networking, databases, and application services [14]. The ability to conduct local, reproducible testing with LocalStack[4], which mimics essential AWS services and permits quick, affordable integration tests and continuous integration workflows without deploying to the public cloud, is another practical justification for concentrating on AWS. Thus, by selecting AWS, it is possible to investigate translation problems across various resource types and intricate inter-resource relationships while keeping the proof-of-concept's scope manageable. However, the mapping framework is designed to support multi-cloud scenarios and future extensions to other providers such as Azure and Google Cloud Platform.
- **Extensibility Validation:** While the framework architecture is designed to support multiple IaC technologies through a plug-in-based approach, the implementation validates this extensibility primarily through architectural analysis rather than multiple working implementations.
- **Target Standard:** TOSCA 2.0 (core) is the translation target. Since version 2.0, the Simple Profile has been maintained by the open-source TOSCA community and is available for import as an external profile; it is no longer a part of the normative specification. this research adopts that community's Simple Profile[5] as a practical starting point because it offers a ready-made set of modeling idioms

[4]`https://github.com/localstack/localstack`
[5]`https://github.com/oasis-open/tosca-community-contributions`

and types that were initially part of TOSCA 1.3, which speeds up development while maintaining model alignment with TOSCA 2.0 through imports.

- **Resource Coverage:** The study gives top priority to core infrastructure resources within the AWS provider that are commonly found in actual Terraform configurations, such as compute instances, virtual networks, storage volumes, security groups, load balancers, and database services. For these fundamental resource types, the translation framework creates patterns that can be later expanded to the entire AWS service catalog. The proof-of-concept acknowledges but does not fully address edge cases involving highly specialized AWS services or previewed features.

1.5.2 AI Approach Scope

The investigation into AI-assisted translation is structured as a systematic feasibility study (ADR Cycle 3), focusing on the application of Large Language Models (LLMs) enhanced with Retrieval-Augmented Generation (RAG). The primary objective of this study was to evaluate the viability of this approach for automating IaC-to-TOSCA translation.

The scope of this study includes the following:

- The development of a specialized knowledge base from official TOSCA 2.0 specifications and Terraform provider documentation.
- An investigation into prompt engineering strategies and RAG architectures to optimize translation quality.
- The implementation of quality assurance procedures for validating the accuracy and compliance of LLM-generated outputs.

To maintain a focused investigation, alternative machine learning techniques (such as graph neural networks for topology inference, supervised fine-tuning with annotated data, and reinforcement learning) were explicitly excluded from the scope of this study. Given the exploratory nature of this investigation, RQ2 addresses feasibility and preliminary findings, rather than definitive performance claims. Cycle 3 served to establish whether RAG-enhanced LLMs can meaningfully contribute to IaC translation, identify key technical challenges, and outline directions for future rigorous evaluation.

1.5.3 Excluded from Scope

The following elements are specifically left out in order to preserve the feasibility and focus of the research:

- **Orchestration at Runtime:** Static translation of IaC definitions to TOSCA templates is the main focus of the study. Runtime orchestration issues, such as deployment execution, state management, provisioning error handling, and integration with particular orchestration engines beyond Puccini's compilation capabilities, are not covered.
- **Reverse Translation:** The framework is unidirectional, translating from provider-specific IaC to vendor-neutral TOSCA. Translation from TOSCA back to provider-specific IaC (which would enable full round-trip migration) is not addressed, although the modular architecture does not preclude such extensions.
- **State Migration:** Current Terraform state files that monitor the alignment of deployed cloud resources with infrastructure definitions are not taken into account. Instead of focusing on state reconciliation or the migration of live infrastructure, this study examines the translation of infrastructure definitions as code artifacts.
- **Policy and Compliance:** While TOSCA's policy modeling capabilities facilitate policy definition and compliance verification, this study focuses primarily on topology translation. Although the framework is extensible and could support such additions, security policies, cost optimization guidelines, compliance restrictions, and governance policies are not consistently translated, in part because the community-maintained Simple Profile currently provides a relatively minimal set of policy types and modeling idioms.

1.6 Research Approach

The Action Design Research (ADR) method, introduced by Sein et al. [15] in 2011, shapes the methodology of this thesis to address the research questions. ADR provides a structured framework for conducting rigorous academic research while actively involving practitioners from the industry and open-source communities. Rather than separating theoretical advancements from practical applications, ADR explicitly integrates both perspectives through iterative cycles of artifact building, intervention, and evaluation in authentic organizational contexts. In this study, the organizational context encompassed the open-source TOSCA tooling community, where collaboration with maintainers and users enabled both the validation of the proposed artifacts and the generation of design knowledge applicable beyond the immediate project scope. This dual focus ensures that the research produces not only scholarly contributions but also tangible improvements in real-world tools and practices.

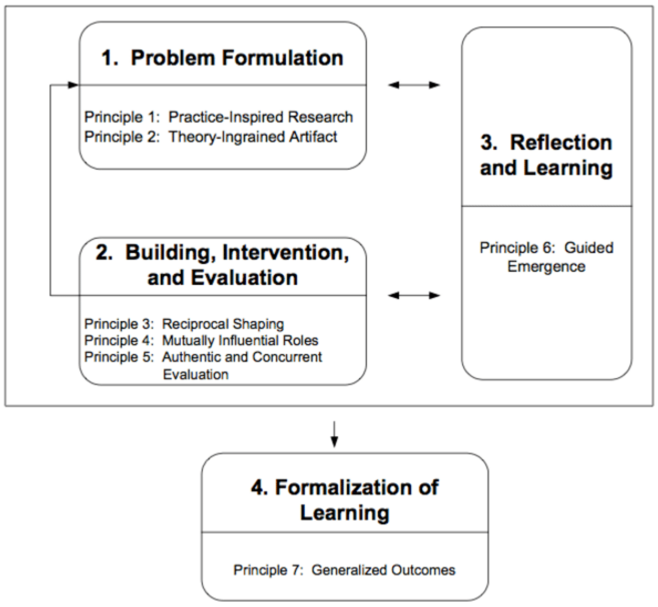

1. Problem Formulation
Principle 1: Practice-Inspired Research
Principle 2: Theory-Ingrained Artifact
2. Building, Intervention, and Evaluation
Principle 3: Reciprocal Shaping
Principle 4: Mutually Influential Roles
Principle 5: Authentic and Concurrent Evaluation
3. Reflection and Learning
Principle 6: Guided Emergence
4. Formalization of Learning
Principle 7: Generalized Outcomes

Figure 1.2: ADR process: four stages guided by seven principles [16].

Figure 1.2 illustrates the basic ADR stages and principles. In the Problem Formulation stage, this study examined the current state of IaC-to-TOSCA translation. This study involved analyzing the architecture of Terraform and AWS resource models, assessing TOSCA 2.0 features, and reviewing existing translation methods. This analysis helped establish the main requirements for the translation framework. By combining insights from a thorough literature review on cloud portability, model transformation, and AI-assisted code generation, we developed design principles that support these requirements.

The development of rule-based and AI-assisted translation techniques is the main goal of the Building, Intervention, and Evaluation phases. These methods are implemented as useful artifacts[6], and continuous cycles are used to collect and incorporate user feedback. Contributions to the Puccini open-source project are part of this process, where community reviews serve as accurate assessments. Looking back on the study's experiences generated insightful conversations about the meta-requirements and their degree of fulfillment during the Reflection and Learning stage. This helped address both RQ1 and RQ2. This reflection also made it possible to compare the two translation approaches, which addressed RQ3.

Finally, by offering prescriptive knowledge that extends beyond the particular Terraform-to-TOSCA proof-of-concept to other IaC technologies and cloud providers, the formalization of the learning stage produced generalized recommendations that are advantageous to both academia and practitioners.

1.7 Thesis Structure

This thesis is organized into nine chapters. Chapter 1 introduces the research problem, its relevance, the research questions, and the scope of this investigation. Chapter 2 provides a comprehensive literature review, establishing the theoretical background, analyzing related work, and identifying the specific research gap that this study addresses. Chapter 3 details the Action Design Research (ADR) methodology that underpins the entire research.

The core of the work is presented in Chapters 4–7. Chapter 4 formalizes the problem space and derives the meta-requirements and design principles. The subsequent three chapters document the iterative "Building, Intervention, and Evaluation" cycles: Chapter 5 describes the initial monolithic prototype (Alpha artifact); Chapter

[6]The complete source code for the artifacts developed in this thesis is publicly available at: `www.github.com/xDaryamo/ReTOSCA`

6 details the development of the modular and extensible framework (Beta artifact); and Chapter 7 presents the exploratory AI-augmented translation pipeline (Gamma artifact).

Finally, Chapter 8 offers a critical discussion of the findings, lessons learned and threats to validity. Chapter 9 concludes the thesis by answering the research questions, summarizing the contributions, acknowledging the limitations, and providing recommendations for future academic and industrial work.

CHAPTER 2

Literature Review

This chapter examines the fundamental and overlapping disciplines pertinent to this work. Infrastructure-as-Code (IaC), the development and principles of the TOSCA standard, and the current state of reverse engineering in cloud environments are the main topics covered in this paper. This chapter also places this work in the context of intent-based orchestration and the use of AI for changing code and models. Finally, the related work section brings these topics together to identify the research gap that this thesis attempts to fill.

2.1 Background

2.1.1 Infrastructure-as-Code

IaC is a big change in the way modern computing environments set up, manage, and maintain their infrastructure. Instead of manually setting up servers, networks, and storage using graphical interfaces or command-line tools, IaC allows individuals to use declarative or imperative code to describe infrastructure components. This method incorporates software engineering concepts, such as version control, automated testing, and continuous integration, into infrastructure management. This significantly changes the way things are done in cloud computing in a big way [1].

The main idea behind IaC is to treat infrastructure as a first-class software artifact. Infrastructure definitions are typically stored in version control systems as structured text files that machines can read. These files can be reviewed using collaborative workflows and deployed using automated pipelines. This codification of infrastructure solves several problems that arise with traditional infrastructure management, such as configuration drift, human error, lack of reproducibility, and limited auditability [17].

From a practical point of view, IaC tools allow businesses to achieve infrastructure immutability, which means that servers and other resources are never changed after they are deployed; instead, they are replaced with new instances when changes are needed. This immutable infrastructure pattern lowers the chance of inconsistencies between environments and makes troubleshooting easier by eliminating the need to track small changes over time [18].

The growth of cloud computing platforms, which make programmatic APIs for resource provisioning and management available, has accelerated the adoption of IaC. These APIs serve as the backbone of IaC tools, converting high-level descriptions of infrastructure into a series of API calls that realize the desired state. Cloud resources are especially well-suited to IaC approaches because of their elasticity and on-demand nature, which allows infrastructure to be quickly provisioned and deprovisioned in response to shifting needs.

Declarative and imperative are the two operational models into which IaC tools can be divided. Declarative IaC tools ask users to define the end state they want the infrastructure to be in and then let the tool determine what steps need to be taken to get there. This method focuses on the "what" rather than the "how," abstracting the procedural aspects of resource creation. Imperative IaC tools, on the other hand, provide users with more control over the provisioning process but also increase complexity by requiring explicit specification of the steps required to reach the desired state [1].

Several IaC tools have become the most popular options, each with unique features and applications. Terraform, created by HashiCorp, has emerged as one of the most popular multicloud IaC tools with a plug-in-based architecture that supports multiple cloud providers. Owing to its deep integration with the AWS ecosystem, AWS CloudFormation offers native IaC capabilities tailored to Amazon Web Services [19]. Originally designed as configuration management tools, Ansible, Puppet, and Chef have since expanded to include infrastructure provisioning features [1]. A more recent generation of IaC tools, Pulumi, enables the definition of infrastructure using general-purpose programming languages, as opposed to domain-specific ones [20].

Although IaC has many advantages, its adoption presents new challenges. IaC tools and their domain-specific languages can have a steep learning curve, especially for organizations moving away from more conventional infrastructure-management techniques. State management (tracking the current state of the deployed infrastructure) is a critical concern, as discrepancies between the actual infrastructure state and the state recorded by the IaC tool can lead to unintended consequences during updates. Because infrastructure codes may expose security misconfigurations that spread across multiple deployments or contain sensitive data, such as credentials, security considerations also change in IaC contexts [1].

Another major issue facing the IaC ecosystem is vendor lock-in. Although IaC codifies infrastructure definitions, which theoretically encourages portability, infrastructure code is frequently linked to particular cloud providers [2]. Migration efforts are made more difficult by dependencies created by provider-specific resource types, proprietary services, and distinct API behaviors. One of the core promises of cloud computing (freedom from vendor dependence) is at odds with this lock-in effect.

In addition, the diversity of IaC languages and tools leads to fragmentation of industries and organizations. Depending on their particular needs or past preferences, various teams may choose to implement different IaC solutions, which could result in redundant work, inconsistent procedures and knowledge silos. The lack of standardization makes sharing infrastructure patterns, switching between tools, and creating generic tooling that works with various IaC platforms more challenging.

These difficulties drive the pursuit of higher-level abstraction mechanisms that can offer standardized representations of infrastructure concepts, tool interoperability, and vendor independence. When businesses distribute workloads across several cloud providers and require uniform infrastructure management strategies, the need for these abstractions becomes even more pressing. Standards such as TOSCA, which have the potential to define infrastructure in a provider-agnostic manner while maintaining the advantages of the IaC paradigm, come into play here.

Terraform and AWS

Among the array of vendor-agnostic IaC tools, HashiCorp Terraform has emerged as the de facto industry standard [21]. Its provider-based architecture, which abstracts the APIs of different service providers, such as SaaS platforms, private clouds, and public clouds, into a single declarative syntax called the HashiCorp Configuration Language (HCL), is its main advantage. Developers and operators can specify the ideal end state of their infrastructure using this declarative approach, and Terraform decides what needs to be created, updated, or deleted to achieve it. In contrast, imperative approaches require the precise order in which commands must be executed.

A state file, a JSON document that keeps track of the managed infrastructure and its configuration, is the central component of Terraform functionality. Terraform can precisely plan updates, manage dependencies, and track changes using this state file, which serves as a map between the HCL code and real-world resources. A secure and reliable method for managing the infrastructure lifecycle is offered by the standard workflow, which uses the Terraform plan to preview changes and Terraform apply to carry them out.

The main technical context for the proof of concept in this study is the powerful combination of Terraform and Amazon Web Services (AWS). From basic components such as EC2 instances and VPC networks to more advanced services such as Lambda functions and Kubernetes clusters (EKS), Terraform's AWS provider is among the most extensive and well-established, providing coverage for hundreds of AWS services [22]. Teams can now codify intricate AWS multi-service architectures in a single, version-controlled repository.

Despite its strengths, an AWS Terraform setup is implementation-specific. HCL files are composed of resource blocks (such as `aws_instance` and `aws_s3_bucket`) that map directly to specific AWS services.

Consequently, the code and vendor's specific service offerings are tightly coupled. Therefore, even if high-level architectural components (such as a virtual server or object storage) are conceptually the same, moving such an architecture to a different cloud provider would necessitate a complete rewrite of the IaC files.

2.1.2 The TOSCA Standard

The Topology and Orchestration Specification for Cloud Applications (TOSCA) is an OASIS[1] standard is designed to provide a language-agnostic, portable, and interoperable framework for describing cloud applications and their underlying infrastructure. The goal of TOSCA is to create a vendor-neutral metamodel for expressing the structure, requirements, capabilities, and management aspects of complex distributed systems, in contrast to IaC tools that usually concentrate on particular providers or require provider-specific syntax. Because of this fundamental philosophical difference, TOSCA may be able to address the interoperability and portability issues that are present in modern IaC practices [23].

The understanding that cloud applications are complex topologies of interconnected components with complex dependencies, lifecycle management requirements, and operational constraints, rather than just infrastructure resources, is where TOSCA started. Whereas TOSCA takes a comprehensive approach that incorporates relationships, constraints, and orchestration logic into a single model, traditional infrastructure definitions frequently treat resources as separate entities. Compared to provider-specific IaC languages, this topology-centric viewpoint allows for higher-level abstraction when reasoning about application deployment.

Reusable node and relationship types can be defined using the type system at the core of TOSCA. Node types are the basic building blocks of applications, ranging from middleware components such as application servers and databases to infrastructure components such as virtual machines and storage volumes to application-specific services. Relationship types, such as "hosted on," "connects to," or "depends on" relationships, represent the connections and dependencies among nodes. Reusability is encouraged, and distinct taxonomies of infrastructure and application components

[1]`www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca`

are established because of the inheritance support of this type of system, which makes it possible to create specialized types that add new attributes or capabilities to base types.

The main artifact used to describe the deployments is the TOSCA service template. In addition to node templates (instances of node types), relationship templates (instances of relationship types), inputs and outputs for parameterization, and policies for expressing constraints and operational requirements, a service template encapsulates the topology of an application. TOSCA crucially separates the definition of the components that make up an application and the implementation or deployment of those components to particular platforms. Owing to this division of responsibilities, platform-specific orchestrators may be able to implement the same high-level TOSCA description across various cloud platforms [24].

Identified Limitations and the Motivation for Change

Although the research community has made significant progress in applying and extending TOSCA, numerous studies have highlighted persistent barriers to its widespread adoption. Empirical analyses and comparative evaluations consistently report that the TOSCA 1.x series suffers from substantial complexity, semantic inconsistencies, and a lack of mature tooling [25, 26, 17]. The low adoption rate observed among practitioners—estimated to involve less than a quarter of industry professionals—suggests a gap between the standard's theoretical strengths and its practical usability [17]. Comparative studies of orchestrators, such as Cloudify, xOpera, and IndigoDC, revealed that even the most advanced tools either extend the standard with custom DSLs or remain under active development, confirming the immaturity of the overall TOSCA tooling ecosystem [26]. Other studies have pointed out the proliferation of incompatible "TOSCA dialects," the verbosity of early XML-based models, and the need for simplified subsets, such as TOSCA Light, to bridge research models and production technologies [27, 28]. These limitations, echoed in both academic and industrial contexts, created a "chicken-and-egg" dynamic, where limited tooling hindered adoption, and low adoption discouraged further tool investment [28, 25, 27, 17]. This sustained critique ultimately motivated the redesign of the standard, leading to TOSCA 2.0, which emphasizes simplicity, extensibility, and practical interoperability.

The Evolution to TOSCA 2.0: A Pragmatic Redesign

By fixing many of the issues found in previous iterations and updating the specification to reflect modern cloud computing practices, the move to TOSCA 2.0 marked a substantial advancement in the standard. With the release of TOSCA 2.0 in July 2025 [24], significant modifications were made to increase its extensibility, consistency, and simplification [29]. One of the biggest changes was simplifying the syntax to remove superfluous constructions and lessen the ambiguity that had accumulated from the 1.x series' gradual evolution.

The type system was improved to offer more comprehensible semantics and better support for modern cloud-native patterns. The specification simplifies the expression and verification of component compatibility by introducing enhanced mechanisms for capability matching and requirement fulfillment. The relationship model was refined to better capture the subtleties of the dependencies in containerized and microservice-based deployments. Furthermore, TOSCA 2.0 gives more weight to intent-based modeling, recognizing that users should be able to articulate goals rather than prescribe implementation steps.

TOSCA profile mechanism of TOSCA 2.0 provides a way to define domain-specific extensions that tailor the base metamodel for particular technologies. Profiles can define conventions for network function virtualization, edge computing, or Kubernetes-based deployments while remaining compatible with the core standard. This modularity enables communities to extend TOSCA to meet their needs without fragmenting the ecosystem.

Another significant change in TOSCA 2.0 is the improved policy and governance model, which allows non-functional requirements, such as compliance, security, and cost optimization, to be expressed and verified declaratively.

Despite these advancements, TOSCA 2.0 introduced syntactic incompatibility with 1.3, creating migration challenges for existing tools and templates to be addressed. However, this also offers an opportunity to modernize the tooling and reduce the accumulated complexity.

The transition from TOSCA 1.3 to 2.0 reflects a broader evolution in cloud modeling thinking, from theoretical completeness to pragmatic implementability. By simplifying adoption and focusing on interoperability, TOSCA 2.0 has the potential to extend beyond research contexts and into mainstream cloud engineering.

Nevertheless, TOSCA remains less widespread than popular IaC tools such as Terraform, largely because of the maturity of provider-specific ecosystems and the lack of production-ready orchestrators. Overcoming these barriers requires new bridges between TOSCA and IaC, which this thesis addresses by proposing a systematic reverse engineering of IaC into TOSCA 2.0, leveraging existing infrastructure definitions to facilitate standard adoption.

2.1.3 Reverse Engineering

Reverse engineering is a systematic process for extracting design knowledge from an existing system or artifact without access to its original documentation [30]. In software engineering, it aims to recover high-level abstractions from low-level implementations, such as deriving architectural models from source code or reconstructing specifications from binaries. Its main motivation lies in understanding, maintaining, or integrating systems whose documentation is outdated, incomplete, or lost [31]. Thus, reverse engineering serves both as a means of software comprehension and as a foundation for reengineering or modernization activities.

Process and Techniques

The process typically involves two stages: (1) extraction, where system elements and their relations are identified using static or dynamic analysis [32], and (2) abstraction, where the extracted information is organized into higher-level models. Techniques vary from manual inspection to fully automated approaches based on parsing, pattern recognition, or program analysis algorithms [33]. The outcome is a representation that facilitates human understanding and supports subsequent transformations.

Reverse Engineering for Configuration and Infrastructure

With the advent of IaC, reverse engineering has been extended beyond traditional software artifacts to include infrastructure definitions and deployment specifications. IaC encodes infrastructure in machine-readable files that describe the desired state of cloud resources [17]. Reverse engineering such artifacts is challenging because IaC mixes declarative and imperative logic, embeds dependencies that are often implicit, and incorporates provider-specific semantics that must be interpreted during the analysis [34].

Deterministic and Heuristic Approaches

Reverse engineering techniques range from deterministic, rule-based mappings to heuristic or learning-based methods. Deterministic approaches define explicit and verifiable mappings between known constructs, offering precision and traceability. Heuristic methods use probabilistic reasoning or machine learning to infer semantic relationships that are not captured by formal rules. The two approaches represent a trade-off between reliability and automation, which is central to this study [35].

Reverse Engineering in Model-Driven Engineering

Model-Driven Engineering (MDE) provides a theoretical foundation for viewing reverse engineering as a model transformation problem. Software or infrastructure artifacts can be considered as instances of a source metamodel that are transformed into instances of a target metamodel. This framing enables formal reasoning about transformation properties, such as consistency and semantic preservation, while highlighting challenges such as metamodel heterogeneity and information loss between source and target models [36].

2.1.4 AI for Structured Code Generation

Recent advances in artificial intelligence (especially Large Language Models) have revolutionized automated code generation. Trained on large corpora of programming languages and documentation, these models can understand syntax, infer intent, and generate coherent and syntactically correct code from natural language prompts [37]. Transformer-based architectures with self-attention mechanisms can capture long-range dependencies, making them powerful tools for automating software and configuration tasks.

From Templates to Learning-Based Generation

Earlier code generation methods relied on templates and rule-based systems that required extensive domain engineering [38]. Neural sequence-to-sequence and transformer models have shifted this paradigm toward data-driven learning, where models learn code patterns directly from large repositories [39]. Modern models such as Codex[2] or Claude Code[3] demonstrate few-shot and cross-language capabilities, extending automation to complex programming and configuration tasks.

Structured and Domain-Specific Code Generation

Generating structured or domain-specific code—such as Terraform configurations or TOSCA templates—poses unique challenges [40]. These declarative languages impose rigid hierarchical syntax and strict schema constraints on the data. LLMs, which excel in flexible natural language generation, often struggle to maintain the following:

- Structural consistency, such as correct nesting and block closure.
- Semantic accuracy, such as valid property names and dependency relations.

Because Infrastructure-as-Code languages are underrepresented in the training data, models may hallucinate attributes or generate semantically invalid configurations [12].

[2]`https://openai.com/index/introducing-codex/`

[3]`https://claude.com/product/claude-code`

Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) addresses these issues by combining the generative power of LLMs with dynamic access to external knowledge sources [41]. At inference time, the model retrieves relevant documentation, schema definitions, or examples that are injected into the prompt to ground the generation process. For structured code, this allows the model to respect syntax rules, reuse provider definitions, and align with current standards, reducing hallucinations and improving correctness [41].

In this thesis, RAG is investigated as a preliminary study to assess the feasibility of using LLMs to translate Terraform configurations into TOSCA 2.0 models.

2.1.5 Intent-Based Orchestration

IaC marked a major step toward automation, it still requires human experts to specify the desired state explicitly. Intent-Based Orchestration (IBO) represents the next evolution in infrastructure management, enabling administrators to define high-level objectives rather than concrete configurations [42].

From IaC to Intent-Based Management

Building on concepts from Intent-Based Networking (IBN), IBO captures business-oriented goals—referred to as intents—and allows autonomous systems to translate them into actionable configurations [43]. In this paradigm, administrators specify outcomes (e.g., "maintain authentication latency below 100 ms") instead of deployment details (e.g., "provision two t3.micro instances"). The orchestration system interprets the intent, executes the necessary changes, and continuously monitors compliance, forming a closed feedback loop that can detect and correct "intent drift" caused by failures or performance degradation [6].

Requirements for Intent-Based Systems

IBO requires a formal, machine-readable model of system topology and behavior. An orchestrator cannot reason abstract goals, such as resilience or cost efficiency, from provider-specific IaC files alone. Semantically rich representations of components, their capabilities, and relationships are required to evaluate and modify architectures intelligently [42]. For instance, to improve availability, the system must recognize that a single web server is a bottleneck and that introducing a load balancer and replicas constitutes a valid redundancy.

Vendor-Neutral Models as Enablers

These requirements have driven interest in vendor-neutral modeling standards capable of describing infrastructures independently of specific providers. Among these, TOSCA 2.0 provides a promising foundation; its graph-based topology model and standardized semantics enable reasoning about architectural intent across heterogeneous environments. TOSCA's abstraction capabilities align closely with the needs of IBO systems, offering a potential bridge between declarative configurations and autonomous intent fulfillment, although realizing full intent-based orchestration remains an open research challenge [8].

2.2 Related Work

This section examines the state of the art in reverse engineering Infrastructure-as-Code toward vendor-neutral standards, and specifically toward TOSCA. As this represents a largely unexplored area, we broadened the review to encompass related domains: IaC translation and migration strategies, cloud service abstraction and portability, TOSCA-related research and tooling, and emerging AI-based approaches to infrastructure code analysis.

2.2.1 Reverse Engineering IaC to Vendor-Neutral Standards

To the best of our knowledge, no existing work directly addresses the systematic reverse engineering of provider-specific IaC implementations into vendor-neutral orchestration standards such as TOSCA. Although forward engineering approaches, in which abstract models are first defined and then compiled into provider-specific IaC, have been extensively explored, the reverse direction remains underexplored in the literature. The absence of research in this area is likely due to several factors. First, vendor-neutral standards such as TOSCA have historically been adopted in greenfield scenarios, where infrastructure is designed abstractly from the outset, rather than retrofit scenarios involving legacy IaC. Second, the complexity of semantic mappings between low-level, provider-specific resources and high-level, abstract types presents significant technical challenges that have deterred systematic investigations. Third, the recent introduction of TOSCA 2.0 means that the tooling and methodologies for this version of the standard are still maturing. Given this gap, we examined related work in adjacent domains that inform our approach to IaC reverse engineering toward TOSCA.

2.2.2 Reverse Discovery from Cloud Resources

Reverse discovery focuses on reconstructing Infrastructure-as-Code (IaC) definitions from existing cloud deployments by extracting metadata directly from provider APIs. Rather than translating between IaC dialects, these approaches regenerate declarative configurations that reflect the current state of the infrastructure.

Bhatia and Gabhane [9] formalize this process in the context of Terraform, describing how cloud inventories can serve as a "source of truth" for automatically generating configuration and state files. This enables organizations to onboard unmanaged infrastructure into Terraform with minimal manual effort, although the resulting code remains provider-specific and lacks semantic abstraction.

Several tools have been developed to operationalize this principle. Firefly [44], a commercial multi-cloud governance platform, scans AWS, Azure, and GCP environments to reconstruct IaC definitions. Former2[4] performs a similar task for AWS, exporting live resources into CloudFormation or Terraform templates. aztfexport[5] , maintained by Microsoft, provides equivalent functionality to Azure by generating Terraform configurations from deployed resources. `Terraformer`[6] developed by Google Cloud, generalizes the approach to multiple providers, including AWS, Azure, GCP, and Kubernetes, producing Terraform code and state representations.

[4]`https://github.com/iann0036/former2`

[5]`https://github.com/Azure/aztfexport`

[6]`https://github.com/GoogleCloudPlatform/terraformer`

While effective for documentation, onboarding, and compliance, these tools reproduce provider-specific constructs without capturing architectural intent or higher-level abstractions, limiting their usefulness for vendor-neutral modeling and standard-based representation, such as TOSCA 2.0.

2.2.3 Horizontal IaC Translation

Horizontal Infrastructure-as-Code (IaC) translation refers to the conversion between IaC technologies operating at the same abstraction level—typically across different providers or domain-specific languages (DSLs). These approaches aim to facilitate cloud migration and interoperability but generally focus on syntactic transformations rather than on semantic abstraction.

The open-source tool `cf2tf`[7] converts AWS CloudFormation templates (JSON/YAML) into Terraform's HCL format; however, differences in resource models and parameter semantics limit the conversion accuracy. Manual adaptation is almost always required to produce deployable configuration. For Azure, `Sato`[8] performs a direct translation from ARM templates to Terraform, easing migration efforts but suffering from similar gaps due to non-equivalent constructs and missing mappings.

Pulumi offers the most extensive official support for horizontal conversions. Its importers can transform CloudFormation, Terraform, ARM/Bicep, and Kubernetes YAML definitions into Pulumi programs written in general-purpose languages such as TypeScript, Python, Go, and C# [45]. Although this capability accelerates tool adoption, the generated programs preserve the underlying resource definitions one-to-one without inferring architectural intent or eliminating provider dependencies.

Overall, existing horizontal translation tools improve cross-platform adoption but are limited to structural mapping. They do not generalize or abstract the semantics of infrastructure components, leaving interoperability challenges unresolved in this regard. This motivates the pursuit of higher-level, vendor-neutral representations, such as TOSCA2.0, that capture both the structure and intent across heterogeneous cloud ecosystems.

[7] `https://github.com/DontShaveTheYak/cf2tf`

[8] `https://github.com/JamesWoolfenden/sato`

2.2.4 TOSCA-Oriented Orchestration and Tooling

While TOSCA has achieved limited mainstream adoption, several research efforts and industrial projects have explored its application in specific domains, providing valuable insights into its practical strengths and limitations.

Network Function Virtualization (NFV)

One of the most prominent domains for TOSCA adoption has been Network Function Virtualization (NFV) within the telecommunications industry [46]. The European Telecommunications Standards Institute (ETSI) adopted TOSCA as a foundational standard for describing the structure and lifecycle of Virtual Network Functions (VNFs) and network services [24, 5]. Researchers have leveraged the TOSCA topology model to define complex service chains, manage network resources, and automate deployment across distributed infrastructures [25, 5]. This context demonstrates the standard's strength in capturing dependencies and lifecycle operations that extend beyond traditional enterprise IaC capabilities.

Cloud Orchestration and Multi-Cloud Management

In the broader cloud computing landscape, research has focused on developing TOSCA-compliant orchestrators and management frameworks. Several industrial and academic projects have adopted TOSCA, including Cloudify and Openstack. Cloudify, an open-source orchestration platform, has been widely studied as a TOSCA-compliant orchestrator that supports declarative deployment blueprints. OpenStack-based initiatives, such as RADON, have also leveraged TOSCA templates to define cloud components and manage complex topologies [26, 47].

Academic contributions have introduced model-driven architectures for multi-cloud orchestration, using TOSCA as a unifying modeling language to abstract heterogeneous provider APIs [23, 25]. These studies address challenges such as policy enforcement, elastic scaling, and cost-aware placement optimization, demonstrating the expressive power of TOSCA service templates for managing distributed environments.

Formal Verification and Model Analysis

The formal structure of TOSCA has also inspired research in verification and validation. Scholars have proposed methods to formally analyze TOSCA templates and detect design-time inconsistencies or policy violations before deployment [25, 8]. In this line of work, TOSCA is treated not merely as a deployment script but as a formal system model, enabling automated reasoning and correctness checking, which are difficult to achieve with less structured IaC languages.

Collectively, these efforts illustrate the maturity of TOSCA as a forward engineering and orchestration standard. However, they also highlight a persistent gap: the absence of systematic approaches for *reverse-engineering* existing provider-specific IaC artifacts into TOSCA, particularly into the simplified and extensible TOSCA 2.0 standard.

Validation and Compilation Tools

The practical adoption of TOSCA requires robust tooling ecosystems for parsing, validation, and compilation. Puccini[9], a Go-based open-source compiler and processor, has emerged as one of the most prominent tools in the TOSCA community. Unlike simple validators or linters, Puccini performs a full compilation of TOSCA service templates, resolving imports, validating type hierarchies, inheriting properties, and transforming human-readable YAML definitions into a formal, structured representation. This comprehensive approach ensures that the processed TOSCA templates are both syntactically and semantically correct according to the standard requirements.

The central architectural component of Puccini is Clout (Cloud Orchestration and Unification Topology)[10], a language-agnostic, graph-based intermediate representation designed to capture the fully resolved topology of a service template. When Puccini compiles a TOSCA file, the resulting Clout artifact contains all nodes, relationships, properties, and workflows in a structured graph that is optimized for consumption by downstream tools. This design deliberately separates the complexity of parsing and validating the TOSCA grammar from the orchestration engine

[9] `www.github.com/tliron/go-puccini`

[10] `www.github.com/tliron/go-puccini/tree/main/clout`

logic. Consequently, orchestrators need only interpret the standardized Clout format to perform deployment and management tasks without requiring native TOSCA comprehension.

At the time of this research, Puccini supported TOSCA versions 1.x, with TOSCA 2.0 compatibility not yet implemented in the tool. This limitation reflects the nascent state of the tooling ecosystem following the recent ratification of the updated standard, creating a practical barrier for the validation and adoption of TOSCA 2.0 templates.

Collectively, these efforts illustrate the maturity of TOSCA as a forward engineering and orchestration standard. However, they also highlight a persistent gap: the absence of systematic approaches for reverse-engineering existing provider-specific IaC artifacts into TOSCA, particularly the simplified and extensible TOSCA2.0 standard.

2.2.5 Model-Driven Engineering for Configuration Analysis

Reverse engineering has also been explored within the Infrastructure-as-Code (IaC) ecosystem, primarily through the lens of model-driven engineering (MDE) and code quality analysis. Rather than reconstructing abstract models from provider-specific configurations, these studies focus on detecting design and security issues within existing IaC artifacts.

Rahman et al. [48] conducted one of the first large-scale empirical studies on security smells in IaC scripts. By analyzing over 15,000 Puppet files, they identified seven recurring patterns indicative of configuration weaknesses, such as hard-coded secrets, empty passwords, or insecure bindings, and implemented a static analysis tool (SLIC) to automatically detect them. Their findings highlight the prevalence and persistence of low-level configuration flaws, with some remaining unresolved for years, thus underscoring the need for higher-level reasoning regarding infrastructure quality and intent.

Kumara et al. [49] extended this line of work through a semantic, model-driven approach to smell detection in deployment models. Their framework represents TOSCA-based topologies as OWL 2 ontologies and applies SPARQL reasoning rules to identify structural or security antipatterns, such as Admin by Default, Weak

Cryptography, and Unrestricted IP bindings. This ontology-driven reasoning enables formal validation and extensibility across heterogeneous modeling languages, bridging software quality assurance and semantic interoperability.

Although both studies demonstrate how MDE and semantic reasoning can enhance IaC quality analysis, they remain diagnostic rather than transformative; they detect problematic configurations but do not reconstruct higher-level, vendor-neutral models. Consequently, the reverse engineering of IaC into interoperable representations, such as TOSCA2.0, remains an open challenge, which this thesis addresses through a systematic, model-based translation approach.

2.2.6 LLM-Based Approaches for IaC Automation

Recent studies have explored the use of Large Language Models (LLMs) to automate various stages of the Infrastructure-as-Code (IaC) lifecycle, including code generation, quality assurance, and repair. Benchmark initiatives such as *IaC-Eval* [**?**] systematically evaluate model performance across Terraform and Ansible tasks, revealing recurring limitations such as hallucinated resources, missing dependencies, and inconsistent provider semantics.

LLMs have been increasingly adopted for IaC generation and for refactoring. Low et al. [50] demonstrated the automatic correction of misconfigurations in Terraform templates through fine-tuned generative models, whereas Hong et al. [51] applied prompt-based reasoning to detect and explain IaC smells in Ansible scripts. Similarly, Toprani and Madisetti [**?**] proposed an LLM-driven workflow for vulnerability detection and remediation in infrastructure code, demonstrating how generative models can enhance automated compliance and resilience.

A complementary line of research focuses on IaC translation and semantic preservation. Du et al. [52] examined the challenge of maintaining structural and operational consistency in cross-language program translation, findings that extend directly to IaC DSLs, such as Terraform and CloudFormation. Empirical work also shows that LLMs often struggle to retain cross-resource relationships and up-to-date provider knowledge, leading to configuration drift or undeployable outputs [53, 54].

Overall, current LLM-based approaches improve automation and defect detection in provider-specific IaC but remain confined to performing forward engineering tasks. No prior work systematically addresses the reverse engineering of IaC into vendor-neutral metamodels such as TOSCA2.0, an open research gap that this thesis investigates through a hybrid rule-based and retrieval-augmented translation strategy.

2.2.7 Intent-Based Orchestration in Practice

While the concept of intent-based orchestration (IBO) has been widely discussed in the literature [42, 43, 6], practical implementations remain limited. A recent open-source initiative by Intel, the *Intent-Driven Orchestration (IDO) Planner* [7], demonstrates an early prototype of intent-driven automation applied to container orchestration. Unlike conventional orchestrators, such as Kubernetes, which rely on imperative resource specifications (e.g., number of CPUs or replicas), the IDO framework allows users to express high-level objectives, such as latency thresholds or availability targets, through declarative intent manifests. The system continuously monitors runtime metrics, evaluates compliance with service-level objectives (SLOs), and dynamically adjusts configuration policies (e.g., scaling and resource allocation) to satisfy the declared intents. Its architecture integrates a closed control loop comprising a *planner*, which decomposes goals into actionable policies, and a set of *actuators* that apply configuration changes at runtime. The framework adopts a plug-in-based design, supporting the extension of planning algorithms and resource models, and interoperates with existing Kubernetes-based schedulers and observability stacks, such as Istio or Linkerd.

Although the IDO Planner provides a concrete step toward operationalizing intent-based orchestration, it remains limited to performance-oriented objectives and to containerized workloads. Broader integration with standard modeling frameworks, such as TOSCA 2.0, and formal representations of system intent are still missing.

2.3 Research Gap

The preceding review of the literature reveals significant and parallel advancements in IaC, model-driven orchestration standards, and the application of Artificial Intelligence to code transformation. The IaC paradigm, led by tools such as Terraform, has matured to enable reliable and automated provisioning of complex cloud infrastructure. Concurrently, the TOSCA 2.0 standard offers a powerful vendor-agnostic solution to the fragmentation and vendor lock-in problems. However, a critical gap exists at the intersection of these domains: the practical and systematic transition from concrete, implementation-specific IaC to abstract and portable TOSCA models.

This research gap can be divided into three main categories based on the literature:

- **Absence of Techniques for Vertical Reverse Engineering:** Current research in cloud infrastructure analysis has mainly focused on horizontal translation, that is, the conversion between concrete IaC formats such as AWS CloudFormation and Terraform, or on the direct visualization of existing IaC files. These methods can be useful for specific migration tasks, but they do not address the underlying portability problem because vendor lock-in is preserved. A horizontal converter simply replaces one provider-specific syntax with another; for example, a CloudFormation to Terraform tool still generates AWS-specific Terraform code that cannot be reused with a different cloud provider.
  The literature lacks a methodology for vertical reverse engineering, an act of architectural recovery that raises abstraction levels. This process goes beyond simple syntactic conversion to enable true semantic portability. Instead of one-to-one translation, it abstracts concrete, provider-specific resource definitions (the *how*) into a standardized, provider-agnostic topology model in TOSCA 2.0 (the *what*). This recovered model can then be interpreted by different TOSCA-compliant orchestrators to deploy equivalent infrastructure on any target cloud. Currently, no formal framework exists for this transformation.
- **Unexplored Duality in Translation Methods:** Although deterministic, rule-based systems and contemporary AI-driven models are both employed separately for code transformation tasks, their comparative effectiveness for IaC-to-TOSCA reverse engineering remains unexplored. The inherent trade-offs between the pre-

cision and verifiability of rule-based approaches and the automation and semantic flexibility of AI-based methods (especially those enhanced with RAG) have not been systematically investigated. Understanding the strengths, limitations, and potential synergies of each strategy requires an empirical comparison within a unified framework.

- **Immature Tooling Ecosystem for TOSCA 2.0:** The TOSCA tooling ecosystem, particularly for the recently ratified TOSCA 2.0 standard, remains in its infancy. Although Puccini provides a robust foundation for TOSCA processing, its lack of TOSCA 2.0 support at the time of this study exemplifies the broader ecosystem gap. Comprehensive orchestrators, development environments, and integration tools that would facilitate practical TOSCA adoption are largely absent. Research contributions that advance both the theoretical understanding and practical tooling infrastructure are therefore especially valuable in this context.

By addressing these gaps, this research advances both the theoretical understanding of IaC reverse engineering and the practical capabilities of the TOSCA ecosystem. Specifically, this study proposes a systematic reverse engineering methodology for translating Terraform configurations into TOSCA 2.0-compliant artifacts, enabling the recovery of provider-agnostic architectural models from low-level, implementation-specific IaC definitions. This approach was validated using both rule-based and AI-assisted translation strategies, providing empirical insights into their respective trade-offs. Furthermore, the extension of the Puccini compiler to support TOSCA 2.0 contributes to the practical tooling infrastructure required for the broader adoption of the standard. These contributions establish a foundation for bridging the gap between contemporary IaC practices and the vision of standardized, portable, and intent-based cloud infrastructure management.

CHAPTER 3

Methodology

This chapter describes the research methodology employed to investigate the reverse engineering of Infrastructure-as-Code files into TOSCA 2.0. This study adopts Action Design Research (ADR) as its methodological framework, a design science approach that addresses practical problems while simultaneously contributing to theoretical knowledge [15]. ADR is particularly suited for this research because it facilitates the iterative development and evaluation of artifacts within authentic organizational contexts, ensuring that both practitioner needs and academic rigor are satisfied [55].

3.1 Research Prerequisites and Infrastructure

This research required the establishment of technical infrastructure capable of validating TOSCA 2.0 artifacts before the ADR cycle could commence. Specifically, a processor capable of parsing, validating, and compiling TOSCA 2.0 service templates was essential to ensure that generated artifacts conformed to the specification and could be verified programmatically.

3.1.1 Extension of the Puccini Compiler for TOSCA 2.0

As discussed in Section 2.2.4, Puccini is an established open-source TOSCA compiler that transforms service templates into Clout's intermediate representation. However, at the inception of this research, Puccini supported only TOSCA 1.x, while the recently ratified TOSCA 2.0 standard introduced significant syntactic and semantic changes that were incompatible with the existing implementation.

To address this tooling gap, a substantial engineering effort was undertaken to extend Puccini to support TOSCA 2.0. This extension constituted a foundational contribution that enabled the validation pipeline for all subsequent ADR cycles. The key modifications included the following:

- **Parser Updates:** Adaptation of the YAML parsing logic to handle TOSCA 2.0 syntax, including new keywords, simplified structure and removed deprecated constructs.
- **Type System Refactoring:** Implementation of the updated type hierarchy, capability matching semantics, and requirement fulfillment mechanisms introduced in TOSCA 2.0.
- **Validation Logic:** Integration of constraint validation rules conforming to the TOSCA 2.0 specification, ensuring that processed templates satisfy structural and Semantic correctness criteria.

The implementation was developed in Go, maintaining compatibility with Puccini's existing architecture and plugin system. The extended compiler was validated against the TOSCA 2.0 specification examples and the test cases provided by the OASIS Technical Committee. Upon completion, the implementation was contributed to the upstream Puccini project via pull request[1], where it underwent code review by the project maintainer and was Subsequently, they were merged into the main branch.

This contribution serves a dual purpose: (1) it provides the validation infrastructure required for this research, ensuring that all generated TOSCA 2.0 templates can be mechanically verified for correctness; and (2) it strengthens the broader TOSCA ecosystem by enabling the open-source community to process and validate TOSCA

[1]`tliron/go-puccini@33395a3`

2.0 artifacts, addressing one of the tooling gaps identified in Section 2.3.

Note on Ongoing Development: Following the contribution, the Puccini maintainer initiated a project to reimplement the compiler in Rust to improve its performance and maintainability. While this reimplementation is beyond the scope of this thesis, the Go-based TOSCA 2.0 support developed here provides a reference implementation and design insights that informs ongoing Rust-based development.

3.2 ADR Team

The composition of the Action Design Research team is fundamental to the methodology's success, as ADR explicitly requires collaboration between researchers and practitioners to ensure both theoretical relevance and practical applicability [15]. For this research, the team was deliberately structured to combine complementary expertise in cloud computing research, TOSCA standardization, and practical infrastructure orchestration, reflecting the dual nature of the investigation into both technical implementation and standard development.

The core ADR team consisted of three members who maintained continuous engagement throughout all the research stages. The master's student served as the primary researcher and artifact developer, responsible for the conceptualization, design, implementation, and evaluation of all translation frameworks developed across the three BIE cycles. This role encompassed both technical development work and the theoretical framing necessary to extract generalizable design principles from practical interventions.

The first supervisor, a professor at the Jheronimus Academy of Data Science and an active member of the OASIS TOSCA Technical Committee, provided a unique dual perspective that proved particularly valuable for the research. The supervisor's involvement in the TOSCA standardization process provided direct access to the evolution of the standard from version 1.3 to 2.0, including insights into the design rationale, governance constraints, and emerging requirements from the broader community. This connection ensured that the proposed reverse engineering approaches and intent-based orchestration extensions remained aligned with TOSCA's design philosophy and could potentially influence future standard development. Furthermore, the supervisor's academic expertise provided methodological oversight, ensuring that the research maintained scientific rigor and contributed meaningfully to both practical and theoretical knowledge.

The second supervisor, a PhD researcher at the Jheronimus Academy of Data Science specializing in cloud computing, provided technical guidance and actively participated in the problem formulation and artifact evaluation phases. The supervisor's expertise in cloud infrastructure and contemporary IaC technologies was essential for validating the technical soundness of the translation approaches and ensuring that the framework addressed authentic challenges faced by practitioners. The role included reviewing implementation decisions, suggesting architectural improvements, and contributing to the assessment of artifact maturity in each development cycle.

This relatively compact team structure, while smaller than some ADR studies involving multiple organizational stakeholders, proved advantageous for maintaining a focus on the standards-driven aspects of the research. The depth of expertise within the team compensated for its size, particularly given the specialized nature of TOSCA standardization and technical complexity of multi-IaC translation. The team's composition reflected the research's dual contribution targets: advancing academic knowledge of IaC reverse engineering while simultaneously strengthening real-world tooling and community-driven standards adoption.

3.3 Stage 1: Problem Formulation

Stage 1 is dedicated to problem formulation, which involves sufficient interactions with domain experts to understand the authentic challenges and requirements that the research must address [15]. The approach for this stage began with exploratory discussions to gain a foundational understanding of the current state of IaC-to-TOSCA translation practices, the limitations of existing approaches, and the strategic goals that motivated the research. Through systematic engagement with the supervisory team, perspectives on technical feasibility, architectural requirements, and practical constraints were captured to shape the artifact-development process.

Unlike traditional ADR studies conducted within a single organization with multiple internal stakeholders, this study addressed a problem that spanned multiple organizations and communities. The specialized nature of the domain, which requires expertise in both IaC technologies and TOSCA standardization, necessitated a focused approach to problem formulation. This research leveraged the concentrated expertise of the supervisory team, supplemented by an analysis of the existing literature, standards documentation, and community discussions within the IaC and TOSCA ecosystems.

3.3.1 Structured Expert Discussions

The problem formulation phase involved guided conversations with both supervisors, structured around key thematic areas designed to map the problem space comprehensively. These discussions served dual purposes: eliciting expert knowledge of existing challenges and collaboratively defining the scope and objectives of the research. Given the supervisors' complementary expertise—one in TOSCA standardization and governance and the other in cloud computing and IaC technologies—the conversations provided comprehensive coverage of both the technical and standards-related dimensions of the problem.

The discussions explored several interconnected areas that later informed the meta-requirements. The first area focused on understanding the current landscape of IaC portability challenges, including the practical implications of vendor lock-in, difficulties faced by organizations when migrating between cloud providers or IaC tools, and limitations of existing abstraction approaches. The second area examined the TOSCA standard itself, particularly the evolution from version 1.3 to 2.0, the design decisions underlying the key features, and the community's priorities for improving adoption and tooling. The third area addressed technical feasibility, exploring the semantic gaps between different IaC languages, the challenges of preserving intent during translation, and the trade-offs between automated and manual-mapping approaches.

These conversations were conducted iteratively over the initial phase of the research, with each session building on insights from the previous exchanges and preliminary technical explorations. The iterative nature of these discussions allowed for progressive refinement of the problem understanding, with emerging technical findings feeding back into theoretical considerations and vice-versa.

An important consideration throughout these discussions was ensuring that the problem formulation remained grounded in authentic practitioner needs rather than becoming purely theoretical in nature. The conversations explicitly referenced real-world scenarios drawn from professional experiences, case studies from TOSCA community discussions, and challenges documented by IaC practitioners in technical forums and industry reports. This grounding ensured that the research addressed problems of genuine practical significance while maintaining the flexibility to explore novel solution approaches.

3.3.2 Identifying Issues

The systematic reflection on these expert discussions, combined with analysis of technical literature and community resources, enabled the identification of key issues and requirements that would drive the research. This analytical process organized the emerging themes into coherent categories that captured both the current limitations of IaC translation approaches and the desired capabilities of an effective solution.

Several high-level issue categories emerged from this synthesis study. The first category addressed semantic preservation, encompassing challenges related to accurately capturing the intent and behavior specified in source IaC files when translating them into TOSCA representations. The second category focuses on extensibility and multi-IaC support, recognizing that a robust solution must accommodate diverse IaC languages beyond the initial implementation. The third category concerns vendor abstraction, identifying the tension between preserving provider-specific details and achieving meaningful portability across cloud platforms. Additional categories addressed maintainability challenges, the need for validation and verification mechanisms, and the requirements for community adoption and standard compliance.

To substantiate the identified issues and validate their significance, this analysis integrated the findings from the literature review conducted in parallel with the problem formulation. Academic literature on IaC challenges, industry reports on multicloud strategies, and technical documentation from TOSCA and IaC tool communities provide additional evidence of the prevalence and impact of the identified issues [2, 4]. This triangulation between expert knowledge, literature evidence, and community discussions ensured that the problem formulation reflected genuine research gaps, rather than isolated concerns.

The outcome of this identification process was a structured characterization of issues that would be formalized into meta-requirements in Chapter 4. These issues describe both the current state—the limitations and challenges practitioners face with existing approaches—and the desired future state—the capabilities and qualities that an effective IaC-to-TOSCA translation framework should possess. This gap between the current and desired states provided the foundation for defining the research objectives and guiding iterative artifact development through subsequent BIE cycles.

3.4 Stage 2: Building, Intervention, and Evaluation

Stage 2 revolves around building the artifact, executing interventions, and evaluating these changes through an iterative process. This stage implements the core cyclical nature of ADR, where requirements and solutions are progressively refined based on ongoing evaluations and reflections. Three main principles guided this stage of the research. First, the interconnection between technical artifacts and their operational context was recognized, particularly how TOSCA standardization intersects with real-world cloud infrastructure. Second, valuing the mutual influence between researchers and practitioners, where domain expertise from supervisors informed technical decisions, while research brought theoretical frameworks and systematic evaluation approaches. Third, evaluation is not treated as a separate final step but as a continuous process integrated throughout design, implementation, and refinement activities [15].

The research implemented three distinct BIE cycles, each addressing progressively more complex aspects of the IaC-to-TOSCA translation challenges. Cycle 1 focused on establishing a deterministic baseline to validate technical feasibility. Cycle 2 introduces architectural modularity to enable extensibility across multiple IaC technologies. Cycle 3 explored AI-augmented translation approaches to investigate the frontier of automation while maintaining precision.

This progression reflects the natural evolution from proof-of-concept to production-ready architecture to experimental innovation.

3.4.1 Requirements and Design Principles

The first step in initiating the BIE cycles was translating the issues and goals identified during problem formulation into actionable requirements. The systematic analysis conducted in Stage 1 produced a structured characterization of the challenges and desired capabilities, which were organized into broader meta-requirements (MRs). These meta-requirements capture the high-level qualities that the artifact must possess to effectively address the identified problems, such as semantic preservation during translation, extensibility to multiple IaC languages, and abstraction from vendor-specific implementations.

Meta-requirements were then instantiated as Design Principles (DPs), which provide prescriptive guidance on how to achieve the desired qualities. The formulation of the design principles drew on multiple sources: the meta-requirements themselves, theoretical frameworks from the literature review on software architecture patterns and translation systems, and best practices documented in related work on IaC abstraction and model-driven engineering. Each design principle articulates a specific architectural or implementation strategy that contributes to fulfilling one or more meta-requirements.

The relationship between meta-requirements and design principles was not one-to-one but rather many-to-many, with some design principles supporting multiple meta-requirements and some meta-requirements requiring multiple design principles for their full realization. This mapping was documented systematically to ensure traceability and to evaluate how well each design principle contributed to the overall artifact quality. The complete formulation of the meta-requirements and design principles is presented in Chapter 4, where their theoretical grounding and practical implications are discussed in detail.

To realize intermediate versions of the artifact across the three cycles, specific technical interventions were defined based on design principles. Each intervention represents a concrete implementation step that moves the artifact closer to fulfilling its design goals. By implementing these incremental improvements and evaluating their impact, the validity and relevance of the design principles can be assessed empirically. The cyclical nature of the BIE process allowed for continuous refinement: insights from one cycle informed the priorities for the next, and design principles could be adjusted based on observed outcomes.

3.4.2 Interventions

In each BIE cycle, interventions were designed to instantiate specific design principles in the evolving artifact. The selection and prioritization of interventions balanced several considerations: technical dependencies between components, the value each intervention would provide for demonstrating feasibility, and the resources required for implementation. Across the three cycles, the research aimed to produce at least one proof-of-concept implementation for each design principle, ensuring a comprehensive evaluation of the complete design framework.

In each BIE cycle, interventions were designed to instantiate specific design principles within the evolving artifact. The selection and prioritization of these interventions was a crucial step governed by a strategy that balanced three key factors:

1. **Dependency Ordering:** Foundational interventions were always prioritized first. For example, implementing parsers for the source IaC syntax was a necessary prerequisite before the mapping logic could be developed.
2. **Risk Reduction:** Interventions that addressed the most significant technical challenges or uncertainties were tackled early in the process. This "fail-fast" approach ensured that the core feasibility of the translation was validated upfront, minimizing the risk of discovering insurmountable obstacles late in the research process.
3. **Demonstrable Value:** Each cycle was structured to produce tangible, working examples as quickly as possible. This was essential for enabling stakeholder evaluation and providing a concrete baseline to measure the progress of subsequent interventions.

This prioritization strategy ensured that each BIE cycle began with essential foundational interventions that enabled subsequent, more complex refinements. Across the three cycles, this study aimed to produce at least one proof-of-concept for each design principle, allowing for a comprehensive evaluation of the entire design framework.

In Cycle 1, the interventions focused on establishing a foundational translation mechanism. This included implementing parsers for Terraform's HCL syntax, developing mapping logic to translate resource definitions into TOSCA node templates, extracting relationships between infrastructure components, and defining initial TOSCA type hierarchies for AWS resources. These interventions demonstrated technical feasibility and produced a working baseline against which subsequent improvements can be measured. The Cycle 1 artifact served as both a validation of the core translation concept and a foundation for architectural evolution.

Cycle 2 shifted the focus toward architectural modularity and extensibility. Interventions in this cycle refactored the monolithic Cycle 1 implementation into a plug-in-based architecture, developed a core translation framework with well-defined extension points, and created prototype plug-in interfaces for other IaC technologies to validate the architectural pattern, although full parsing and translation logic was implemented only for Terraform. These prototype plug-ins served primarily to validate the plug-in contract design and identify the necessary extension points. These changes transformed the proof of concept into an extensible platform capable of supporting diverse IaC sources.

Cycle 3 explored the feasibility of AI-augmented translation through experimental interventions conducted primarily in Jupyter Notebook environments. This exploratory approach prioritized rapid prototyping and technical validation over production-ready implementation, allowing the investigation of RAG architecture patterns, prompt engineering strategies, and validation mechanisms to ensure the correctness of AI-assisted translations. These interventions were implemented and evaluated primarily in Jupyter notebook environments, allowing rapid experimentation without disrupting the stable Cycle 2 architecture.

Throughout all cycles, the evaluation was integrated continuously rather than deferred to cycle completion. Technical validation was performed using test suites covering diverse IaC scenarios, correctness verification by comparing translations against expected TOSCA outputs, and architectural review by examining code quality and maintainability. This ongoing evaluation informed mid-cycle adjustments and helped prioritize the remaining interventions as each cycle progressed.

3.4.3 Evaluation Approach

At the conclusion of each BIE cycle, systematic evaluation sessions were conducted with the supervisory team to assess the artifact's evolution and validate design principle effectiveness. These evaluation sessions examined the artifacts from multiple perspectives, ensuring a comprehensive assessment of both technical qualities and alignment with research objectives. The evaluation framework was based on established constructs from design science and information systems research. Several dimensions were considered for each artifact iteration.

- **Technical Correctness** assessed whether translations accurately preserved the semantics of source IaC files, whether generated TOSCA specifications were syntactically valid according to the 2.0 standard, and whether the artifact handled edge cases and complex scenarios appropriately. This dimension was evaluated through: (1) manual semantic comparison between source Terraform configurations and generated TOSCA templates to verify that all infrastructure components, properties, and relationships were preserved; (2) Puccini compilation validation to ensure syntactic correctness and TOSCA 2.0 compliance—templates that failed Puccini parsing indicated translation errors requiring correction. A diverse set of Terraform configurations spanning compute, networking, storage, and database resources served as test cases, deliberately including both common patterns and edge cases (e.g., cross-resource references, conditional logic, and complex dependency chains).
- **Architectural Quality** examined whether the artifact's design supported the extensibility and maintainability goals articulated in the meta-requirements. For Cycles 2 and 3, this included evaluating the effectiveness of the plug-in architecture, clarity of abstraction boundaries, and ease with which new IaC technologies or cloud providers could be integrated into the system. The evaluation was conducted through architectural review sessions, where the code structure, module

coupling, and extension mechanisms were examined against established software architecture principles. The modular refactoring of Cycle 2 was validated through architectural review and design pattern analysis. The refactored Terraform plugin successfully conformed to the new generic interfaces without requiring core framework modifications, demonstrating that the architecture could theoretically support additional IaC technologies through the same plug-in contract.

- **Practical Utility** considered whether the artifact addressed authentic problems in ways that would be valuable to practitioners. This included assessing the completeness of IaC language coverage, the usefulness of the generated TOSCA specifications for real orchestration scenarios, and the feasibility of integrating the artifact into existing infrastructure workflows. Discussion with supervisors drew on their professional experience to ground these assessments in realistic use cases, evaluating whether the translated TOSCA templates would enable actual multicloud migration scenarios or intent-based orchestration applications.
- **Research Contribution** evaluated how well each artifact iteration advanced knowledge about IaC reverse engineering and contributed to the broader goals of the dissertation. This included assessing the new insights each cycle produced about translation challenges, the patterns that emerged as effective or ineffective, and how the findings could be generalized beyond the specific implementations.

Architectural Quality examined whether the artifact design supported the extensibility and maintainability goals articulated in the meta-requirements. For Cycles 2 and 3, this included evaluating the effectiveness of the plug-in architecture, clarity of abstraction boundaries, and ease with which new IaC technologies or cloud providers could be integrated. Evaluation was conducted through architectural review sessions where code structure, module coupling, and extension mechanisms were examined against established software architecture principles. Cycle 2 modular refactoring was validated by successfully implementing plugins for multiple IaC technologies (e.g., Ansible), demonstrating that the architecture genuinely supported extension without core framework modifications. Practical Utility considered whether the artifact addressed authentic problems in ways that would be valuable to practitioners. This included assessing the completeness of IaC language coverage, the usefulness of generated TOSCA specifications for real orchestration scenarios, and the feasibility of integrating the artifact into existing infrastructure workflows. Discussion with supervisors drew on their professional experience to ground these assessments in realistic use cases, evaluating whether the translated TOSCA templates would enable actual multi-cloud migration scenarios or intent-based orchestration applications. Research Contribution evaluated how well each artifact iteration advanced knowledge of IaC reverse engineering and contributed to the broader goals of the dissertation. This included assessing what new insights each cycle produced about translation challenges, what patterns emerged as effective or ineffective, and how findings could be generalized beyond the specific implementations.

While this section outlines the overall structure and rationale of the three BIE cycles, the specific interventions, implementation details, and evaluations of each cycle are discussed in detail in subsequent chapters. This mirrors the iterative nature of ADR, where each cycle builds upon the previous one to progressively refine artifacts and validate the corresponding design principles.

3.5 Stage 3: Reflection and Learning

Stage 3 focuses on reflection and learning. It examines the design/redesign work of Stage 2 and evaluates whether the artifacts and interventions align with the conclusions from Stage 1. Guided emergence holds that artifacts evolve not only through deliberate design choices but also through their use in authentic contexts, stakeholder feedback, and insights from implementation and evaluation [15].

Within each cycle, continuous reflection on implementation issues informed the mid-cycle adjustments. In Cycle 1, we initially planned to manually parse Terraform HCL to extract resource definitions; however, early implementation showed that Terraform's dynamic features—variable interpolation, conditional resources, and module expansion—require executing Terraform's evaluation logic rather than merely parsing the syntax.

Consequently, we chose to consume `terraform plan -json`, which provides fully evaluated configuration. This trades source-level detail for semantic accuracy: the translated TOSCA captures the intended deployment state and correctly handles complex Terraform constructs that would be impractical to parse manually. This decision illustrates guided emergence: technical realities shaped the architecture more than a purely theoretical design.

In Cycle 2, modular refactoring was prioritized; the resulting plug-in interfaces, abstraction layers, and extension points were designed so that future IaC integrations could consume evaluated configurations when available.

Between cycles, structured reflection sessions reviewed how well the design principles satisfied the meta-requirements, which technical approaches performed better or worse than expected, and what new insights emerged regarding IaC-to-TOSCA translation. This included reassessing the validity and completeness of the initial meta-requirements and the design principles. For example, vendor abstraction proved to be more complex than anticipated, revealing trade-offs between generalization and information preservation that became apparent only during implementation.

Finally, the comparative findings for RQ3 are qualitative: the deterministic pipeline underwent several refinement cycles with extensive test coverage, whereas the AI-augmented path remained an exploratory prototype. This asymmetry precludes like-for-like performance comparisons but surfaces core architectural and operational trade-offs—verifiability vs. flexibility, maintenance overhead vs. adaptation capability, and deterministic guarantees vs. semantic inference potential. We distilled these observations into an initial evaluation framework that can guide more rigorous comparative studies.

Overall, the reflections in Stage 3 contribute to answering RQ1 (deterministic Terraform-to-TOSCA mapping principles), RQ2 (automation potential of LLMs with RAG), and RQ3 (precision–automation tradeoffs).

3.6 Stage 4: Formalization of Learning

Stage 4 is dedicated to the formalization of learning that occurred throughout the research. The principle that guides this stage is that of generalized outcomes, which emphasizes the transition from situational knowledge embedded in specific implementations to generalizable design knowledge that can inform future research and practice [15]. While the ADR's emphasis on authentic contexts and concrete artifacts produces rich situated insights, this final stage requires stepping back to extract broader patterns and principles that transcend the specific technical choices and implementation details of this research.

The formalization process in this study involved several interconnected activities. First, the meta-requirements and design principles formulated in Chapter 4 were reconsidered in light of the experiences and evaluations of all three BIE cycles. This reconsideration examined which meta-requirements proved most critical for achieving effective IaC-to-TOSCA translation, which design principles most successfully instantiated those requirements, and where gaps or refinements emerged during practical implementation. The revised understanding of these foundational constructs represents a key contribution to the generalizable design knowledge.

Second, the maturity of the final artifact was assessed through a structured evaluation considering technical completeness, production readiness, maintainability, and extensibility—qualities that determine whether the artifact could transition from a research prototype to practical tooling.

Third, broader lessons extracted from the research process were systematically reflected. These lessons encompass methodological insights into conducting ADR in standards-driven research contexts, technical insights into the fundamental challenges of semantic translation between IaC languages and orchestration standards, and practical insights into the trade-offs inherent in different translation paradigms. Particular attention was given to the comparative findings regarding deterministic versus AI-augmented approaches, as they represent a central tension in contemporary software engineering research.

Finally, the research process was scrutinized to identify potential threats to the validity of the obtained design knowledge by examining four dimensions [56].

Internal validity concerns whether observed effects genuinely resulted from interventions. The iterative ADR approach with continuous evaluation strengthens this, although the small team size and researcher proximity to artifact development introduce potential evaluation bias.

External validity addresses generalizability beyond the specific context examined. Key limitations include the following: the focus on Terraform and AWS (although the architecture supports extension to other IaC tools and providers); reliance on LocalStack for deployment validation, which validates syntactic correctness and basic provisioning but cannot guarantee semantic preservation for production AWS deployments owing to emulation fidelity limitations; and test case coverage representing common patterns but not comprehensive edge cases or complex production scenarios.

Construct validity evaluates whether meta-requirements and design principles capture essential qualities for effective translation. These were derived from expert discussions and literature but were not validated through broader practitioner engagement, potentially limiting their comprehensiveness for operational contexts.

Conclusion validity assesses whether inferences from evaluation data are warranted. The primarily qualitative evaluation approach, while strengthened by objective Puccini compilation validation, relied on manual semantic comparison without quantitative metrics, thus introducing subjectivity into correctness claims.

The complete formalization of learning is presented in Chapter 10 (Discussion), where the refined meta-requirements and design principles are articulated, artifact maturity is analyzed, lessons learned are synthesized, and threats to validity are discussed. Chapter 11 (Conclusions) builds on this formalization to answer the research questions comprehensively, articulate the research contributions to both theory and practice, acknowledge limitations, and provide recommendations for future work in IaC reverse engineering and TOSCA-based orchestration.

CHAPTER 4

Formulation of Problem and Design Principles

This chapter initiates the core of the Action Design Research cycle by formally defining the problem space and deriving the principles that will guide the artifact's construction. The problem formulation phase begins by grounding the research in tangible challenges in the contemporary cloud computing landscape. By synthesizing insights from the structured expert discussions and the literature review detailed in Chapter 2, this chapter first identifies the systemic issues within the Infrastructure-as-Code (IaC) ecosystem. These issues are then abstracted into a set of formal Meta-Requirements (MRs), which are translated into actionable Design Principles (DPs) that form the theoretical and architectural foundation of this thesis.

4.1 Identifying Challenges in the IaC Ecosystem

The IaC paradigm has transformed how infrastructure is provisioned, configured, and managed. By encoding infrastructure as declarative/imperative artifacts, IaC enables version control, reproducibility, and automation at scale [57]. Despite these advantages, the current IaC landscape hinders portability, interoperability, and vendor neutrality owing to fragmentation across tools and provider-bound abstractions.

4.1.1 Fragmentation of the IaC Landscape

The proliferation of provider-specific IaC tools and heterogeneous domain-specific languages has been extensively documented as a fundamental barrier to infrastructure portability (Section 2.1.1).

Guerriero et al. [17] identified "Too Polyglot IaC" as a recurring industrial challenge, where organizations maintaining multiple technologies—CloudFormation, ARM/Bicep, Terraform—face duplicated logic, fragmented knowledge bases and incompatible operational assumptions. The consequences include portability barriers that necessitate per-provider rewrites [4], knowledge silos that slow multi-cloud initiatives, and maintenance overhead from duplicated definitions across heterogeneous tools.

4.1.2 The Vendor Lock-In Dilemma

Vendor lock-in in IaC operates at both syntactic and semantic levels, as extensively discussed in Section 2.2.3. While provider-native tools create explicit dependencies, even "agnostic" tools like Terraform exhibit semantic lock-in through provider-specific resource schemas. Opara-Martins et al. [4] demonstrated that infrastructure code becomes a repository of provider-specific knowledge, where high-level intent—such as "deploy a serverless function with HTTP trigger"—is buried beneath incompatible properties, defaults and constraints across providers [29].

4.1.3 Interoperability and Integration Challenges

Modern enterprises rarely rely on a single IaC tool, as documented in Section 2.1.1. Heterogeneous landscapes typically combine Terraform, Ansible, Kubernetes manifests, and manual configuration, each maintaining independent state models, and dependency graphs. Guerriero et al. [17] found that cross-tool orchestration requires ad hoc glue code or proprietary integration layers, adding fragility and impeding the reuse of high-level architectural patterns across different technologies.

4.1.4 The TOSCA Adoption Paradox (Chicken-and-Egg)

As detailed in Section 2.1.2, TOSCA offers a vendor-neutral modeling language for describing infrastructure topologies at higher abstraction, decoupling architectural intent from provider-specific implementations [24, 5]. However, its adoption faces a chicken-and-egg dynamic: limited production-ready tooling discourages organizational migration, while low adoption rates suppress ecosystem investment in orchestrators and developer tools [25, 26] (Section 2.2.4). This paradox is compounded by substantial sunk costs in existing IaC workflows and network effects favor incumbent technologies.

Systematic reverse engineering could bootstrap TOSCA adoption by preserving existing IaC investments, while enabling gradual migration.

4.2 Derivation of Key Issues from Expert Discussions

Building on Section 4.1, we integrate (i) expert discussions, (ii) TOSCA 2.0 and Terraform/HCL documentation, and (iii) a focused literature scan to surface cross-cutting issues shaping the IaC→TOSCA problem space. Purely syntax-driven translation proves inadequate; challenges are primarily semantic, architectural, and verification-oriented, motivating the meta-requirements in 4.3.

4.2.1 Semantic Intent Preservation

Effective translation must go beyond syntax to recover architectural intent. In Terraform, resource blocks encode operational specifics (e.g., AMI, instance type, networking), while the "why" remains implicit. For example:

```
resource "aws_instance" "web" {
    ami = "ami-..."
    instance_type = "t3.medium"
    subnet_id = aws_subnet.public.id
}
```

Mapping `aws_instance` to a generic `tosca.nodes.Compute` captures the structure but risks semantic loss unless relationships, roles, and policies are reconstructed. TOSCA 2.0 explicitly separates what (topology, capabilities) from how (implementation variability), encouraging type systems that reflect roles and behaviors [24]. Thus, preserving functional/architectural/non-functional/operational intent requires deterministic rules plus, where needed, LLM/RAG-assisted inference with provenance and guardrails.

4.2.2 Extensibility Across IaC Paradigms

The framework must accommodate diverse IaC paradigms—declarative state-based (Terraform, Pulumi), templated (CloudFormation, ARM), procedural (Ansible/imperative Pulumi), and policy-as-code (Kubernetes CRDs, Crossplane). A Terraform-shaped core is brittle. The literature on model-driven engineering and DSLs supports a layered design: (i) generic transformation logic (graph analysis, type/relationship inference), (ii) language-specific adapters to normalize source IaC into an IR, and (iii) target generators for TOSCA 2.0 emission [58]. Adding a new IaC language primarily means implementing a new adapter, leaving the core intact.

4.2.3 The Abstraction–Precision Trade-off

Portability demands abstraction, yet deployability and fidelity often require provider-specific detail. Over-abstraction risks a "lowest common denominator" while over-specificity reintroduces lock-in. Expert feedback converged on layered abstraction: a high-level intent layer (patterns and goals), cloud-agnostic implementation layer (common primitives), and provider-specific extension layer (optional details/artifacts). This allows orchestrators and users to consume the model at the appropriate level without discarding valuable semantics.

4.2.4 TOSCA 2.0 Toolchain Maturity Gap

While TOSCA 2.0 advances the type system, policies, and workflows, production-grade orchestration support remains comparatively limited compared to 1.3, as noted in expert feedback. This imposes three constraints: (i) the translator must include syntactic and semantic validation rather than relying solely on orchestrators; (ii) feature targeting must be pragmatic, focusing on the most likely to be supported; and (iii) interoperability requires strict adherence to the spec. In this context, Puccini (as a TOSCA processor that compiles templates to graphs) provides a practical validation path, and extending it toward 2.0 semantics helps close the ecosystem gap.

4.2.5 Heterogeneous Mapping Strategies

Some correspondences—e.g., Terraform `depends_on` or reference edges mapping to TOSCA relationships—are rule-friendly. Many others are context-sensitive: the same `aws_instance` may act as a bastion host or as an application server; composite patterns such as Auto Scaling Group (ASG) + Launch Template (LT) + Application Load Balancer (ALB) + Target Group (TG) require pattern recognition; and best practices or compliance intents depend on external knowledge. In the broader literature, these observations motivate hybrid strategies that combine deterministic rules with AI-assisted inference [12].

In this study, we did not merge strategies within a single pipeline. We implemented two separate pipelines: a deterministic rule-based translator and a RAG-augmented translator, validated against the same TOSCA 2.0 checks to enable a controlled comparison of precision, coverage, explainability, reproducibility, and reviewer effort.

4.2.6 Synthesis: Issues Informing Requirements

Across evidence sources, five issues consistently shape the solution space: semantic intent preservation, extensibility by design, abstraction–precision balance, toolchain-aware verifiability, and heterogeneous mapping strategies. These are interdependent facets of the central aim: systematically reverse-engineering provider-specific IaC into vendor-neutral TOSCA without eroding meaning or maintainability. The next section distills these issues into meta-requirements, providing a traceable bridge from problem identification to design guidance.

4.3 Meta-Requirements

Following the identification of systemic challenges within the IaC ecosystem, this section formalizes a set of Meta-Requirements (MRs). In Action Design Research, meta-requirements represent the high-level, essential qualities and capabilities that the designed artifact must possess to be considered a successful solution to the identified problems. They are derived directly from the problem formulation and serve as stable guiding objectives throughout the iterative design and evaluation cycles. These MRs define what the framework must achieve, providing the criteria against which the subsequent Design Principles and final implementation will be validated.

4.3.1 MR1 - Semantic Fidelity and Intent Preservation

Description. The translation framework must preserve the original architectural intent expressed in the source IaC files. This requires moving beyond a simple one-to-one syntactic conversion of resources to accurately reconstruct the semantic relationships, roles, and dependencies among infrastructure components.

Rationale. As identified in Section 4.2.1, the core value of an IaC file lies not in its syntax but in the architectural design decisions it represents. A direct translation of an `aws_instance` resource into a similarly named TOSCA node would simply be a transliteration, failing to capture the intent that this resource functions as a "web server" or a "database host." This MR is critical for combating the vendor lock-in dilemma (Section 4.1.2) at the semantic level, ensuring that the resulting TOSCA model is a meaningful architectural abstraction and not merely a reformatted provider-specific file.

Verification Criteria.

- The generated TOSCA topology graph must accurately reflect the dependency graph of the source IaC.
- TOSCA node types in the output must represent architectural roles (e.g., `Compute`, `Database`) rather than provider-specific resource names.

- Relationships between nodes must capture both explicit and implicit dependencies from the source IaC.

4.3.2 MR2 - Extensibility and Maintainability

Description. The architecture of the framework must be inherently modular and technology-agnostic at its core, providing well-defined extension points for the future addition of new IaC technologies (e.g., Ansible, CloudFormation, Bicep) or cloud providers (e.g., Azure, GCP). While the proof-of-concept implementation focuses on Terraform and AWS, the architecture must demonstrate extensibility readiness through a clear separation of concerns and plugin-based design patterns.

Rationale. The fragmentation of the IaC landscape (Section 4.1.1) is the primary motivation for this research. A solution tightly coupled to Terraform would be a short-sighted tactical tool rather than a strategic generalizable framework. To address the challenge of "Too Polyglot IaC" and prevent the creation of yet another knowledge silo, the architecture must enforce a clear separation between the core transformation logic and technology-specific adapters, as motivated in Section 4.2.2.

Verification Criteria.
- The core architectural components of the framework must contain no direct dependencies on any specific IaC tool or cloud provider's libraries.
- The architecture must provide documented extension points (interfaces, abstract classes, or plugin specifications) that enable adding support for new IaC technologies without altering the core translation engine. Extensibility is validated through architectural reviews and design pattern analyses.
- The design should theoretically support processing inputs from multiple, heterogeneous IaC sources in a unified pipeline.

4.3.3 MR3 - Verifiability and Specification Compliance

Description. All artifacts generated by the framework must be syntactically correct and semantically compliant with the target TOSCA 2.0 specification. The framework itself must serve as the primary guarantor of correctness, providing a high degree of confidence in its output.

Rationale. The "TOSCA Adoption Paradox" (Section 4.1.4) is exacerbated by a perceived maturity gap in the toolchain (Section 4.2.4). To build trust and facilitate adoption, the translation framework cannot rely on downstream orchestrators to detect errors. By producing verifiably correct and standard-compliant TOSCA, the artifact not only becomes more reliable but also contributes positively to the ecosystem by providing a source of well-formed templates. This aligns with the practical contribution made to the Puccini project, which underscores the importance of a robust spec-compliant tooling ecosystem.

Verification Criteria.

- Generated TOSCA files must be valid YAML documents.
- The generated files must pass validation against the official TOSCA 2.0 schema, including the Simple Profile.
- The output must be successfully parsed and compiled by a reference TOSCA processor, such as Puccini.

4.3.4 MR4 - Abstraction–Specificity Balance

Description. While the primary goal is to abstract to a vendor-neutral model, the translation process must be lossless. The framework must provide a structured, explicit mechanism to retain all provider-specific attributes and configuration details that do not have a direct semantic equivalent in the standard TOSCA model.

Rationale. This requirement directly resolves the "Abstraction–Precision Trade-off" (Section 4.2.3). A purely abstract TOSCA model risks being undeployable because critical provider-specific details (e.g., a specific AWS AMI ID, Azure VM size, or performance-tier setting) are lost. By preserving this information within a standardized metadata block, the framework can produce an artifact that is both a high-level architectural model and a high-fidelity record of the original implementation, enabling portability and platform-specific optimization.

Verification Criteria.

- Every attribute from a source IaC resource must be present in the generated TOSCA model, either as a mapped TOSCA property or within a structured metadata block.
- The translation process should be theoretically reversible, meaning the original IaC file could be reconstructed from the combination of the TOSCA topology and its associated metadata.

4.3.5 Summary of Meta-Requirements

Table 4.1 provides a consolidated overview of these meta-requirements, their core focus, and their primary motivation derived from the problem space analysis. These meta-requirements collectively define the success criteria for the artifact and provide the foundation upon which the Design Principles, detailed in the following section, are built.

ID	Meta-Requirement	Core Focus	Primary Motivation	Mapped Challenge(s)
MR1	Semantic Fidelity and Intent Preservation	Translate the architectural meaning and component relationships, not just the syntax of the source IaC.	To combat semantic vendor lock-in and produce a genuinely useful, high-level architectural model.	Vendor Lock-In Dilemma
MR2	Extensibility and Maintainability	Design a modular, technology-agnostic core architecture that can be easily extended to support new IaC tools and clouds.	To address the fragmented IaC ecosystem and ensure the framework's long-term viability and relevance.	Fragmentation of IaC Landscape
MR3	Verifiability and Specification Compliance	Ensure all generated TOSCA 2.0 artifacts are syntactically correct and semantically valid according to the official standard.	To build trust and overcome the toolchain maturity gap by making the framework a reliable source of correct models.	TOSCA Adoption Paradox
MR4	Abstraction–Specificity Balance	Create a lossless translation by systematically retaining all provider-specific details that lack a standard TOSCA equivalent.	To resolve the critical abstraction-precision trade-off, ensuring the generated model is both portable and deployable.	Vendor Lock-In Dilemma

Table 4.1: Overview of Meta-Requirements for the IaC-to-TOSCA Framework

4.4 Design Principles

While Meta-Requirements define what the artifact must achieve, Design Principles (DPs) provide prescriptive guidance on how to construct it. Grounded in the ADR methodology, DPs are distilled from the problem space and theoretical foundations to serve as actionable rules that shape the architecture of the artifact. They are formulated to directly address meta-requirements and ensure that the implementation remains aligned with the project's strategic goals [59].

The following principles are organized into two categories, corresponding to the dual translation strategies investigated in this study and directly addressing the sub-questions of the research. The first set of principles guides the construction of the deterministic, rule-based mapping (RQ1), whereas the second set governs the design of the AI-augmented approach (RQ2). .

4.4.1 Principles for Deterministic Mapping

These principles focus on creating a precise, maintainable, and extensible framework for rule-based translation.

DP1 – Separation of Technology-Specific Concerns

Principle. The translation framework should clearly separate technology-specific logic from core transformation mechanisms. The core must remain technology-agnostic, whereas any functionality that depends on a particular IaC language or provider should be modularized and isolated behind well-defined interfaces.

Justification. This separation promotes extensibility and maintainability by preventing technology-specific details from constraining the overall architecture of the system. This ensures that support for new IaC paradigms or providers can be added or modified independently without requiring fundamental changes to the core transformation logic. By enforcing this boundary, the framework remains adaptable to technological evolution and experimentation in the future.

DP2 - Prioritize Semantic Mapping over Syntactic Translation

Principle. IaC resources must be mapped to TOSCA node types that reflect their architectural role (e.g., `Compute`) rather than their provider-specific implementation names (e.g., `aws_instance`).

Justification. This enforces the extraction of architectural intent, which is a cornerstone of MR1 (Semantic Fidelity). A direct syntactic translation would merely reproduce vendor lock-in in the TOSCA format. By focusing on semantics, the principle ensures that the framework produces a truly abstract and portable architectural model.

DP3 – Preserve Provider-Specific Details in a Traceable Form

Principle. Information specific to a particular cloud provider or platform should be retained rather than discarded when producing a target representation. When a source attribute or configuration element has no direct semantic counterpart in the target model, it should be captured in a consistent and traceable manner that maintains its interpretability and potential reusability.

Justification. This principle addresses the abstraction–precision balance by preventing information loss during the translation process. This ensures fidelity to the original infrastructure intent while maintaining transparency and auditability, thereby supporting MR5 (abstraction–Precision Balance) and MR3 (Verifiability and Specification Compliance). By preserving context-rich details, this approach enables downstream tools to leverage provider-specific knowledge without compromising model portability.

DP4 - Infer Relationships from Multiple Context Cues

Principle. The framework must reconstruct the application topology by analyzing multiple sources of dependency data. This includes not only explicit dependency clauses (such as Terraform's depends_on) but also implicit relationships inferred from property references (e.g., a resource referencing a `subnet_id`) and semantic resource patterns (e.g., an `aws_route_table_association` implies a relationship to a route table and a subnet).

Justification. IaC dependencies are often implicitly expressed through configuration rather than explicit directives. A robust topology can only be reconstructed by considering all these context cues, making this principle essential for achieving MR1 (Semantic Fidelity).

DP5 – Mirror the Hierarchical Structure of the TOSCA Model in Internal Representations

Principle. The internal mechanisms for constructing or manipulating TOSCA service templates should explicitly reflect the hierarchical and compositional structures of TOSCA. The model representation should make the relationships between nodes, capabilities, and requirements directly observable and verifiable, thereby maintaining the structural consistency across the translation pipeline.

Justification. By aligning the internal representation with the conceptual structure of TOSCA, the framework improves the clarity, maintainability, and verifiability of the transformation logic. This correspondence reduces the risk of producing structurally inconsistent outputs, simplifies reasoning about model composition, and supports MR2 (Extensibility and Maintainability) and MR3 (Verifiability and Specification Compliance).

4.4.2 Principles for AI-Augmented Mapping

These principles guide the integration of Large Language Models to create a flexible and adaptable translation approach.

DP6 - Model Domain Knowledge as a Structured, Queryable Knowledge Representation

Principle. The specifications of the target orchestration standard, including its type hierarchies, capability definitions, and relationship constraints, should be represented as a structured, queryable knowledge base that enables semantic retrieval beyond simple text matching.

Justification. Cloud orchestration standards are inherently relational and hierarchical in nature. Representing them as structured knowledge enables semantic queries (e.g., "identify all types that satisfy a given capability requirement") rather than keyword-based retrieval alone. This provides the translation system with a verifiable and navigable source of domain knowledge, reducing ambiguity and improving mapping accuracy, thereby supporting MR1 (Semantic Fidelity).

DP7 - Enable Multi-Source Contextual Retrieval with Relevance Prioritization

Principle. The Retrieval-Augmented Generation (RAG) pipeline must retrieve contextual information from multiple, distinct document sources, including both the source IaC technology's documentation and the target TOSCA specification. A reranking mechanism should be employed to prioritize the most relevant retrieved documents before they are passed to the LLM.

Justification. Accurate IaC-to-standard translation requires an understanding of both source semantics and target constraints. Multi-source retrieval ensures comprehensive domain coverage, whereas relevance prioritization reduces noise and focuses on the most salient information. This enhances translation quality and directly contributes to MR1 (Semantic Fidelity) by grounding the translation in authoritative and contextually appropriate documentation.

DP8 - Constrain Translation Outputs to Validated Domain Schemas

Principle. All translation decisions, including type selection, resource decomposition, and structural mappings, must conform to predefined validated schemas that reflect the target standard's type system and structural constraints.

Justification. TUnconstrained outputs risk producing syntactically invalid or semantically incorrect translation. By enforcing conformance to validated schemas populated with domain-valid constructs, the translation system transitions from probabilistic generation to constrained and verifiable transformation. This principle is essential for ensuring MR3 (verifiability) and guards against erroneous or non-compliant outputs, particularly in AI-augmented approaches, where generation freedom must be carefully bounded.

4.4.3 Mapping Design Principles to Meta-Requirements

This section serves to explicitly connect the Design Principles (the "how") back to the Meta-Requirements (the "what"), demonstrating the traceability and alignment that are central to the Action Design Research methodology. The following table provides a consolidated view of how each design principle contributes to fulfilling one or more of the high-level goals established for the ReTOSCA Framework. This mapping ensures that every architectural decision is purposeful and directly linked to solving the identified problems in the IaC ecosystem.

ID	Design Principle Statement	Fulfills Meta-Requirement(s)
Principles for Deterministic Mapping		
DP1	Separation of Technology-Specific Concerns	MR2, MR3
DP2	Prioritize Semantic Mapping over Syntactic Translation	MR1, MR4
DP3	Preserve Provider-Specific Details in a Traceable Form	MR5, MR3
DP4	Infer Relationships from Multiple Context Cues	MR1
DP5	Mirror the Hierarchical Structure of the TOSCA Model in Internal Representations	MR2, MR3
Principles for AI-Augmented Mapping		
DP6	Model Domain Knowledge as a Structured, Queryable Knowledge Representation	MR1
DP7	Enable Multi-Source Contextual Retrieval with Relevance Prioritization	MR1
DP8	Constrain Translation Outputsto Validated Domain Schemas	MR3

Table 4.2: Mapping of Design Principles to Meta-Requirements

As illustrated in Table 4.2, the relationship between principles and requirements is many-to-many, highlighting the interconnected nature of design. Foundational principles, such as DP1, are crucial for achieving both MR2 and MR3 by enforcing modularity and separation of concerns.

CHAPTER 5

Building, Interventions and Evaluation: Cycle 1

This chapter marks the transition from theoretical design to practical implementation, launching the first Building, Intervention, and Evaluation (BIE) cycle of our Action Design Research. Here, we instantiate the meta-requirements and design principles from Chapter 4 into a tangible artifact, chronicling the iterative process of learning and refinement.

The journey began with the construction of a simple, monolithic prototype—the Alpha artifact of our research—an initial intervention designed to test the fundamental hypothesis: can provider-specific IaC be semantically translated to TOSCA? The answer was yes, but the evaluation of this prototype revealed significant shortcomings in maintainability and extensibility, providing critical, practice-based validation of our architectural design principles. This learning, which represents the primary outcome of Cycle 1, directly triggered the need for complete refactoring.

Therefore, this chapter is structured around the evolution of the first cycle. We first describe the design and implementation of the monolithic Alpha artifact. We then focused on a deterministic, rule-based pipeline for Terraform on AWS, applying this refined artifact to representative use cases. The cycle concludes by evaluating the generated TOSCA 2.0 templates for correctness and compliance, but more importantly, by reflecting on the architectural limitations discovered during the process. These reflections demonstrate the success of this BIE cycle in generating critical design knowledge and provide a solid foundation for the modular Beta artifact presented in the next chapter.

5.1 Design of the Alpha Artifact

Following the problem formulation phase, the initial Meta-Requirements (MRs) and Design Principles (DPs) from Chapter 4 were discussed with the supervisory team. A shared conclusion emerged: the first BIE cycle should validate the core technical hypothesis—that provider-specific IaC can be systematically translated to TOSCA 2.0 while preserving semantic intent (MR1)—before investing in architectural sophistication. Consequently, the Alpha artifact deliberately deprioritized DP1 (Plugin Architecture) and DP5 (Fluent Builder API) in favor of a monolithic, end-to-end prototype optimized for proof of feasibility.

A pivotal technical decision was related to Terraform parsing. Early attempts to author a custom HCL parser exposed the difficulty in reproducing Terraform's dynamic evaluation semantics (variable interpolation, conditional resource creation, module expansion, and data source resolution). Reimplementing this logic risks semantic divergence and substantial maintenance overheads [60]. Therefore, the team adopted a pragmatic alternative: leveraging Terraform's evaluation pipeline via `terraform plan -json`, which yields a machine-readable representation of fully resolved planned values. This trades some source-level traceability for semantic correctness—acceptable within Alpha's scope and directly supportive of MR1 (Semantic Fidelity).

The Alpha artifact comprises four targeted interventions that together realize a minimal, yet complete, translation pipeline. DP1 and DP5 are acknowledged and intentionally deferred to later cycles; Alpha's monolithic implementation serves as an empirical baseline motivating those principles.

Interventions overview:

1. Terraform Configuration Extraction via `terraform plan -json`
2. AWS Resource → TOSCA Node Type Mapping (DP2, DP3)
3. Relationship & Dependency Inference (DP4)
4. TOSCA 2.0 Service Template Generation (MR3)

5.1.1 Intervention 1 - Terraform Configuration Extraction

As established in the overall design, the first step in the translation pipeline is to obtain a stable, semantically complete representation of the source infrastructure. This intervention is achieved by consuming the JSON output emitted by `terraform plan -json`. Instead of parsing raw HCL, the process defers to Terraform as the authoritative interpreter of its language.

This choice was not merely based on convenience. It is a methodological commitment to semantic fidelity (MR1): if the goal is to preserve architectural intent, the input to mapping must reflect what Terraform would actually realize. However, this comes with a trade-off. The process provides some trace-level provenance; for instance, which specific variable file influenced a particular property; however, it gains robustness where it matters most: the correctness of the resulting topology. In other words, the Alpha artifact favors the ground truth over granular traceability, a decision that keeps the early design focused and reduces the risk of misrepresenting the infrastructure's meaning.

5.1.2 Intervention 2 - From AWS Resource to TOSCA Node Type Mapping

Once the infrastructure is available in a resolved, machine-readable form, the next step is to lift provider-specific constructs into TOSCA's vendor-neutral abstractions. The Alpha artifact treats this as an exercise in explicit semantics: it uses a deterministic rule-based mapping that documents, in plain terms, how each AWS resource corresponds to a TOSCA node type and how its attributes project into standard properties. This approach embodies DP2 (Semantic Mapping over Syntactic Translation) and keeps the mapping legible enough to be reviewed, contested, and improved.

The mapping covers a representative set of resources—compute (e.g., `aws_instance` → `Compute`), storage (`aws_s3_bucket` → `ObjectStorage`, etc.). Where TOSCA's Simple Profile offers no precise counterpart (as with security groups), the design opts for a well-scoped custom type, explicitly acknowledging the semantic gap rather than forcing an ill-fitting abstraction.

Two practical design moves make this intervention robust. First, provider-specific details are preserved as structured metadata (DP3)—ARNs, AMIs, zones, tags, and the original Terraform resource type—so that no information is lost. Second, where a property requires interpretation (for example, translating `instance_type` into CPU and memory capabilities), the system relies on documented lookups rather than implicit assumptions. In ambiguous cases, lightweight heuristics (e.g., security group associations, naming signals, and attached volumes) help hypothesize an architectural role while keeping the logic transparent and open to refinement. The result is a mapping layer that is both deterministic and reviewable, providing a clear baseline against which future AI-driven automation can be evaluated.

5.1.3 Intervention 3 - Relationship & Dependency Inference

A set of nodes does not make an architecture; relationships do. This intervention reconstructs the topology graph, which gives the system its shape and meaning. The Alpha artifact combines three complementary cues to infer the edges. It honors explicit declarations through `depends_on`; recognizes implicit dependencies where a resource references another's attributes; and applies pattern knowledge where certain resource types are essentially connective tissue (e.g., `aws_route_table_association` implies a link between a route table and a subnet). Together, these cues operationalize DP4 (Infer Relationships from Multiple Context Cues) and turn isolated components into a coherent model of intent (MR1).

The resulting edges are cast into TOSCA relationships with clear semantics: `HostedOn` to express placement or containment within a network context; `ConnectsTo` to represent communication or access paths (often mediated by security groups); `DependsOn` to capture ordering constraints and build-time prerequisites;

and `AttachesTo` for attachment semantics, such as EBS volumes linked to compute instances. Importantly, the inference logic remains explainable: for any given edge, the system can point back to the originating cue—an explicit `depends_on`, a discovered attribute reference, or a recognized association pattern. This explainability is essential for expert validation and for iteratively refining the rules in which corner cases arise.

5.1.4 Intervention 4 - TOSCA 2.0 Service Template Generation

With nodes and relationships in hand, the final step is to synthesize a TOSCA 2.0 service template that is both readable and spec-compliant (MR3). The Alpha artifact keeps generation intentionally simple: it constructs a minimal TOSCA document with the required header (`tosca_definitions_version: tosca_2_0`), imports the Simple Profile to anchor types, and assembles a `service_template` populated by the mapped node templates and inferred relationship templates. Provider-specific fidelity is retained in each node's metadata block (DP3), ensuring that the neutral representation does not erase details that may be useful for analysis, auditability, or round-trip migration.

The correctness is verified externally by compiling the generated template with Puccini, which serves as an independent gate for both syntax and basic semantics. This external validation step reflects a deliberate engineering philosophy: the generator's job in Alpha is not to be sophisticated but to be good enough to produce artifacts that a standards-compliant processor will accept. The known fragility of ad hoc or string-based generation is, in this phase, a feature rather than a flaw; it visibly motivates the transition toward a Fluent Builder API (DP5) in a subsequent iteration, where maintainability, reuse, and richer invariants can be guaranteed by construction.

5.2 Implementation of Alpha Artifact

This section explains how the Alpha artifact was implemented in practice. The codebase is organized around the four interventions defined in 5.1 and aims for an end-to-end translation that is simple, and explainable and verifiable. The narrative below focuses on the key choices, trade-offs, and safeguards built into each step, keeping the implementation grounded in semantic fidelity (MR1) and specification compliance (MR3).

5.2.1 Intervention 1 - Terraform Configuration Extraction

The implementation began by confronting a core question: whether to parse Terraform's HCL directly or to defer evaluation to Terraform's own engine. A brief prototype of a custom HCL pipeline (lexer + grammar + evaluator) quickly revealed the breadth of semantics that would have to be reproduced with precision: variable interpolation and scoping, `count/for_each`, conditional creation, module expansion, data sources, built-ins and type coercion. Re-implementing this interpreter is error-prone and brittle [60].

Therefore, the extractor delegates the evaluation to Terraform and consumes the JSON representation of the fully resolved plan. This enables the subsequent steps to operate on the intended deployment state rather than on partially interpreted syntax, directly serving MR1. The trade-off is the reduced traceability of "how" a value was derived—an acceptable compromise at the Alpha stage, offset by higher robustness.

Practically, the extractor drives three CLI invocations: initialization, planning a binary artifact, and conversion of that artifact to JSON. The resulting structure is normalized into a compact representation (type, name, provider, values), while non-managed entries (e.g., data sources) are filtered out so that only deployable components flow downstream.

Algorithm 1 Terraform Configuration Extraction

```
Require: path dir to Terraform config
Ensure: list normalized
 1: Confirm directory contains $\geq$ 1 .tf file
 2: Run terraform init in dir
 3: if init fails then
 4:     log error; return []
 5: end if
 6: Run "terraform plan -out=tfplan.binary -input=false"
 7: if If plan fails then
 8:     log error; return []
 9: end if
10: Run "terraform show -json tfplan.binary" $\rightarrow$ tfplan.json
11: Load tfplan.json as dict
12: resources = dict['planned_values']['root_module']['resources']
13: normalized = []
14: for r in resources do
15:     if r.get('mode') != 'managed' then
16:         continue
17:     end if
18:     normalized.append({
          'type': r['type'],
          'name': r['name'],
          'provider': r['provider_name'],
          'values': r['values']
      })
19: end for
20: return normalized
```

5.2.2 Intervention 2 — From AWS Resource to TOSCA Node Type Mapping

With a stable inventory of resolved resources, the mapper lifts AWS constructs into TOSCA 2.0 abstractions via a deterministic ruleset. Each rule returns a TOSCA node template comprising (i) TOSCA node type, (ii) semantically mapped properties, and (iii) a structured metadata block (DP3) that preserves provider-specific fidelity. This approach prioritizes semantic intent over literal syntax (DP2).

Alpha subset.

- `aws_instance` → `Compute`
- `aws_s3_bucket`→ `Storage.ObjectStorage`
- `aws_ebs_volume` → `Storage.BlockStorage`
- `aws_vpc/aws_subnet` → `Network`
- `aws_security_group` → `Root`

On Simple Profile limitations and fallbacks. In several cases, the TOSCA 2.0 Simple Profile does not offer a precise, first-class abstraction. For example, the `aws_security_group` is mapped to type `Root`, with ingress/egress semantics preserved in structured metadata. More generally, when a provider concept lacks a faithful Simple Profile counterpart, the mapper falls back to generic base types and retains provider-specific metadata to avoid information loss. This strategy keeps templates valid and interoperable while making the limitations explicit and future-extensible.

Algorithm 2 Map EC2 Instance to TOSCA Compute

```
Require: normalized Terraform resource $r$ with $r$.type = "aws_instance"
Ensure: TOSCA node template $n$
 1: $v = r['values']$
 2: $specs$ = INSTANCESPECSLOOKUP($v$['instance_type'])
 3: $disk$ = DERIVEDISKSIZE($v$)
 4: $region$ = EXTRACTREGION($v$['availability_zone'])
 5: $n = \{\}$
 6: $n$['name'] = $r$['name']
 7: $n$['type'] = "Compute"
 8: $n$['properties'] = {}
 9: $n$['properties']['name'] = $r$['name']
10: $n$['properties']['num_cpus'] = $specs$['vcpus']
11: $n$['properties']['mem_size'] = FORMATSIZE($specs$['memory_gb'], "GB")
12: $n$['properties']['disk_size'] = FORMATSIZE($disk$, "GB")
13: $n$['metadata'] = {}
14: $n$['metadata']['terraform_type'] = "aws_instance"
15: $n$['metadata']['aws_region'] = $region$
16: $n$['metadata']['instance_type'] = $v$['instance_type']
17: $n$['metadata']['ami'] = $v$['ami']
18: $n$['metadata']['tags'] = $v$['tags']
19: $n$['metadata']['user_data'] = $v$['user_data']
20: return $n$
```

5.2.3 Intervention 3 — Relationship & Dependency Inference

While the previous interventions focused on translating individual resources into TOSCA nodes, this intervention addressed the critical task of reconstructing the infrastructure's topology. This was achieved by inferring the relationships between these nodes, thereby implementing DP4 (Infer Relationships from Multiple Context Cues). The implementation integrated three complementary strategies, applied in a hierarchical manner, to build a comprehensive and accurate dependency graph from the `terraform plan -json output`.

1. Symbolic Reference Extraction from Configuration Expressions:

The primary and most powerful inference strategy leverages the configuration block within the JSON plan. Crucially, while other parts of the plan resolve attribute references to their literal values, this section preserves the original symbolic expressions used in the HCL code. For instance, a reference such as `subnet_id = aws_subnet.public.id` is explicit.

The implementation systematically parses these symbolic references for every resource. This approach is highly reliable because it directly reflects the author's intent as written in the IaC source, eliminating the need for heuristic-based reverse lookups or pattern matching for most dependencies.

Algorithm 3 Infer Relationships From Expressions

```
Require: resource resource, terraform plan terraform_plan
Ensure: list relationships
1: relationships = []
2: expressions = FINDEXPRESSIONSFORRESOURCE(resource, terraform_plan.configuration)
3: for each expr in expressions do
4:     for each symbolic_ref in expr.references do
5:         target_resource_address = symbolic_ref
6:         relationship_type = INFERRELATIONSHIPTYPE(
             expr.attribute_name,
             resource.type,
             target_resource_address.type)
7:         ADDRELATIONSHIP(
             relationships,
             source = resource,
             target = target_resource_address,
             type = relationship_type)
8:     end for
9: end for
10: return relationships
```

2. Explicit `depends_on` Declarations:

As a supplementary strategy, the implementation also processes Terraform's explicit `depends_on` meta-argument. This array, available for each resource in the plan, provides a clear and manually defined dependency ordering. The prototype iterates through this list and creates a default `DependsOn` relationship for each entry, capturing dependencies that may not be evident through attribute references alone.

Algorithm 4 Infer Relationships From `depends_on`

```
Require: resource resource
Ensure: list relationships
1: relationships = []
2: for each dependency_address in resource.depends_on do
3:     relationship_type = "DependsOn"
4:     ADDRELATIONSHIP(
         relationships,
         source = resource,
         target = dependency_address,
         type = relationship_type)
5: end for
6: return relationships
```

3. Heuristic Property Pattern Matching (Fallback):

In rare cases where a dependency is not captured by the first two strategies, a final, heuristic-based fallback mechanism is employed. This method inspects the resolved attribute values of a resource and looks for patterns that suggest a relationship. For instance, it checks whether an attribute named `vpc_id` contains a value that matches the ID of a known VPC resource within the plan.

This is the only strategy that involves a form of reverse lookup, but its scope in the current implementation is intentionally limited to a few common patterns (such as `vpc_id`) to act as a safety net rather than as a primary inference mechanism.

Algorithm 5 Infer Relationships From Attribute Patterns

```
Require: resource resource, map all_resources_map
Ensure: list relationships
 1: relationships = []
 2: if "vpc_id" in resource.attributes then
 3:     vpc_id_value = resource.attributes["vpc_id"]
 4:     if vpc_id_value in all_resources_map then
 5:         target_resource = all_resources_map[vpc_id_value]
 6:         relationship_type = "DependsOn"                ▷ or a more specific type
 7:         AddRelationship(
           relationships,
           source = resource,
           target = target_resource,
           type = relationship_type)
 8:     end if
 9: end if
10: return relationships
```

By combining these three strategies, the Alpha artifact successfully reconstructed the dependency graph for all test cases. The hierarchical application—prioritizing symbolic expressions, followed by explicit declarations, and finally using heuristics as a fallback—ensures a high degree of accuracy and robustness in topology inference.

5.2.4 Intervention 4: TOSCA 2.0 Service Template Generation

The final intervention of the Alpha artifact's implementation was to synthesize the extracted nodes and inferred relationships into a complete and valid TOSCA 2.0 service template. The primary objective was to ensure that the generated output was syntactically correct and semantically compliant with the TOSCA 2.0 specification, making it directly verifiable by the Puccini compiler.

A dedicated generation component was developed to programmatically assemble the final YAML documents. This component constructs the service template following the standard TOSCA structure, including the `tosca_definitions_version` header, metadata, necessary imports for standard types (such as the TOSCA Simple Profile), and the core `service_template`.

Algorithm 6 Generates a Service Template from `nodes` and `relationships`

```
Require: list nodes, list relationships
Ensure: YAML string template_yaml
1: template = CreateToscaScaffold()
2: for each node in nodes do
3:     AddNodeTemplate(
         template,
         name = node.name,
         type = node.type,
         properties = node.properties,
         metadata = node.metadata)
4: end for
5: for each rel in relationships do
6:     source_node_name = rel.source
7:     target_node_name = rel.target
8:     requirement_name = InferRequirementName(rel.type)
9:     AddRequirementToNode(
         template,
         source_node_name,
         requirement_name,
         target_node_name)
10: end for
11: template_yaml = SerializeToYaml(template)
12: return template_yaml
```

A significant challenge was ensuring that all property values adhered to TOSCA's strict type system. Terraform attributes are often typeless strings, whereas TOSCA properties have specific types, such as integers or scalar units. size. The generator component incorporates a type coercion mechanism to format values correctly. For example, a raw memory size value such as "8192" would be transformed into the valid TOSCA scalar-unit string "8 GB". This step was critical for producing a compliant template.

Instead of using the verbose `relationship_templates` section, the generator models dependencies using the more concise and readable requirements block within each node template. This approach co-locates a node's dependencies with its definition, making the topology easier to understand from the perspective of a single component. The name of the requirement (e.g., network, storage) was inferred based on the type of relationship or target node.

A crucial element of this intervention was the integration of a validation step directly into their workflow. After generating each template, the prototype automatically invoked the Puccini compiler as an external process to verify the output. This immediate feedback loop was instrumental during development, allowing for the rapid identification and correction of several categories of common errors.

- **Type Mismatches:** Properties assigned with incorrect data types
- **Missing Required Properties:** Forgetting to include mandatory properties defined by standard TOSCA node types.
- **Invalid Relationship Targets:** References to non-existent nodes within the requirements section.
- **Name Collisions:** Multiple node templates being assigned the same identifier.

By iteratively addressing the errors reported by Puccini, the generator was refined until it consistently produced valid TOSCA 2.0 templates for all the defined test cases. This tight integration of generation and validation ensured that the Alpha artifact's output met the fundamental requirements of correctness and compliance (MR3).

5.3 Evaluation of the Alpha Artifact

The final stage of the first Building, Intervention, and Evaluation (BIE) cycle is the formal assessment of the developed Alpha artifact. The purpose of this evaluation is twofold: first, to determine whether the prototype successfully achieved its primary objective of demonstrating feasibility, and second, to critically analyze its architectural characteristics to generate the knowledge required to inform the design of the subsequent beta artifact.

5.3.1 Evaluation Approach

To ensure a rigorous evaluation, a structured approach was adopted, using a curated set of test cases and clearly defined success criteria linked directly to the Meta-Requirements (MRs) established in Chapter 4.

Test Cases:

A representative suite of Terraform configurations for AWS was created to test the prototype against scenarios of increasing complexity. This suite included:

1. A single, standalone `aws_instance resource`.
2. An `aws_instance` with an attached `aws_ebs_volume`, testing a simple `AttachesTo` relationship.
3. A basic network topology consisting of an `aws_vpc`, an `aws_subnet`, and an `aws_instance` within the subnet, testing `LinksTo` and containment relationships.

Evaluation Criteria:

The Alpha artifact was assessed against the following three criteria:

Evaluation Criteria:

1. **Correctness and Compliance (MR3):** This is the most critical criterion for establishing feasibility. The evaluation measured whether the generated output was a syntactically correct TOSCA 2.0 document or not. The definitive test for this was the successful compilation of the output file using the Puccini TOSCA compiler, which was enhanced to support the TOSCA 2.0 standard as part of the practical contributions of this study.
2. **Semantic Fidelity and Intent Preservation (MR1, MR4):** This qualitative criterion assessed whether the translation correctly captured the architectural intent of the source IaC. The manual inspection focused on two key aspects: a) the correct abstraction of concrete resources to their corresponding abstract TOSCA node types (e.g., `aws_instance` to `Compute`), and b) faithful preservation of all provider-specific attributes within the node's metadata block.

3. **Architectural Fitness for Extension (MR2):** This criterion involved a qualitative "thought experiment" to assess the prototype's monolithic architecture against the meta-requirements for extensibility and maintainability. The assessment considered the effort required to perform two hypothetical modifications: adding support for a new Terraform resource (`aws_s3_bucket`) and adding support for a new IaC technology (e.g., Ansible).

5.3.2 Evaluation Results and Discussion

The evaluation yielded clear and conclusive results, confirming both the viability of the core translation concept and the anticipated limitations of the monolithic design.

Success in Correctness and Semantic Fidelity (MR1, MR3, MR4)

The Alpha artifact successfully translated all test cases in the evaluation suite. For each case, the generated TOSCA 2.0 template was successfully compiled using the Puccini processor without errors. This result provides a definitive empirical validation of the fundamental feasibility of the IaC-to-TOSCA reverse-engineering process. This confirms that the implemented interventions were capable of producing outputs that were fully compliant with the target specifications.

Manual inspection further confirmed that the translation preserved the semantic intent. For instance, in the networking test case, the `aws_instance` resource was correctly abstracted to a `Compute` node template. Critically, its relationship to the `aws_subnet` was correctly modeled via a requirements block with a network requirement pointing to the corresponding `Network` node. Concurrently, provider-specific attributes, such as the AMI ID (ami-0123...) and instance type (t2.micro), were preserved within the metadata block, satisfying MR4.

Anticipated Failure in Architectural Fitness (MR2)

While the artifact succeeded in its primary objective, the evaluation of its architecture against MR2 revealed significant and, by design, anticipated shortcomings.

The thought experiment of adding support for a new `aws_s3_bucket` resource highlighted the brittleness of the design. This modification would require manually editing the central hard-coded mapping dictionary and potentially adding new attribute-handling logic directly into the core application code. This process is error-prone and unsustainable at scale.

More damning was the assessment of adding support for a new IaC technology, such as Ansible. This scenario exposes the fundamental flaw of the monolithic design. The entire parsing logic, which is tightly coupled to the output of the terraform plan -json, would be completely irrelevant for processing an Ansible playbook. Supporting Ansible would necessitate a near-complete rewrite of the prototype's core logic, confirming that the architecture of the Alpha artifact is a dead end for achieving the project's long-term extensibility goals.

5.3.3 Reflections and Learning from Cycle 1

The evaluation of the Alpha artifact concludes the first BIE cycle. The key takeaway is not just that the translation is possible, but also a deeper understanding of how it must be architected to be viable beyond a simple prototype.

The functional success of the prototype validated the core concepts behind the deterministic mapping strategy and provided an initial, partial answer to RQ1. However, the primary contribution of this cycle is the knowledge generated from the architectural failures of artifacts. The difficulty of maintenance and the impossibility of extension provide powerful, practice-based evidence for the necessity of the Design Principles articulated in Chapter 4. The evaluation empirically demonstrates that a scalable solution must adhere to DP1 (Isolate Technology-Specific Logic via a Plugin Architecture) to overcome the observed limitations.

Therefore, the primary contribution of this first BIE cycle is not the Alpha artifact itself but the critical design knowledge generated from its evaluation. This knowledge provides an unequivocal mandate for complete architectural refactoring. This finding directly informs the next cycle of Action Design Research, which will focus on the design and implementation of a modular, plug-in-based Beta artifact built explicitly to address the shortcomings identified here.

CHAPTER 6

Building, Interventions and Evaluation: Cycle 2

This chapter presents the second iteration of the Building, Interventions, and Evaluation (BIE) cycle within the Action Design Research methodology. In this phase, the architectural limitations and insights from the Alpha evaluation, together with the meta-requirements and design principles defined in Chapter 4, served as the foundation for the Beta artifact. The Alpha evaluation confirmed the core hypothesisthat provider-specific IaC can be semantically translated to TOSCA 2.0, but revealed that its monolithic architecture hinders extensibility and maintainability (MR2). Consequently, this iteration focused on a complete architectural refactoring aimed at transforming the prototype into a modular, production-ready framework by operationalizing DP1 (Plugin Architecture) and DP5 (Fluent Builder API). The central goal was to empirically validate these principles by demonstrating that modularization enables a straightforward extension to new IaC technologies and cloud providers while preserving semantic fidelity.

In collaboration with the supervisory panel, the research team defined five core interventions for the Beta artifact: (1) definition of technology-agnostic protocol contracts, (2) implementation of reusable base classes, (3) development of a concrete proof-of-concept Terraform plugin as an exemplar, (4) construction of a type-safe Fluent Builder API for TOSCA generation, and (5) integration of orchestration pipelines with automated validation. The implementation proceeded iteratively, with each intervention building on the previous one to form a cohesive modular architecture. After finalizing all functional components, the Beta artifact was evaluated through an expanded test suite, code-quality metrics, and extensibility experiments. The following sections detail the architectural design, implementation, and evaluation outcomes of the second BIE cycle.

6.1 Design of Beta Artifact

Following the evaluation of the Alpha artifact, as described in Section 5.3, the research team held a meeting with the supervisory panel. The final proposal for the Beta version artifact was agreed upon. The main considerations were the architectural failures identified during the Alpha evaluation, specifically the impossibility of extending the monolithic implementation to support additional IaC technologies, and the theoretical design principles from Chapter 4 that had been intentionally deferred during the first cycle. Building upon the four interventions from Cycle 1 (Alpha), the Beta artifact introduced five additional interventions (Interventions 5-9). Rather than incrementalimprovements, these interventions implement a complete architectural refactoring designed to operationalize DP1 (Plugin Architecture) and DP5 (Fluent Builder API) while preserving the proven semantic-mapping logic from Alpha (DP2, DP3, DP4). The interventions are outlined below:

6.1.1 Architectural Vision and Lessons from Alpha

The evaluation of the Alpha artifact yielded clear results: the monolithic prototype validated the core hypothesis that Terraform configurations for AWS can be semantically translated into TOSCA 2.0, but its architecture proved a major barrier to extensibility and maintainability (MR2, DP1). Attempts to extend the Alpha system—for example, to Ansible or to new AWS resources—revealed severe coupling to the `terraform plan -json` structure and centralized mapping logic. These findings empirically confirm the essentiality of architectural modularity. Consequently, the Beta artifact was guided by a single principle: separating what must be done from how it is done. While the translation pipeline naturally comprises phases—parsing, mapping, relationship inference, and template generation—Alpha conflates these concerns. Beta addressed this through protocol-based abstraction [61], allowing each phase to define contracts that are independent of specific implementations.

The resulting architecture was structured into three layers: (1) Protocol Layer (Core Contracts): Abstract interfaces such as `PhasePlugin`, `SourceFileParser`, and `ResourceMapper` define the input/output types and the validation rules. (2) Base Implementation Layer (Reusable Scaffolding): abstract base classes such as `BasePhasePlugin` and `BaseResourceMapper` encapsulate shared logic and lifecycle handling. (3) Plugin Layer (Technology-Specific Implementations): Concrete plugins, including a proof-of-concept Terraform plugin, instantiate these contracts, and future modules (such as Ansible or CloudFormation) can do so without core modifications. This layered design embodies the separation-of-concerns principle [62], ensuring localized change: adding a new AWS resource now requires only a new `SingleResourceMapper`, not a rewrite of the pipeline.

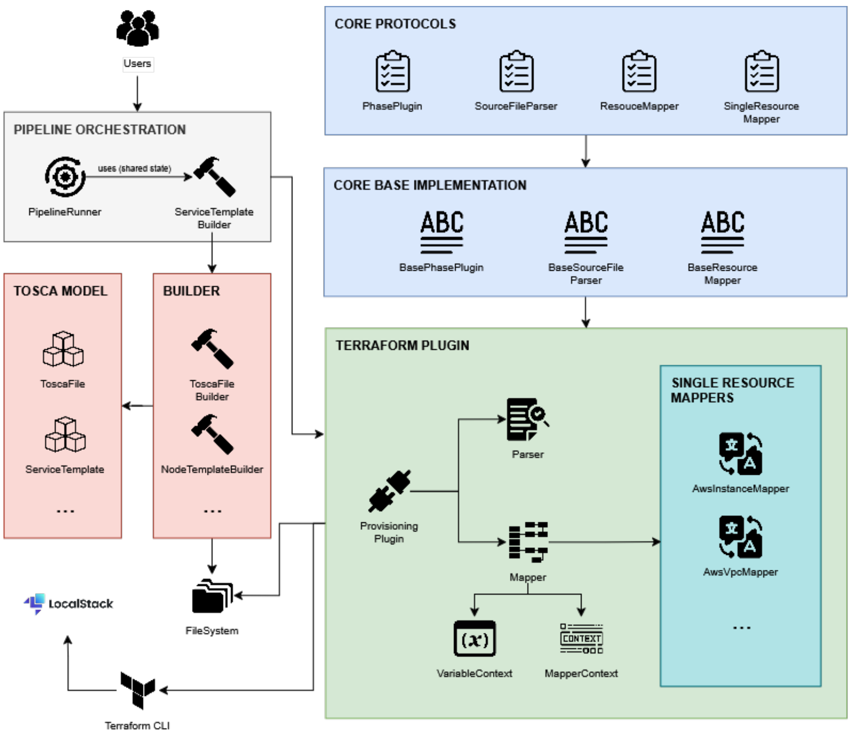


Figure 6.1: Layered architectural model of the Beta artifact showing the separation between protocol, base, and plugin layers.

Although adopting this architecture introduced complexity, it achieved long-term maintainability. Supported by the Strategy and Template Method patterns [63], it trades initial simplicity for cumulative reusability, where each new plug-in or resource type demands less effort. Another limitation identified in Alpha was the fragile string-based YAML generation. To resolve this, Beta introduced DP5 (Fluent Builder API), which replaces ad-hoc string concatenation with a type-safe, compositional API that builds valid TOSCA structures and shifts error detection to compile time.

Extensibility Validation through Architectural Review. The plugin-based architecture was evaluated collaboratively by the ADR team through structured architectural review sessions, rather than through multiple complete plugin implementations. This approach, consistent with the ADR's emphasis on reflective practice, involved a systematic analysis of the protocol contracts, base class abstractions, and extension points to assess whether the architecture demonstrably supports multi-IaC extensibility. The evaluation focused on the traceability of technology-specific logic (protocol isolation), clarity of extension contracts (interface documentation and typing), and theoretical effort required to integrate new IaC technologies. Through guided walkthroughs, the team examined how a hypothetical Ansible plugin would be implemented, identified the required components (parser for YAML playbooks, mappers for task-to-node translation, and context object for inventory management), and confirmed that no modifications to the core framework classes would be necessary. While the Terraform plugin serves as the sole empirical instantiation within the scope of Cycle 2, the architectural patterns it exemplifies—protocol conformance, registry-based dispatch, and base class reuse—were validated as generalizable through expert consensus. This validation strategy prioritizes architectural fitness over implementation completeness, acknowledging that full empirical validation through multiple working plug-ins remains a direction for future work.

In essence, Beta re-engineered the Alpha prototype into a modular, extensible framework suitable for production and future multi-IaC research, operationalizing DP1 and DP5 while preserving the semantic fidelity validated in Cycle 1.

The architecture's extensibility is validated architecturally rather than empirically, establishing a foundation for subsequent plug-in development while maintaining focus on the deterministic Terraform-to-TOSCA translation as the primary contribution of Cycle 2.

6.1.2 Intervention 5 - Fluent Builder API for TOSCA Generation

This intervention replaces the fragile, string-based TOSCA generation of the Alpha artifact with a type-safe, compositional Fluent Builder API for programmatic construction of TOSCA 2.0 documents. Instead of producing YAML through concatenation, the system uses a structured hierarchy of builders that enforces syntactic and semantic correctness at build time. This represents a methodological shift toward correctness by construction (MR3), wherein validation rules from the TOSCA specification are embedded within the API itself, ensuring that invalid templates cannot be created. Errors that previously emerged only during external validation were detected during template assembly, reducing debugging overhead and improving reliability.

The design introduces a two-layer architecture that separates validation from the construction concerns. The validation layer employs Pydantic models to define canonical TOSCA 2.0 structures (e.g., `ServiceTemplate`, `NodeTemplate`, and `RequirementAssignment`), enforcing type safety and specification compliance through declarative schemas. All models inherit from `ToscaBase`, which provides a shared validation logic and ensures that only well-formed TOSCA elements can be instantiated. The builder layer exposes a fluent, method-chaining interface (e.g., `ToscaFileBuilder`, `ServiceTemplateBuilder`, and `NodeTemplateBuilder`) that accumulates the state incrementally and delegates validation to the underlying Pydantic models only at build time. This separation enables a flexible construction order while maintaining strict correctness guarantees, following the fluent interface pattern [64]. Builders are immutable by design, a principle borrowed from functional programming that prevents side effects and allows safe reuse.

Validation is handled through a hybrid strategy: property-level checks (types, required fields) are performed eagerly for immediate feedback, whereas topology-level constraints (dependency and relationship validity) are validated at the build phase. This balances early error detection and flexibility during incremental template construction. The API also incorporates progressive disclosure, offering simple methods for common use cases and advanced options for complex scenarios, thereby ensuring usability without sacrificing completeness.

Additional mechanisms include centralized type coercion, which automatically converts IaC values (e.g., integers or strings) into TOSCA-compliant scalar units, and a controlled escape hatch (`with_raw_section()`) for unsupported features. Together, these design choices eliminate recurring types and syntax errors, address maintainability concerns, and provide a stable foundation for future automation.

This intervention directly fulfills MR3 (Verifiability and Specification Compliance) and MR2 (Extensibility and Maintainability) while operationalizing DP5 (Fluent Builder API). It transforms TOSCA generation into a robust, reusable, and specification-driven process that will also underpin the AI-assisted translation envisioned in Cycle 3.

6.1.3 Intervention 6 - Core Protocol Definition

This intervention operationalizes the architectural requirement for extensibility by introducing a set of technology-agnostic protocol interfaces that form the backbone of the Beta artifact. Instead of embedding translation logic into concrete classes, the system defines abstract contracts that any IaC technology, such as Terraform, Ansible, or CloudFormation, can implement. This shift goes beyond refactoring; it embodies a methodological commitment to extensibility and maintainability (MR2), ensuring that the translation logic remains universal while technology-specific concerns are isolated. By elevating protocols to first-class architectural elements, the design establishes explicit interaction boundaries between plug-ins and the framework, making extension points transparent and reducing complexity for future contributors.

This approach introduces a trade-off between simplicity and flexibility. The Alpha artifact's monolithic pipeline was straightforward to follow but rigid for extension. In contrast, the Beta design adds abstraction layers and indirection, demanding a deeper conceptual model while enabling modular growth. This deliberate choice prioritizes long-term adaptability over short-term learnability, addressing the extensibility failures observed in cycle 1.

The intervention defines four core protocols corresponding to the logical phases of the translation pipeline:

1. **PhasePlugin:** a top-level lifecycle contract governing validation, execution, and output consistency across phases.
2. **SourceFileParser:** defines how IaC files are parsed into a normalized, technology-agnostic structure.
3. **ResourceMapper:** governs the conversion of normalized resources into TOSCA topology graphs, including relationship inference (DP4).
4. **SingleResourceMapper:** handles the mapping of individual resources to node templates, enabling registry-based extensibility.

These protocols are minimal yet complete, aligning with the open-closed principle [65]. The chosen granularity, neither too coarse nor overly fine, balances flexibility with usability, allowing developers to replace or extend specific phases without reimplementing the entire pipeline.

Ultimately, this intervention lays the foundation for plug-in-based extensibility, enabling new IaC technologies or AI-driven mapping strategies to be integrated seamlessly into the system. It fulfills MR2 by ensuring maintainability through modularity.

6.1.4 Intervention 7 - Base Implementation Classes

This intervention introduces an abstract base layer that provides reusable scaffolding, shared utilities, and consolidated algorithmic logic. Its purpose is to prevent plug-in developers from re-implementing common functionality while maintaining consistent quality across implementations. Built upon the protocol layer, this design allows plug-ins to focus on technology-specific behavior while inheriting a stable, tested foundation. This approach embodies the DRY principle and promotes knowledge reuse. Alpha's validated relationship-inference algorithms are refactored into reusable methods within `BaseResourceMapper`, ensuring that proven dependency-resolution logic is shared across all plugins and that maintenance improvements automatically propagate, fulfilling MR2 (maintainability).

The base layer defines three abstract classes that are aligned with the protocol hierarchy.

1. **BasePhasePlugin (Lifecycle Management):** Implements the `PhasePlugin` protocol and defines a fixed four-step workflow—validation, execution, cleanup, and result packaging—implemented through the Template Method pattern. The `execute()` method acts as a template method and is never overridden by the plug-ins. Instead, it delegates customization to abstract hooks such as `get_parser()` and `get_mapper()` and to overridable utilities such as `find_source_files()`. Centralized logging, error handling, and state management ensure consistency across all the phases.
2. **BaseSourceFileParser (I/O and Error Handling):** Implements `SourceFileParser` and standardizes file reading, encoding detection, and structured error reporting. It applies a hybrid error-handling strategy: fail-fast for structural issues (e.g., malformed JSON) and collect-and-report for semantic issues (e.g., unknown resource types), thereby improving developer feedback during parsing.
3. **BaseResourceMapper (Topology Inference):**Implements `ResourceMapper` and encapsulates Alpha's relationship-inference logic—symbolic reference extraction, explicit dependencies, and heuristic pattern matching—while handling edge deduplication, cycle detection, and relationship classification. Protected helper methods enable reuse or selective overrides, thereby upholding the open–closed principle.
4. **SingleResourceMapper:** handles the mapping of individual resources to node templates, enabling registry-based extensibility.

To prevent the fragile base class problem, workflow control is frozen in `execute()`, while operational flexibility is exposed through fine-grained, composable utilities.

Overall, this intervention operationalizes DP1 (Plugin Architecture) and MR2 (maintainability) by providing a reusable infrastructure that bridges abstract protocols and concrete plugins.

6.1.5 Intervention 8 — Terraform Plugin Implementation

This intervention introduces a proof-of-concept Terraform plugin as the first full instantiation of the plugin-based architecture defined in previous interventions. Its purpose is twofold: to provide Beta's core translation capability—from Terraform AWS configurations to TOSCA 2.0—and to validate the extensibility and maintainability of the modular framework. Rather than rewriting Alpha's logic, the design preserves its proven parsing and mapping strategies while reorganizing them within the protocol and the base-class structure. This ensures functional continuity: the Beta plugin should produce semantically identical TOSCA outputs to Alpha for equivalent inputs, fulfilling MR1 (Semantic Fidelity) and serving as a regression benchmark for architectural refactoring.

The Terraform plugin serves as a proof of concept rather than a complete Terraform-to-TOSCA translator. Its goal is to demonstrate the viability of the plug-in mechanism and the modular separation of the parsing, mapping, and building phases. The implementation covers 31 AWS resource types organized across major infrastructure domains, as follows:

- **Compute:** `aws_instance`
- **Networking:** `aws_vpc`, `aws_subnet`, `aws_security_group`, routing, gateways, and Elastic IPs
- **Storage:** `aws_s3_bucket`, `aws_ebs_volume`, `aws_volume_attachment`
- **Load Balancing:** `aws_lb`, `aws_lb_target_group`, `aws_lb_listener`, etc.
- **Database:** `aws_db_instance`, `aws_rds_cluster`, `aws_db_subnet_group`
- **Caching:** `aws_elasticache_cluster`, `aws_elasticache_replication_group`, `aws_elasticache_subnet_group`
- **DNS and IAM:** `aws_route53_zone`, `aws_route53_record`, `aws_iam_role`, `aws_iam_policy`

The architecture of the plugin is composed of four cooperating components:

1. **TerraformPhasePlugin (Phase Orchestration):** Extends `BasePhasePlugin` and orchestrates parsing, mapping, and TOSCA generation through the builder API. It operates statelessly for thread safety and is testable.
2. **TerraformParser (Source Extraction):** Invokes Terraform's CLI and normalizes its

JSON output into technology-agnostic resource objects while preserving provenance metadata.

3. **TerraformResourceMapper (Topology Construction):** Extends `BaseResourceMapper` and employs a *registry-based architecture* to associate each AWS resource type with a dedicated mapper, enabling dynamic extensibility.
4. **SingleResourceMappers (Granular Translation Rules):** Implement fine-grained mappings for the 31 supported AWS resources, including metadata preservation (DP3) and type coercion logic.

The plugin integrates error-handling mechanisms that strengthen its robustness and support its role as a reference implementation for future extensions. It also serves as a reference for future IaC technologies. This intervention validates MR1 (Semantic Fidelity) and MR2 (extensibility), operationalizes DP, DP3, and DP4, and demonstrates that Alpha's translation logic can be modularized into a reusable, extensible proof-of-concept plug-in.

6.1.6 Intervention 9 – Pipeline Orchestration

After defining the core components—protocols, base classes, builders, and the Terraform plugin—the Beta design introduced a coordination layer that integrates them into a cohesive translation pipeline. This intervention established (1) the orchestration logic through a `PipelineRunner` and (2) the simulation and validation infrastructure composed of LocalStack and the updated Puccini compiler.

The `PipelineRunner` implements a phase-based orchestration pattern, executing plugins sequentially—provisioning, configuration, policy—while maintaining a shared `ServiceTemplateBuilder`. This shared builder embodies the collaborative construction principle: each plug-in contributes incrementally to the same TOSCA model without global knowledge of the topology. Although shared mutability conflicts with functional immutability principles, the design mitigates risks through a controlled API: plugins may only add or enrich elements and never delete or arbitrarily modify them. This balances extensibility (MR2) and verifiability (MR3).

The `PipelineRunner` initializes the builder, discovers available plug-ins, executes each `execute()` method in the correct phase order, logs their contributions, and finalizes the resulting model. Phase sequencing is convention-based; plugins identify their role via naming or metadata, avoiding hard-coded ordering while supporting future extensibility. Logging and intermediate checkpoints make the builder state inspectable, enabling fine-grained debugging and traceability across all phases.

LocalStack Integration provides a lightweight simulation of the AWS environment to test the pipeline end-to-end without deploying it to real cloud resources. The Terraform plugin can operate through `tflocal`, redirecting API calls to `localhost:4566`, thereby enabling the orchestration logic to run in a controlled sandbox. This allows the verification of pipeline behavior and resource discovery under realistic conditions while avoiding cost and latency.

For template validation, the generated TOSCA 2.0 service templates were compiled and verified using the updated Puccini processor, which ensures both syntactic and semantic conformance to the latest standard. Puccini performs type resolution, inheritance checks, and topology validation, confirming that the translation output is a formally correct TOSCA 2.0 document.

Together, the `PipelineRunner`, LocalStack simulation, and Puccini validation complete the Beta architecture, transforming isolated plugins into a verifiable, extensible, and standard-compliant IaC-to-TOSCA translation pipeline.

6.2 Implementation of Beta Artifact

The Beta artifact was implemented iteratively over three months using a bottom-up strategy. Starting with the Fluent Builder API, successive interventions (protocols, base classes, the Terraform plugin, and finally pipeline orchestration with LocalStack) were incrementally developed and validated. Each layer was tested before building dependent layers to ensure modular integrity, reduced integration risk, and systematic debugging.

6.2.1 Intervention 5 – Fluent Builder API

Building upon the hybrid validation and immutability principles introduced in Section 6.1.2, this intervention operationalizes these concepts through the Fluent Builder API, which replaces the fragile, string-based TOSCA generation used in the Alpha artifact.

Two-Layer Architecture

The architecture separates validation and construction concerns into two distinct layers (Figure 6.2):

- **Layer 1 (Validation Models):** Definisce le strutture canoniche di TOSCA 2.0 utilizzando Pydantic. Il pannello sinistro della Figura 6.2 mostra la gerarchia di ereditarietà radicata in `ToscaBase`, che fornisce la logica di validazione condivisa per tutti gli elementi TOSCA (`ServiceTemplate`, `NodeTemplate`, `RequirementAssignment`, ecc.).
- **Layer 2 (Builder API):** Espone un'interfaccia a catena di metodi, illustrata nel pannello destro della Figura 6.2. I builder accumulano lo stato in dizionari Python e delegano la validazione al primo livello solo al momento della chiamata di `build()`.

This separation introduces a clear validation boundary: intermediate construction can be incomplete, but only `build()` produces compliant TOSCA objects. This design resolves Alpha's core limitation—validation scattered across the codebase—and operationalizes DP5 (Fluent Builder API) and MR3 (verifiability).

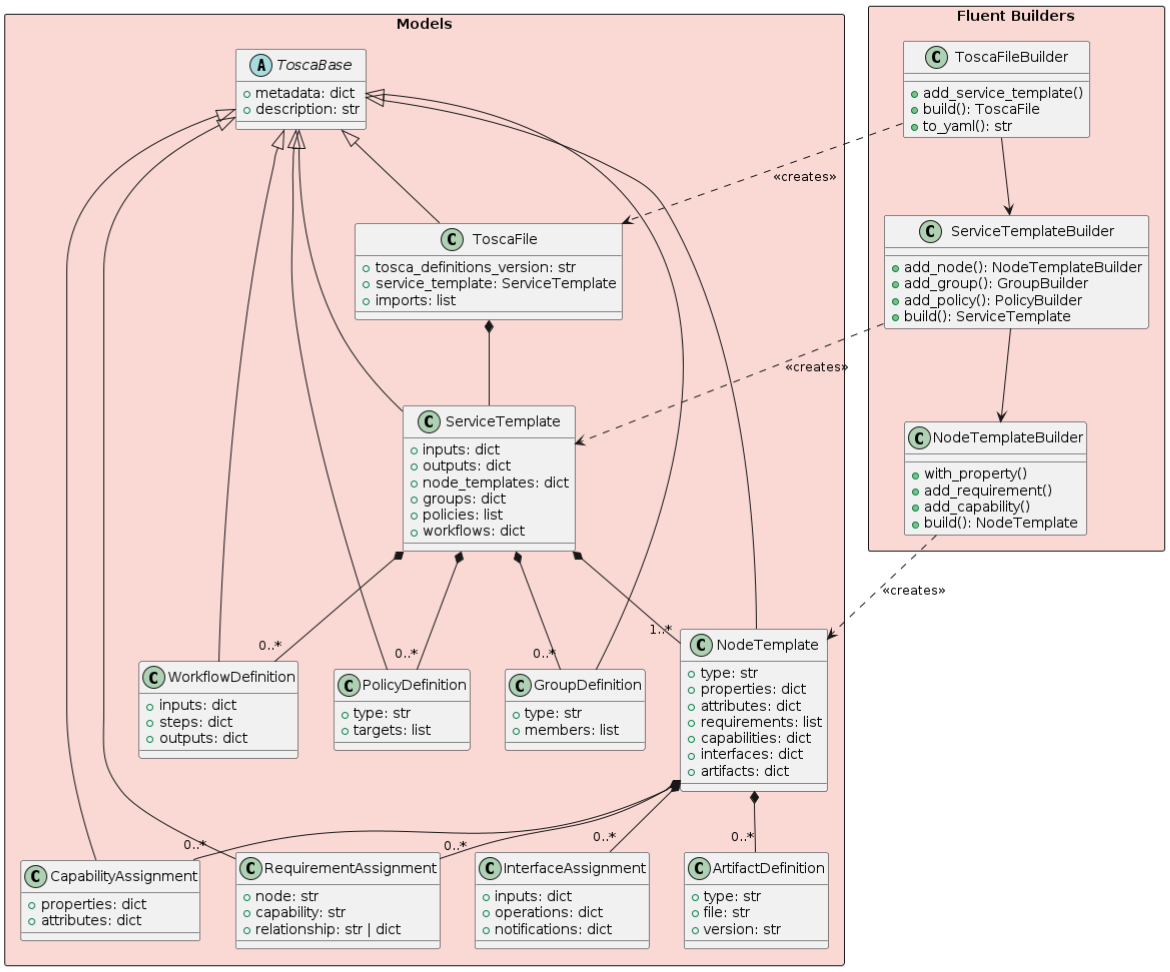


Figure 6.2: Left: Pydantic models ensure TOSCA 2.0 compliance via inheritance from `ToscaBase`. **Right:** Fluent builders use method-chaining for incremental construction, with the `ServiceTemplateBuilder` coordinating multiple `NodeTemplateBuilder` instances and their sub-builders.

The Fluent Builder API implements validation and construction principles through a two-layer architecture that separates schema enforcement from object creation. The validation layer defines a set of Pydantic models that represent the canonical TOSCA2.0 constructs. All models inherit from `ToscaBase`, which ensures type safety through Python typing, forbids undefined attributes, and applies cross-field validation using custom model validators.

For instance, the `RequirementAssignment` model enforces that at least one of `capability`, `node`, or `relationship` is specified, guaranteeing both syntactic and semantic compliance with the standards.

The builder layer encapsulates TOSCA element creation through method chaining, which enables declarative and incremental topology specifications. The pattern is exemplified by the `NodeTemplateBuilder`.

```
# Usage example - declarative node construction
node = (NodeTemplateBuilder("web_server", "Compute")
        .with_property("num_cpus", 2)
        .with_property("mem_size", "4 GB")
        .add_requirement("network")
            .to_node("private_net")
            .and_node()
        .build())
```

Validation is deferred until `build()` is called, allowing a flexible construction order. Nested builders, such as requirements or capabilities, maintain parent references to support natural method chaining (`and_node()`). Each method returns `self`, thereby avoiding side effects and ensuring immutability.

At the top level, the `ServiceTemplateBuilder` coordinates multiple node builders into a unified topology (Figure 6.2). It maintains a registry of builders, exposes factory methods (`add_node()`, `add_group()`), and acts as the shared state for the `PipelineRunner`, enabling collaborative construction across the plugins. When executed, its `build()` method aggregates validated child outputs into a complete `ServiceTemplate` that is compatible with tools such as Puccini.

This layered design improves type safety, flexibility, and maintainability while facilitating extensibility through reusable sub-builders (e.g., `RequirementBuilder`, `CapabilityBuilder`). The main drawback is the additional abstraction introduced by deferred validation, although clear error reporting and strong typing mitigate this cost. Overall, the Fluent Builder API establishes a robust foundation for generating modular TOSCA-compliant templates and serves as a core enabler of the reverse-engineering framework.

6.2.2 Intervention 6 - Core Protocol Definition

Building on the conceptual foundation introduced in Section 6.1.3, this intervention focuses on the technical realization of the protocol layer that operationalizes the extensibility principle. Whereas the earlier design phase established the rationale and structure of the four core protocols, the present stage translates those ideas into a concrete Python implementation that enables independent plug-in development.

A key design decision was the definition of contracts. Two paradigms were evaluated: inheritance-based protocols using Python's `abc.ABC` and structural protocols using `typing.Protocol`. The latter was adopted because it minimizes coupling; any class that exposes the required methods automatically satisfies the protocol without inheriting from the framework base classes. This approach enables independent plug-in development and aligns with Python's dynamic philosophy while maintaining static verifiability through tools such as mypy. Runtime validation is then delegated to the explicit registration checks performed by the orchestrator.

Protocol Hierarchy and Responsibilities

The implementation defines four complementary protocols that mirror the conceptual phases of the translation pipeline (Figure 6.3). The architecture follows a clear separation between protocol contracts (bottom panel) and their base implementations (top panel), establishing explicit boundaries for the plug-in development.

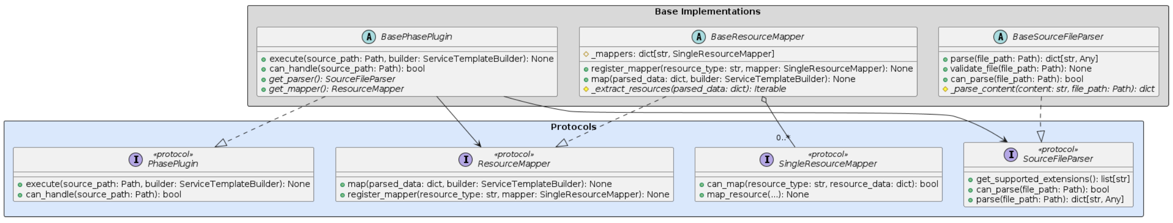

Base Implementations
BasePhasePlugin
execute(source_path: Path, builder: ServiceTemplateBuilder): None
can_handle(source_path: Path): bool
get_parser(): SourceFileParser
get_mapper(): ResourceMapper
BaseResourceMapper
_mappers: dict[str, SingleResourceMapper]
register_mapper(resource_type: str, mapper: SingleResourceMapper): None
map(parsed_data: dict, builder: ServiceTemplateBuilder): None
_extract_resources(parsed_data: dict): Iterable
BaseSourceFileParser
parse(file_path: Path): dict[str, Any]
validate_file(file_path: Path): None
can_parse(file_path: Path): bool
_parse_content(content: str, file_path: Path): dict
Protocols
«protocol» PhasePlugin
execute(source_path: Path, builder: ServiceTemplateBuilder): None
can_handle(source_path: Path): bool
«protocol» ResourceMapper
map(parsed_data: dict, builder: ServiceTemplateBuilder): None
register_mapper(resource_type: str, mapper: SingleResourceMapper): None
0..*
«protocol» SingleResourceMapper
can_map(resource_type: str, resource_data: dict): bool
map_resource(...): None
«protocol» SourceFileParser
get_supported_extensions(): list[str]
can_parse(file_path: Path): bool
parse(file_path: Path): dict[str, Any]

Figure 6.3: Four structural protocols define technology-agnostic contracts using `typing.P⌋ rotocol`. Each declares capability query methods (`can_parse()`, `can_map()`, `can_handle()`) that enable dynamic runtime dispatch and decouple plugin logic from specific technologies.

As illustrated in Figure 6.3, the protocol hierarchy comprises:

1. **SourceFileParser (bottom-right):** Defines the contract for converting IaC source files into structured data. The generic return type `dict[str, Any]` reflects the choice to defer normalization to the mapping phase, thereby avoiding premature schema enforcement across heterogeneous IaC formats. Runtime parser selection is handled via `can_parse()`.

2. **SingleResourceMapper (bottom-center-right):** Translates individual resources by mutating a shared `ServiceTemplateBuilder`, implementing the collaborative construction principle. The optional `context` parameter passes provider-specific metadata while maintaining a generic interface.
3. **ResourceMapper (bottom-center-left):** Coordinates multiple `SingleResou⌋ rceMapper` instances using the Registry pattern. Registration and execution are decoupled (`register_mapper()` and `map()`), allowing for reusable and customizable mapping strategies.
4. **PhasePlugin (bottom-left):** Composes parser and mapper into a single translation phase. It exposes a uniform interface to the `PipelineRunner`, treating each plug-in as a modular unit that consumes IaC input and enriches the shared builder.

The protocol layer defines the execution backbone of the framework, organizing translation into a clear, hierarchical flow::

```
PhasePlugin.execute(source_path, builder)
|-- parser.parse(source_path) -> dict[str, Any]
`-- mapper.map(parsed_data, builder)
     `-- single_mapper.map_resource(...) -> builder.add_node(...)
```

A `PhasePlugin` orchestrates this process by delegating parsing and mapping to specialized components. Parsers normalize provider-specific syntax, resource mappers route data to domain-specific handlers, and single-resource mappers create granular TOSCA nodes. Each layer remains self-contained, enabling internal evolution without affecting global orchestration.

Three principles underpinned the design. Technology agnosticism is ensured by relying only on primitive Python types and the shared `ServiceTemplateBuilder`, allowing new IaC technologies to be integrated through uniform interface contracts. Capability-based dispatch leverages methods like `can_parse()` or `can_map()` to dynamically select components at runtime, following the Look Before You Leap-principle[1] for predictable behavior. Finally, validation occurs both staticallythrough `mypy` type checkingand optionally at runtime via `@runtime_checkable`, ensuring protocol compliance throughout the development cycle.

[1] `www.realpython.com/ref/glossary/lbyl/`

Reusable base classes complement these interfaces by encapsulating the common logic. `BaseSourceFileParser` handles file I/O and parsing errors, `BaseResourceMapper` manages dispatch and registry operations, and `BasePhasePlugin` coordinates the parser–mapper lifecycle. This balance between structural typing and inheritance eliminates the boilerplate while preserving flexibility.

Together, protocols and bases fulfill DP1 (Plugin Architecture), MR2 (Extensibility), and MR3 (Verifiability), forming a modular and technology-neutral foundation that supports the evolution of the reverse engineering framework.

6.2.3 Intervention 7 - Base Implementation Classes

With protocols defining architectural contracts and builders providing the primitives for TOSCA construction, this intervention introduced the intermediate layer that bridges abstract interfaces and concrete plugins. Its goal was to reduce boilerplate across technologies while maintaining flexibility for specialization. The solution formalized three reusable base classes—`BaseSourceFileParser`, `BaseResourceMapper`, and `BasePhasePlugin`—each implementing common orchestration logic and delegating technology-specific behavior to subclasses.

Architectural Overview

Figure 6.4 illustrates the base implementation layer and its relationship to the protocol contracts defined in Intervention 6. Each base class implements its corresponding protocol (gray boxes) while providing a reusable infrastructure for concrete plugins (shown in blue). The design embodies the Template Method pattern: invariant workflows are implemented in public methods (e.g., `execute()`,`parse()`,`map()`), whereas technology-specific behavior is delegated to abstract protected methods (highlighted in yellow).

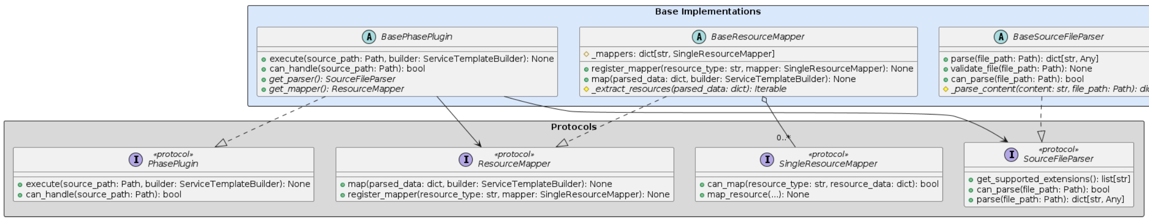

Base Implementations
BasePhasePlugin
execute(source_path: Path, builder: ServiceTemplateBuilder): None
can_handle(source_path: Path): bool
get_parser(): SourceFileParser
get_mapper(): ResourceMapper
BaseResourceMapper
_mappers: dict[str, SingleResourceMapper]
register_mapper(resource_type: str, mapper: SingleResourceMapper): None
map(parsed_data: dict, builder: ServiceTemplateBuilder): None
_extract_resources(parsed_data: dict): Iterable
BaseSourceFileParser
parse(file_path: Path): dict[str, Any]
validate_file(file_path: Path): None
can_parse(file_path: Path): bool
_parse_content(content: str, file_path: Path): dict
Protocols
«protocol» PhasePlugin
execute(source_path: Path, builder: ServiceTemplateBuilder): None
can_handle(source_path: Path): bool
«protocol» ResourceMapper
map(parsed_data: dict, builder: ServiceTemplateBuilder): None
register_mapper(resource_type: str, mapper: SingleResourceMapper): None
0..*
«protocol» SingleResourceMapper
can_map(resource_type: str, resource_data: dict): bool
map_resource(...): None
«protocol» SourceFileParser
get_supported_extensions(): list[str]
can_parse(file_path: Path): bool
parse(file_path: Path): dict[str, Any]

Figure 6.4: Three abstract base classes provide reusable scaffolding for plugin development. `BasePhasePlugin` coordinates the translation lifecycle between parser and mapper, `BaseResourceMapper` implements a registry pattern for dynamic mapper composition and fallback handling, and `BaseSourceFileParser` standardizes file I/O and error management.

The base implementation layer unifies common behaviors across file parsing, resource mapping, and phase orchestration, offering reusable scaffolding that ensures consistency and reduces duplication across the technologies. Its primary objective is to provide a stable foundation where developers implement only technology-specific logic while inheriting shared functionality for validation, logging, and error handling.

The `BaseSourceFileParser` defines a standardized parsing workflow that encompasses file validation, reading, parsing, and structured logging. Only the ⌋ `_parse_content()` method is overridden by subclasses, allowing them to focus exclusively on the syntax transformation. Centralized encoding detection and fault management guarantee predictable behaviors and uniform error semantics across heterogeneous IaC formats.

The `BaseResourceMapper` converts normalized IaC data into TOSCA node templates through a controlled workflow that combines the Template Method and Registry patterns. It manages mapper registration, resource extraction, and dispatching to appropriate `SingleResourceMapper` strategies. A fallback mechanism generates generic `Root` nodes when mappings are unavailable, ensuring completeness and preserving the provider-specific metadata (DP3). Additional helpers handle name normalization to maintain compliance with TOSCA.

At the orchestration level, the `BasePhasePlugin` governs the translation lifecycle by composing the parser and mapper components into cohesive execution units. Its `execute()` method coordinates source discovery, parsing, and mapping, while subclasses define only `get_parser()`, `get_mapper()`, and `get_plugin_inf` ⌋

`o()`. This approach balances standardization and flexibility, allowing technology-specific customization without altering the global orchestration logic. A best-effort error-handling policy further enhances the robustness of large-scale translations.

Together, these classes integrate three key design patterns—Template Method, Registry, and Strategy [63]—to promote modularity, reusability, and maintainability of the code. Protocol conformance ensures type safety, which is verified statically using `mypy`. The result is a clean, extensible architecture in which integrating new IaC technologies requires minimal effort, directly addressing DP1 (Plugin Architecture) and MR2 (Extensibility).

6.2.4 Intervention 8 - Terraform Plugin Implementation

With the framework's architectural foundation complete (builders, protocols, and reusable base classes), the fourth intervention demonstrated the architecture's capability to support a real-world, large-scale translation scenario. The Terraform plugin implements the full provisioning phase of the Beta artifact, translating Terraform configurations targeting AWS into TOSCA 2.0 models. It serves as the system's principal translator and validates the architectural choices established in previous interventions. The plugin's structure, comprising approximately 9,000 lines across 31 AWS resource mappers, reflects the layered composition foreseen in the design: high-level orchestration, parsing, mapping, and a collection of fine-grained resource translators.

Plugin Architecture Overview

Figure 6.5 illustrates the modular composition of the Terraform plugin. The architecture follows a clear separation of concerns across four layers:

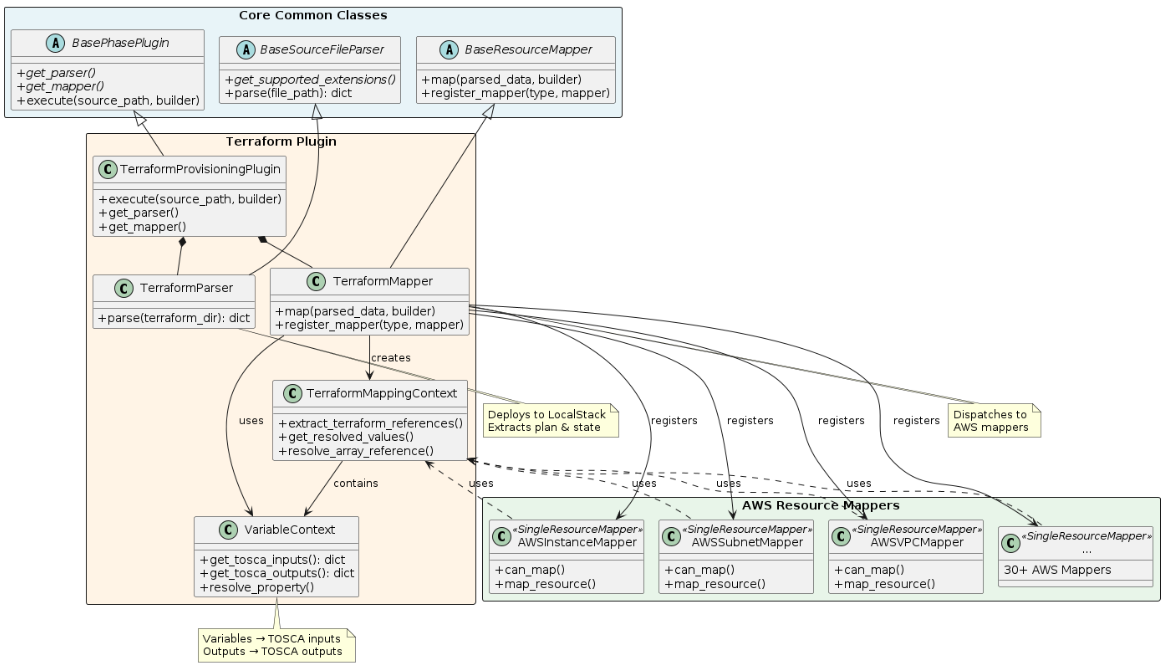


Figure 6.5: The `TerraformProvisioningPlugin` coordinates the parsing and mapping of Terraform configurations through dedicated components. The `TerraformParser` and `TerraformMapper` interact via a shared `TerraformMappingContext` that supports variable resolution and cross-resource queries. The lower layer comprises multiple AWS-specific `SingleResourceMapper` implementations (e.g., EC2, VPC, S3) dynamically dispatched through the registry pattern. Dashed lines indicate dependency injection; solid arrows represent delegation and inheritance.

The Terraform plugin operationalizes the framework's plugin architecture by instantiating a complete translation pipeline from the Terraform configurations to the TOSCA 2.0 templates. Built on the common base classes, it demonstrates how a new IaC technology can be integrated through minimal, well-scoped components. The plugin's orchestration logic remains deliberately lightweight: the `TerraformProvisioningPlugin` delegates to a parser and a mapper, while context objects manage variable resolution and cross-resource dependencies. This modular structure exemplifies the principle of lightweight composition: most orchestration concerns are inherited, and the plugin itself focuses exclusively on Terraform-specific behavior.

The `TerraformParser` implements a two-mode parsing strategy with an automatic fallback. The default mode performs full deployment to LocalStack: it executes `terraform plan -json` to extract declarative intent, deploys the configuration to resolve dynamic attributes (instance IDs, subnet references), and consolidates the results via `terraform show -json`. If deployment fails because of LocalStack free-tier limitations or provisioning errors, the parser automatically degrades to plan-only mode, which extracts topology and dependencies from symbolic expressions but cannot resolve output variables or computed attributes. This fallback preserves usability while accepting partial semantic completeness, a trade-off that impacts MR1 (Semantic Fidelity). The operational mode is determined at runtime based on the deployment success and does not require manual configuration.

On the mapping side, `TerraformMapper` applies a multi-pass process to reflect Terraform's dependency ordering. It first maps the input variables to TOSCA inputs, then creates primary infrastructure nodes, followed by relationship resources such as associations or attachments, and finally derives TOSCA outputs. This staged algorithm, combined with the `TerraformMappingContext`, supports dependency injection for variable resolution, cross-resource lookup, and distinction between metadata and property contexts, addressing the limitations observed in the Alpha prototype.

Resource translation follows a registry-based strategy pattern. Each AWS resource type (e.g., EC2, VPC, S3) is handled by a dedicated `SingleResourceMapper` registered within the `TerraformMapper` module. During execution, resources are dynamically matched to mappers via capability checks, and unmapped resources revert to generic TOSCA nodes with preserved metadata, ensuring traceability (DP3). Adding a new resource type requires only the definition of a mapper subclass and its registration, confirming the horizontal scalability of the design.

The Terraform plugin validates the main architectural goals of this framework. DP1 (Plugin Architecture) is achieved through compositional reuse: the plugin orchestration logic is inherited from the base classes, minimizing the custom code.

MR2 (extensibility) is demonstrated by the independent and incremental integration of new mappers, requiring minimal effort.

MR3 (verifiability) is ensured by strict protocol conformance verified through static analysis, whereas the dual-source parser increases the semantic completeness of the extracted attributes.

Finally, DP3 (Metadata Preservation) is realized by embedding Terraform-specific attributes and addresses into TOSCA metadata, enabling bidirectional traceability between the source and generated models.

Overall, the Terraform plugin exemplifies the extensibility and rigor of the proposed framework design. Its modular composition, dependency injection mechanisms, and hybrid parsing strategy illustrate how complex IaC translations can be achieved with minimal duplication and high semantic fidelity.

6.2.5 Intervention 9 – Pipeline Orchestration

This intervention transforms the plug-in-based architecture into a working runtime by coordinating multiphase transformations over a shared `ServiceTemplateBuilder` and producing complete TOSCA 2.0 files. Two components orchestrate the workflow: `PipelineRunner`, which executes the plugins sequentially against a single builder instance, and `PluginRegistry`, which manages the plugin discovery and instantiation. The operational flow is illustrated in Figure 6.6.

The pipeline orchestration layer operationalizes the collaborative construction principle by coordinating multiphase translation through a shared `ServiceTemplateBuilder`. This builder instance persists across all plugins, allowing each to enrich the model incrementally by adding nodes, relationships, and metadata produced from distinct IaC technologies. Such collaboration eliminates the need for inter-plugin communication or intermediate serialization while maintaining a unified and continuously validated topology. The orchestration engine, implemented in the `PipelineRunner`, executes a linear workflow: it initializes the shared builder, iterates through registered plug-ins, delegates parsing and mapping tasks, and finally triggers validation and YAML serialization to produce a compliant TOSCA 2.0 service template. This mechanism ensures interoperability and controlled extensibility, thereby fulfilling MR2.

Plugin discovery and instantiation rely on a centralized `PluginRegistry` based on the Registry pattern [63]. Instead of storing concrete instances, the registry maintains references to plugin classes, creating fresh instances on demand. This design decouples the pipeline from specific technologies, allowing new plug-ins to be reg-

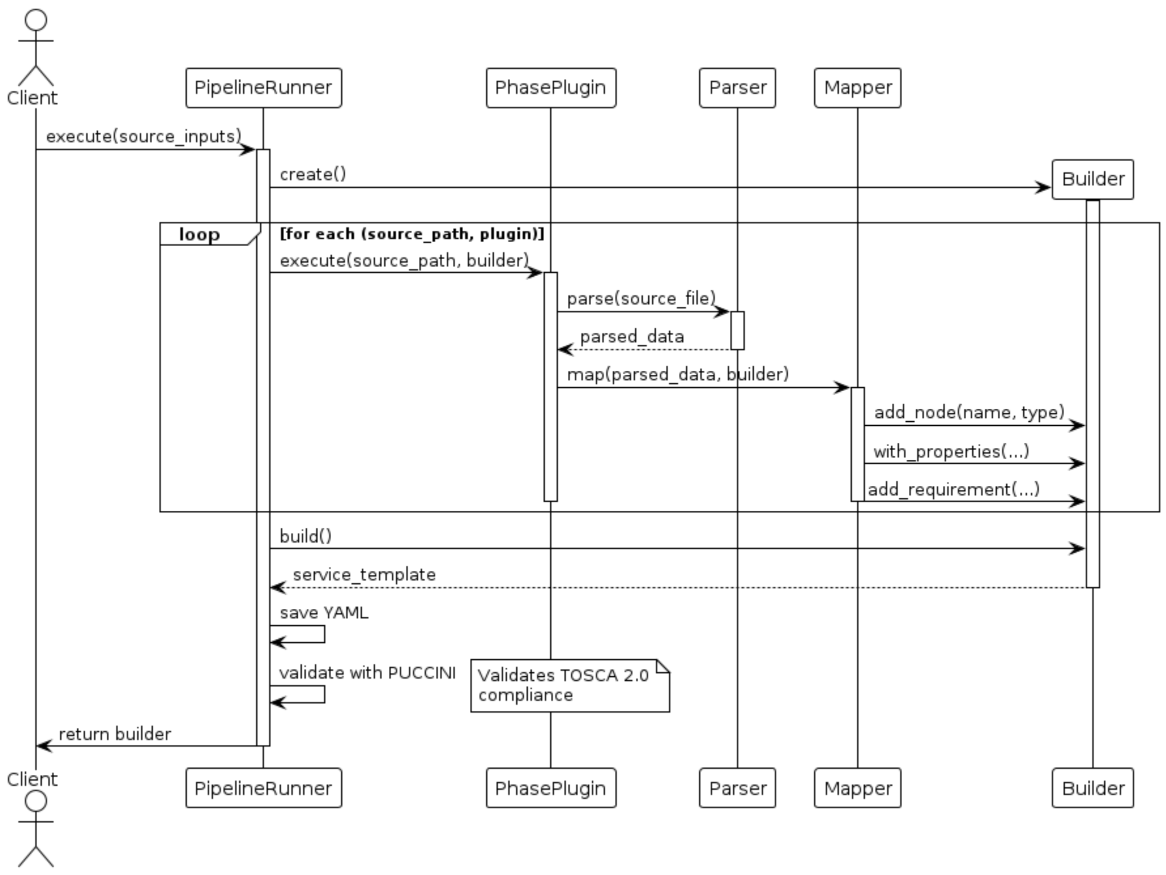


Figure 6.6: The `PipelineRunner` coordinates the end-to-end translation workflow through four main phases: initialization of the shared `ServiceTemplateBuilder`, sequential plugin execution, IaC parsing and mapping into TOSCA structures, and final validation and persistence. Each plugin contributes incrementally to the shared builder, resulting in a fully validated and serialized TOSCA 2.0 service template.

istered dynamically without altering the core infrastructure. Each plugin instance remains isolated and stateless, ensuring testability and preventing interference between the translation phases.

The validation responsibilities are distributed across multiple layers to ensure robustness and early fault detection. Parsers verify the integrity of the source configurations and gracefully degrade when external dependencies, such as LocalStack, are unavailable. The builder layer enforces TOSCA schema compliance via Pydantic validation, detecting structural or type inconsistencies as soon as the nodes are constructed. The runner performs orchestration-level checks on the inputs and outputs, whereas the final conformance is verified through Puccini compilation to confirm standard compliance. Together, these layers implement a balance between fail-fast behavior and best-effort recovery, ensuring resilience, even in partially inconsistent IaC repositories.

The architecture deliberately accepts certain trade offs. Shared mutability in the builder simplifies collaboration at the cost of strict immutability, but the risk is mitigated by restricting the plugins to additive operations. Sequential execution guarantees determinism and reproducibility, favoring clarity over parallelism, as the primary bottleneck remains the IaC tool itself. Finally, the best-effort execution model allows partial translations, preserving progress when individual plug-ins fail, which is a pragmatic choice aligned with real-world infrastructure variability.

In summary, the orchestration layer integrates the protocols, base classes, and builders into a cohesive and extensible translation system. This provides a robust foundation for hybrid reverse engineering, where heterogeneous IaC technologies collaboratively contribute to a unified, validated TOSCA 2.0 model.

6.3 Evaluation of the Beta Artifact

Throughout the development of Beta, the ADR team maintained continuous interaction through weekly meetings and implementation review sessions. The final stage of the second Building, Intervention, and Evaluation (BIE) cycle is the formal assessment of the beta artifact. This evaluation serves to validate architectural refactoring, confirm that the design principles were successfully operationalized, and identify remaining limitations that inform future work.

6.3.1 Evaluation Approach

To ensure a rigorous evaluation, a structured approach was adopted, using a curated set of test cases organized in four progressive phases and clearly defined success criteria linked directly to the Meta-Requirements (MRs) and Design Principles (DPs) established in Chapter 4.

Test Cases

A representative suite of Terraform configurations for AWS was created to test the framework against scenarios of increasing complexity, organized into four phases:

Phase 1: Single Resource Mapping. This phase tested the correctness of the individual AWS resource translations to TOSCA node types. Each resource was defined in isolation using examples from the official Terraform AWS provider documentation[2]. The evaluated resource types included

- **Compute**: `aws_instance` (EC2)
- **Networking (base)**: `aws_vpc`, `aws_subnet`, `aws_security_group`
- **Storage**: `aws_s3_bucket`, `aws_ebs_volume`
- **Database**: `aws_db_instance` (RDS)
- **Networking (connectivity)**: `aws_internet_gateway`, `aws_route_table`
- **Identity**: `aws_iam_role`, `aws_iam_policy`

Phase 2: Inter-Resource Dependencies. This phase validated the relationship inference across the connected resources. The test scenarios included the following:

- **Compute + Subnet**: Verifying `HostedOn` relationship from EC2 instance to VPC subnet
- **Compute + Security Group**: Testing `ConnectsTo` relationships for network access control
- **Compute + Storage**: Validating `AttachesTo` relationship for EBS volume attachment

Phase 3: Terraform Language Constructs. This phase tested the parser's handling of advanced Terraform features.

- **Variables and Outputs**: Mapping Terraform `variable` blocks to TOSCA inputs and `output` blocks to TOSCA outputs
- **Count and Iteration**: Handling `count = 3` to generate multiple distinct node templates with indexed names

[2] `https://registry.terraform.io/providers/hashicorp/aws/latest/docs`

- **Data Sources**: Processing `data "aws_ami"` blocks and excluding them from the topology (non-managed resources)

Phase 4: Multi-Tier Web Application. The final validation deployed a realistic three-tier architecture that represented a production-grade web application with load balancing, caching, and persistent storage. The test configuration comprised 40 Terraform resources organized across the following layers:

- **Presentation Tier**: Application Load Balancer with an HTTP listener, a Target Group, and two EC2 instances distributed across availability zones.
- **Application Tier**: EC2 instances in public subnets with security group rules allowing traffic from the ALB on port 8080.
- **Data Tier**: RDS PostgreSQL database with a DB Subnet Group, and an ElastiCache Redis cluster with a subnet group.
- **Storage Layer**: S3 bucket with versioning and lifecycle policies for backup retention.
- **Networking Infrastructure**: VPC with DNS support, an Internet Gateway, four subnets (two public, two private), a Route Table with associations.
- **Security Configuration**:Four security groups implementing defense-in-depth for the ALB, application, database, and cache layers.
- **DNS Resolution**: A Route53 private hosted zone with an alias record mapping to the ALB endpoint.

Evaluation Criteria

The Beta artifact was assessed against the following criteria:

1. **Correctness and Compliance (MR3):** Whether the generated TOSCA 2.0 templates were syntactically correct and successfully compiled by the Puccini processor across all test phases.
2. **Semantic Fidelity and Intent Preservation (MR1, MR4):** Whether the translation correctly captured architectural intent through: (a) appropriate abstraction to TOSCA node types, (b) accurate relationship inference, and (c) comprehensive metadata preservation of provider-specific attributes.

3. **Architectural Fitness for Extension (MR2, DP1):** Whether the modular architecture successfully addressed the limitations identified in Alpha, assessed through: (a) effort required to add new resource mappers, and (b) architectural readiness for supporting additional IaC technologies.
4. **Builder API Effectiveness (DP5):** Whether the Fluent Builder API eliminated the string-based generation issues from Alpha and improved maintainability.

6.3.2 Evaluation Results and Discussion

The evaluation yielded clear results that validated the architectural refactoring while revealing inherent limitations in abstraction-based translation.

Success in Correctness and Compliance (MR3)

All test cases across the four phases produced TOSCA 2.0 templates that successfully compiled with the extended Puccini processor without errors. This provides definitive empirical validation that the beta artifact can translate complex real-world Terraform configurations into specification-compliant TOSCA models. The Fluent Builder API eliminated the syntax errors observed during the Alpha development, confirming that type-safe construction prevents malformed templates. The multi-tier application test, comprising 40 resources, generated a template with 40 node templates and 58 relationship edges that passed Puccini validation, demonstrating scalability to production-grade infrastructures.

Success in Semantic Fidelity and Intent Preservation (MR1, MR4)

Manual inspection of generated templates confirmed that architectural intent was preserved across all test phases. In the multi-tier application test, the resulting topology correctly reflected the three-tier design pattern: the Application Load Balancer appeared as the public entry point, establishing a `RoutesTo` relationship with a Target Group, which, in turn, distributed traffic to the application instances. The data tier resources (RDS and ElastiCache) were correctly modeled in private subnets, with their isolation and connectivity from the application tier governed by the dependencies established between their respective security groups.

The ADR team noted that relationship inference successfully captured dependencies through three complementary mechanisms: explicit `depends_on` declarations, symbolic references in resource attributes (e.g., `subnet_id`, `vpc_security_group_ids`), and implicit associations (e.g., the `aws_lb_target_group_attachment` resource was correctly translated into `RoutesTo` relationships from the target group to the instances). This validates DP4 (Infer Relationships from Multiple Context Cues).

Metadata preservation was comprehensive for most resources: security group ingress/egress rules, RDS engine versions, ALB listener configurations, and ElastiCache node types were all retained in structured `metadata` blocks. For example, the ALB's `internal = false` attribute and the database's `publicly_accessible = false` setting were preserved, capturing the critical deployment constraints.

However, the team acknowledged a semantic limitation: when the LocalStack deployment failed during Phase 4 (due to free-tier resource limits), the plan-only fallback mode could not resolve computed attributes such as ALB DNS names or database endpoints. This represents an accepted trade-off between operational robustness and semantic fidelity.

Success in Architectural Fitness (MR2, DP1)

The architectural evaluation validated that Beta successfully addressed the extensibility failures of Alpha. During test development, the team demonstrated this by incrementally adding mappers for AWS-specific resources, such as Route53 records and ElastiCache subnet groups. The ability to add support for these distinct resource types without modifying the core framework classes confirms that the registry pattern and protocol-based abstraction enable independent, non-invasive extensions.

The ADR team conducted a structured architectural review to assess readiness for supporting additional IaC technologies. Through guided walkthroughs, they examined how a hypothetical Ansible plugin would be implemented, identified the required components (YAML playbook parser, task-to-node mappers, and inventory context manager), and confirmed that the protocol contracts and base classes provided sufficient scaffolding. While the Terraform plugin remains the sole empirical instantiation, the architectural patterns it exemplifies—protocol conformance, registry-based dispatch, and base class reuse— have been validated as generalizable through expert consensus.

This represents a fundamental improvement over Alpha, where adding Ansible support would require a complete rewrite of the monolithic prototype. Beta's layered architecture localizes technology-specific logic and enables horizontal scaling via plug-ins.

Identified Limitations: Simple Profile Expressiveness

The evaluation revealed a fundamental constraint that affects semantic fidelity: the TOSCA Simple Profile lacks native representations for many AWS-specific constructs. To bridge this gap, the translator adopted a strategy of mapping provider-specific resources to the semantically closest abstract type available in Simple Profile. For instance, an `aws_lb_target_group` was mapped to `LoadBalancer` and an `aws_elasticache_cluster` was abstracted to `DBMS`.

In cases where no semantically related type existed, the mapping fell back to the most generic type, `Root`. This was the case for critical resources such as `aws_security_group`, whose firewall-like semantics have no direct equivalent in the Simple Profile.

Furthermore, some AWS resources that represent configuration or placement constraints rather than deployable components, such as `aws_db_subnet_group`, were modeled not as nodes but as TOSCA `Placement` policies. While this ensures that no information is lost (MR4) by preserving the details in the metadata, the resulting TOSCA models sacrifice semantic precision for portability.

This limitation is inherent to abstraction-based translation: achieving complete semantic equivalence between a provider-specific DSL and a vendor-neutral standard is theoretically constrained by the expressiveness gap between the metamodels. The ADR team recognized this as a design trade-off rather than an implementation failure; TOSCA's portability comes at the cost of reduced semantic granularity for provider-specific features.

6.3.3 Reflections and Learning from Cycle 2

The evaluation of the Beta artifact validates the core architectural thesis of Cycle 2: systematic modularization through protocol-based abstraction and fluent construction APIs enables extensible, maintainable IaC-to-TOSCA translation at scale.

The functional success across all test phases, culminating in a realistic multi-tier application, provides empirical evidence that deterministic rule-based translation can handle production-grade infrastructures while preserving semantic intent. The architectural review confirmed that DP1 (Plugin Architecture) and DP5 (Fluent Builder API) were successfully operationalized, transforming the brittle Alpha prototype into a framework capable of supporting multiple IaC technologies.

However, the evaluation also surfaces a fundamental challenge: the limited expressiveness of the TOSCA Simple Profile constrains semantic fidelity for provider-specific constructs.

Therefore, the primary contributions of Cycle 2 are: (1) an empirically validated modular architecture that addresses Alpha's extensibility failures; (2) a comprehensive Terraform plugin demonstrating scalability to complex topologies; and (3) identification of the Simple Profile expressiveness gap as a fundamental constraint requiring ecosystem-level solutions.

These findings provided a foundation for Cycle 3. The next cycle will shift to an experimental phase, investigating AI-driven translation as an alternative to the deterministic pipeline. The purpose is purely exploratory: to assess whether an AI-based approach can significantly reduce the effort required to build an exhaustive plugin for each new IaC technology to be supported.

CHAPTER 7

Building, Interventions and Evaluation: Cycle 3

Cycle 3 constitutes the final and most exploratory iteration of the Action Design Research process, focusing on the potential of AI-augmented approaches for IaC-to-TOSCA translation. The motivation stemmed from Cycle 2's findings: supporting only 31 AWS resources in Terraform required approximately 14,000 lines of handcrafted code, revealing the scalability limits of purely deterministic mappings. This prompted an investigation into whether Large Language Models (LLMs) enhanced with Retrieval-Augmented Generation (RAG) could reduce manual engineering effort while maintaining acceptable semantic accuracy.

The Gamma artifact was implemented as a set of exploratory Jupyter-based prototypes, emphasizing technical feasibility over production. Owing to limited time and resources, the evaluation was strictly qualitative, aiming to observe whether AI-assisted translation could produce meaningful mappings at all.

Preliminary experiments indicated that basic resources, excluding relationships and more complex dependencies, can be successfully mapped without relying on large deterministic mappers. However, several issues have emerged, primarily because of the limitations of the underlying language models. Consequently, the current results do not allow firm conclusions regarding the flexibility or scalability of the AI-based approach. Moreover, time constraints prevented extensive or systematic testing, leaving the overall potential of this strategy open for future studies.

7.1 Design of the Gamma Artifact

This section presents the design of the Gamma artifact, the exploratory AI-augmented translation system developed in Cycle 3. Following the architectural reflections from Cycle 2, which revealed the maintenance burden of deterministic mappings, the ADR team shifted its focus toward investigating whether Large Language Models enhanced with Retrieval-Augmented Generation could reduce manual engineering effort while preserving semantic fidelity. The design emphasizes technical feasibility over production integration, deliberately maintaining separation from the validated beta framework to enable rapid experimentation without compromising established contributions.

7.1.1 Motivation and Transition from Deterministic to AI-Augmented Translation

The evaluation of the Beta artifact in Cycle 2 confirmed the technical viability and architectural soundness of deterministic rule-based translation. However, a critical scalability concern emerged during the implementation review: supporting only 31 AWS resource types required approximately 14,000 lines of handcrafted Python code, including dedicated SingleResourceMapper implementations for each resource type. Extrapolating this effort to comprehensive AWS coverage—over 400 resource types in the Terraform AWS provider alone— reveals an unsustainable maintenance burden. Each new resource requires a manual analysis of the Terraform documentation, semantic mapping decisions, property type coercion logic, and integration testing. This "linear scaling problem" motivated the investigation of alternative approaches that could reduce per-resource engineering effort through automated reasoning.

The research questions driving Cycle 3 were specifically RQ2 (To what extent can Large Language Models augmented with RAG support automated IaC-to-TOSCA translation?) and preliminary exploration of RQ3 (What are the observed trade-offs between deterministic and AI-augmented approaches?). The objective was not to replace the validated deterministic pipeline but to assess whether AI-based methods could complement it, potentially handling straightforward resource mappings, while deterministic rules address complex relationship inference and edge cases.

The scope of this cycle was explicitly exploratory and qualitative in nature. Given the novelty of applying RAG-enhanced LLMs to IaC-to-TOSCA translation—a problem space with no prior academic precedent—the ADR team adopted a feasibility-first approach focused on answering fundamental questions: Can LLMs correctly classify Terraform resources into TOSCA types when constrained by specification knowledge? Can they generate syntactically valid TOSCA templates without hallucinating non-existent properties? Under what conditions do AI-augmented approaches fail? The evaluation prioritized the identification of boundaries and failure modes over quantitative performance metrics. The decision to develop Gamma as a parallel exploratory prototype rather than an integrated Beta extension was strategic. Jupyter notebooks provide rapid iteration cycles essential for experimenting with different LLM models, RAG architectures, and prompt engineering strategies without disrupting the production-ready beta framework. This separation preserved Beta's stability as a baseline for future comparative evaluation while limiting risk; if AI-augmented translation proved infeasible, the deterministic framework remained intact. Conversely, if Gamma demonstrates sufficient maturity, its components can be retrofitted into Beta's plugin architecture (MR4 - Support for Hybrid Strategies), leveraging the protocol contracts designed precisely for this extensibility.

7.1.2 Intervention 10 - TOSCA Knowledge Graph Construction

The first intervention aimed to address a fundamental challenge in AI-augmented translation: grounding LLM decisions in authoritative specification knowledge to prevent hallucination of non-existent TOSCA types or properties. While LLMs demonstrate impressive natural language understanding, they are prone to "confabulating" plausible-sounding but incorrect technical details when operating without external knowledge constraints. This risk is particularly acute for IaC-to-TOSCA translation because the LLM must select from a finite set of valid TOSCA types defined in the Simple Profile; however, these types are underrepresented in the model training data compared to mainstream programming languages. Without explicit grounding, the LLM might invent seemingly reasonable but non-standard types such as `WebServer` or `DatabaseCluster` that do not exist in the Simple Profile.

The proposal for this intervention was to construct a queryable Neo4j Knowledge Graph encoding the complete TOSCA Simple Profile as a graph database. Importantly, this intervention focused specifically on the Simple Profile, a community-maintained collection of reusable types and modeling idioms, rather than the core TOSCA 2.0 specification. This design decision was motivated by TOSCA's inherently graph-structured semantics: node types are derived from parent types through inheritance chains, capabilities define interaction points between nodes, and requirements specify dependencies that must be satisfied by matching capabilities. A graph database naturally represents these relationships, enabling traversal queries like "find all properties inherited by `Compute` from its ancestor types" or "discover which node types can satisfy a `host` requirement." The related design principles were DP6 (Represent Domain Knowledge as a Queryable Graph) and DP7 (Multi-Source RAG), linked to meta-requirements MR1 (Semantic Fidelity) and MR3 (Verifiability).

The knowledge graph design distinguishes between type definitions (abstract specifications such as `CapabilityType`) and type instances (concrete definitions such as a specific capability defined within a node type). This separation enabled precise semantic queries: "Which capability types exist in the Simple Profile?" versus "Which capability definitions does the `Compute` node expose?" The graph schema comprised eight primary node types (`NodeType`, `CapabilityType`, `RelationshipType`, `DataType`, `ArtifactType`, `InterfaceType`, `PolicyType`, `GroupType`) and nine relationship types modeling inheritance (`DERIVED_FROM`), composition (`HAS_PROPERTY`, `DEFINES_CAPABILITY`), and constraint relationships (`REQUIRES_CAPABILITY`, `VALID_FOR_CAPABILITY`).

A critical design feature was the structured query interface, which included pre-built Cipher functions encapsulating common traversal patterns. For example, `find_node_hierarchy(node_type)` returns the complete inheritance chain from a specific type to its root ancestor, enabling the LLM generation chain to access all inherited properties without manual-traversal logic. Similarly, `get_type_with_full_context(node_type)` aggregates the properties, capabilities, and requirements from the entire inheritance hierarchy into a single structured response, providing the LLM with comprehensive type information in its generation context.

This intervention was chosen for implementation as the foundational component of Gamma because it operationalizes the core architectural principle of symbolic-neural hybridization: combining precise specification knowledge (symbolic) with flexible semantic reasoning (neural). The expected outcome was constraint-based LLM classification, where type selection was limited to valid enumeration rather than unconstrained generation.

7.1.3 Intervention 11 - Terraform Documentation Vector Store

The second intervention addressed the complementary challenge of providing the LLM with authoritative knowledge about the source domain—Terraform resource semantics. While Intervention 10 grounded the system in TOSCA-type definitions, successful translation requires understanding what each Terraform resource means in infrastructure terms: Does `aws_instance` represent a virtual machine, container, or serverless function? What capabilities does `aws_s3_bucket` provide—block storage, object storage, or file system? Without access to official Terraform documentation, the LLM must rely solely on its training data, which may contain outdated provider schemas, community discussions with conflicting information, or incomplete coverage of recently added resource types.

The proposal for this intervention was to construct a Vector Store with Semantic Search containing the complete AWS provider documentation from the official Terraform registry. This design leveraged Retrieval-Augmented Generation (RAG) principles: rather than expecting the LLM to "memorize" hundreds of resource specifications, the system dynamically retrieves relevant documentation sections at translation time, injecting this context into the LLM prompt. The related design principle was DP7 (Augment Generation with Multi-Source, Reranked Context), linked to meta-requirement MR1 (Semantic Fidelity) through authoritative grounding, and MR4 (Support for Hybrid Strategies) through complementary knowledge sources.

A critical design decision concerned the document chunking strategy. Naïve approaches, such as fixed-size character windows or paragraph-based splitting, risk fragmenting semantic units or losing hierarchical context. The intervention adopted markdown header-based segmentation: documentation was split at heading boundaries (`##`, `###`), preserving logical sections such as "Arguments," "Attributes Reference," or "Import." Each chunk was enriched with metadata capturing the resource type (extracted from "Resource: `aws_xxx`" headers) and hierarchical position (`h1`, `h2`, and `h3` headers), enabling precise filtering during retrieval. Recursive character splitting with 1200-character chunks and 150-character overlap then subdivided oversized sections while maintaining cross-boundary continuity.

The embedding strategy employed `google/embeddinggemma-300m`, a model specifically pretrained on code and technical documentation, offering superior semantic understanding of infrastructure terminology compared to general-purpose sentence transformers. A dual-prompt configuration optimized embeddings: query prompts framed search intent ("Documentation for `aws_instance` Terraform resource"), whereas document prompts optimized content representation, improving retrieval precision.

The retrieval design implements a two-stage pipeline to balance recall and precision. First, dynamic resource type extraction parsed the input Terraform code with regex to identify the mentioned resource types (`aws_instance`, `aws_vpc`), enabling targeted retrieval rather than generic semantic search. Second, a broad semantic search (k=20) with metadata filters (`resource_type == "aws_instance"`) cast a wide net, followed by cross-encoder reranking (`BAAI/bge-reranker-large`) to the top-4 most relevant chunks. This two-stage approach mitigated the "lost in the middle" problem, where LLMs ignore mid-context information, ensuring that only the most salient documentation reached the generation phase.

This intervention was chosen for implementation because it operationalized the multi-source RAG principle: TOSCA knowledge (Neo4j) provides the target schema, Terraform documentation (Vector Store) provides source semantics, and the LLM synthesizes mappings between them. The expected outcome was context-aware classification, where type selection reflected both the Terraform resource purpose and TOSCA abstraction philosophy.

7.1.4 Intervention 12 - LLM-Based Classification Chain

With the knowledge sources established, the third intervention addressed the core reasoning task: mapping a given Terraform resource to its most appropriate semantic equivalent in the TOSCA Simple Profile. This is a classification problem and not a simple string match. For example, `aws_instance` must be mapped to `Compute`, which requires architectural understanding. This intervention proposed an LLM-based classification chain designed to perform this semantic leap, acting as a critical bridge between the source and target knowledge domains. The design was guided by DP8 (Enforce Correctness and Compliance via Structured Output) and aimed to fulfill MR1 (Semantic Fidelity) and MR3 (Verifiability).

The primary design challenge was to constrain the vast generative capabilities of the LLM to a specific, well-defined decision space. An unconstrained prompt might lead the LLM to hallucinate a non-existent but plausible-sounding type, such as VirtualMachine. To prevent this, the chain was designed to provide the LLM with three crucial inputs:

1. The source Terraform resource code block.
2. The contextually relevant documentation retrieved from the Vector Store (Intervention 11).
3. A manually curated, static list of all valid NodeType names, extracted directly from the official TOSCA Simple Profile documentation.

This third input is a critical design choice for the Gamma prototype. Although the Neo4j Knowledge Graph (Intervention 10) contains this information, a dynamic query was deemed an unnecessary complexity for this exploratory phase. Using a static list provides a direct and reliable method to test the core hypothesis—that an LLM's classification can be successfully constrained—without introducing dependencies on the graph query interface.

A key design decision was to enforce structured output using a Pydantic model. The LLM was not asked to return a simple string but to populate a ToscaTypeSelection object containing the chosen `resource_name`, its category (e.g., `"node_types"`), and, critically, a reasoning field. This forces the LLM to justify its selection in technical terms, providing invaluable explanations for debugging and validation.

Prompt engineering for this chain instructed the LLM to act as an expert cloud architect, analyzing the provided documentation and code to infer the architectural role of the resource. Choosing a type not present in the provided static list of valid TOSCA types was explicitly forbidden. This transforms the LLM from a free-form text generator to a constrained reasoning engine.

This intervention was essential because it separated the problem of what a resource is (classification) from how it is configured (generation). The expected outcome was a validated, machine-readable ToscaTypeSelection object that could be confidently passed to the subsequent generation chain, ensuring that the next step would be grounded in a correct and specification-compliant type selection process.

7.1.5 Intervention 13 - LLM-Based TOSCA Generation Chain

Following the successful classification of a Terraform resource to a specific TOSCA type, the final intervention focused on generating the complete, syntactically correct TOSCA node template. This moves from high-level classification to low-level implementation, mapping specific Terraform arguments (e.g., `instance_type`, `ami`) to the corresponding TOSCA properties (e.g., `num_cpus`, `os`) and preserving any non-mappable details. The design heavily relied on DP6 (Represent Domain Knowledge as a Queryable Graph) and DP3 (Retain Provider Specifics as Structured Metadata) to meet MR1 (Semantic Fidelity), MR3 (Verifiability), and MR4 (Abstraction–Specificity Balance).

The core of this intervention was the design of an "inheritance-aware" generational context. A naive approach would provide the LLM with only the direct properties of the chosen TOSCA type (e.g., `Compute`), ignoring the rich set of properties inherited from its ancestors (`Root` and `Abstract.Compute`). This would inevitably lead to hallucinations or incomplete templates, as the LLM would be unaware of valid inherited properties such as `private_address`.

To solve this, the generation chain was designed to use the output of the classification chain (for example, `Compute` to execute the `get_type_with_full_context` query against the Neo4j Knowledge Graph. This single query returns a comprehensive JSON object detailing the chosen type, including the following:

- Its full inheritance chain.
- A consolidated list of all properties, both direct and inherited.
- A complete list of its capabilities and requirements.
- The detailed schemas of those capabilities and requirements.

This rich, structured context was injected directly into the LLM's prompt, alongside the original Terraform code. The prompt engineering strategy instructed the LLM to generate a YAML node template that strictly adhered to the proposed schema. It was tasked with semantically mapping Terraform arguments to the available TOSCA properties and, crucially, placing all unmappable or provider-specific arguments (such as ami or `instance_type`) into a metadata block. This directly implements the lossless translation principle required by MR4.

The output was designed as a clean, modular YAML snippet representing a single node template. This makes the output composable, allowing an orchestrator to assemble multiple generated nodes into a complete `service_template`.

This intervention represents the culmination of the design of the Gamma artifact. By leveraging the constrained classification from Intervention 12 and the complete, inheritance-aware context from Intervention 10, TOSCA templates that were not only syntactically correct but also semantically faithful to both the source IaC and the target specification were produced. The expected outcome was a verifiable TOSCA node template that demonstrated the potential of a hybrid symbolic-neural approach for complex code generation tasks.

7.2 Implementation of Interventions

This section details the practical implementation of the Gamma artifact, translating the design principles and architectural components from Section 7.1 into a functional prototype. The implementation was conducted in a Python-based environment, primarily using Jupyter Notebooks to facilitate rapid iteration and experimentation. The following subsections describe the technical realization of each intervention, detailing the key libraries, data flows, and code structures that constitute the AI-augmented translation pipeline.

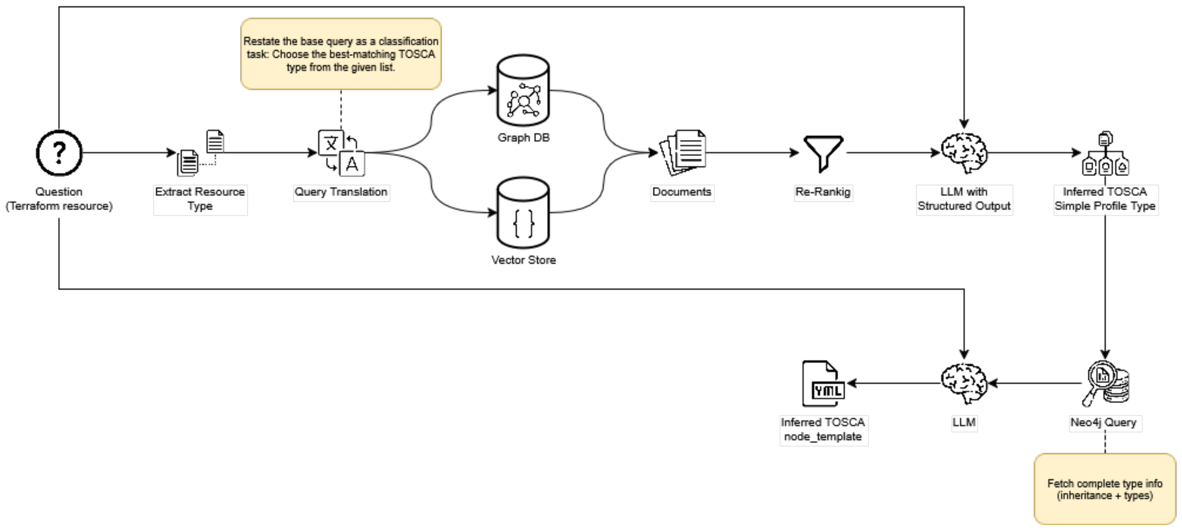

Restate the base query as a classification task: Choose the best-matching TOSCA type from the given list.
Question (Terraform resource)
Extract Resource Type
Query Translation
Graph DB
Vector Store
Documents
Re-Rankig
LLM with Structured Output
Inferred TOSCA Simple Profile Type
Neo4j Query
Fetch complete type info (inheritance + types)
LLM
Inferred TOSCA node_template

Figure 7.1: Two-Stage AI-Augmented Translation Pipeline (Gamma). Top: the Classification Chain uses RAG over Terraform docs and a Neo4j-derived list of valid types to allow an LLM to select the correct TOSCA Simple Profile NodeType. Bottom: the Generation Chain queries Neo4j for the type's inheritance-aware schema, which constrains a second LLM to emit a compliant TOSCA node$_{t}emplate$.

7.2.1 Intervention 10 - TOSCA Knowledge Graph Construction

The implementation of the TOSCA Knowledge Graph translated the design principles from Section 7.1.4 into a functional Neo4j-based system capable of representing the complete TOSCA Simple Profile as a queryable graph structure. The implementation comprised three core components: YAML parsing and normalization, graph population algorithms, and structured query function.

YAML Parsing and Normalization

The implementation began with a flexible document acquisition mechanism capable of ingesting TOSCA profile definitions. This design decision reflected the distributed nature of the TOSCA Simple Profile, which is maintained as a collection of separate YAML files in the community repository[1]. The `parse_yaml_file()` function implements a dual-mode parser: for URLs, it leverages the `requests` library to download content before parsing; for local paths, it performs direct file system access.

[1]`www.github.com/oasis-open/tosca-community-contributions`

This approach enabled the seamless integration of official TOSCA specifications without manual file management, supporting the exploratory iteration cycles essential to the Gamma prototype.

Graph Population Strategy

The graph construction followed a phased population algorithm designed to respect dependency ordering across TOSCA type definitions. The implementation processed eight type categories in sequence: primitive data types, artifact types, capability types, interface types, relationship types, node types, policy types, and group types. This ordering was critical because later phases depend on earlier ones; for example, node type properties reference data types, and requirements reference capability types. Processing out of order creates dangling references or requires complex two-pass algorithms. Each type category was handled by a dedicated processing function (`process_node_types()`,`process_capability_types()`,etc.) that implemented a consistent three-step pattern: (1) create or retrieve the type node using `get_or_create_node()`, (2) establish inheritance relationships via `DERIVED_FROM` edges if a parent type exists, and (3) create composition relationships for nested elements, such as properties, capabilities, and requirements. The `get_or_create_node()` transaction function implements idempotent insertion: it first queries existing nodes matching the type and name, only creating new nodes if none exist. This ensured that repeated imports or incremental updates did not duplicate the graph elements. A critical implementation detail is the property and capability scoping. Rather than storing properties as simple attributes within type nodes, the implementation created dedicated `Property` and `CapabilityDef` nodes linked via `HAS_PROPERTY` and `DEFINES_CAPABILITY` relationships. The property node names were scoped to their parent type (e.g., `Compute.num_cpus`), enabling precise attribution during the inheritance chain traversal. This design choice aligned with the graph database principle of "nodes for entities, relationships for connections," facilitating complex queries like "find all properties inherited by `Database` from its ancestors."

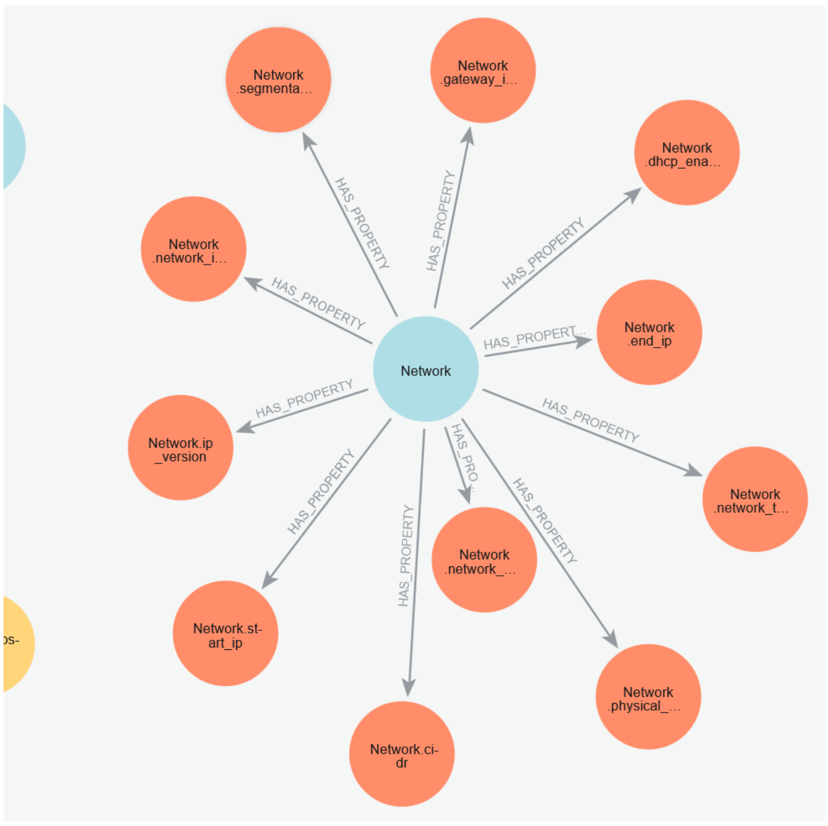


Figure 7.2: Neo4j visualization of the TOSCA Knowledge Graph showing the `Network` node type (center) with its `HAS_PROPERTY` relationships to property nodes (orange) and inheritance connections. The graph encodes 234 nodes representing the complete TOSCA Simple Profile type system, enabling structured queries for LLM context enrichment.

Structured Query Interface

The implementation provided four pre-built query functions that encapsulated common traversal patterns identified during the design phase. The `find_node_hierarchy()` function leverages Neo4j's recursive path matching (`-[:DERIVED_FROM*]-`) to return the complete inheritance chain from a given type to its root ancestor. The `get_type_with_full_context()` function, which is critical to the generation chain, implements multi-hop aggregation: it traverses the inheritance hierarchy, collects all `HAS_PROPERTY` relationships at each level, and returns a consolidated JSON object containing the direct and inherited properties, capabilities, and requirements.

This single query eliminated the need for the LLM generation chain to understand graph traversal logic, instead receiving a complete, flattened type schema that was ready for prompt injection.

Validation and Statistics

The implementation included two validation mechanisms to ensure graph completeness. First, `create_constraints()` establishes uniqueness constraints on the `name` property for all type labels, preventing duplicate type definitions that would violate TOSCA semantics. Second, `get_statistics()` provides a summary count of nodes by label, enabling a quick verification that all expected type categories have been populated. Processing the nine TOSCA Simple Profile YAML files successfully populated the Neo4j database with 234 nodes distributed across 13 labels: 16 `NodeType` nodes (including fundamental types such as `Compute`, `Database`, `WebServer`), 13 `CapabilityType` nodes, 8 `RelationshipType` nodes, 25 `DataType` nodes (primitives and complex types), 78 `Property` nodes representing the complete property space across all types, 19 `CapabilityDef` nodes modeling concrete capability instances, and 9 `Requirement` nodes capturing dependency specifications. This distribution confirmed the successful ingestion of the complete Simple Profile specification, providing a comprehensive knowledge base containing both type taxonomy (node/capability/relationship types) and structural semantics (properties, attributes, requirements) essential for grounding the LLM classification and generation chains. The implementation deliberately prioritizes simplicity over optimization in this exploratory phase. Transaction functions create individual database connections per operation rather than batching writes, and graph traversal queries use unoptimized Cipher patterns without index hints. While production systems would require performance tuning for large-scale type libraries, these trade-offs enable rapid iteration and transparent debugging, which are essential for validating the symbolic-neural architecture proposed in the Gamma design.

7.2.2 Intervention 11 - Terraform Documentation Vector Store

This intervention implements a semantic retrieval layer over the official Terraform AWS provider documentation, enabling the AI pipeline to ground its decisions in authoritative, up-to-date resource semantics. The implementation covers four parts: (1) document acquisition and loading, (2) markdown-aware splitting with metadata enrichment, (3) domain-specific embeddings, and (4) persistent vector store construction.

Document Acquisition

Document Acquisition and Dual-Source Strategy A locally maintained corpus of ∼1500 Markdown files was scraped from the official Terraform AWS Provider Registry. The corpus was ingested with a DirectoryLoader, providing fast and deterministic access without network dependencies during experimentation.

Markdown-Aware Splitting & Metadata

To preserve structure and improve retrieval precision, documents were segmented with `MarkdownHeaderTextSplitter` at `#`, `##`, and `###` boundaries. Each segment carries `h1`, `h2`, and `h3` metadata (e.g., "Arguments," "Attributes Reference"), keeping the

A regex pass extracts the canonical resource identifier from the top-level header pattern Resource: `aws_xxx`.

```
RESOURCE_RE = re.compile(r"^\s*Resource:\s*([A-Za-z0-9_.:-]+)\s*$")
```

The captured value is stored as `resource_type` metadata (e.g., `aws_instance`). This enables precise filtering at query time, for example, `where={"resource_type": "aws_instance"}`—which significantly reduces irrelevant context.

Oversized sections are further chunked using RecursiveCharacterTextSplitter (size=1200, overlap=150). This two-stage approach maintains semantic coherence via headers while keeping chunks within embedding-friendly length. Processing ∼1500 files yields ∼15k enriched chunks (header metadata, source path, `resource_type`).

Embeddings

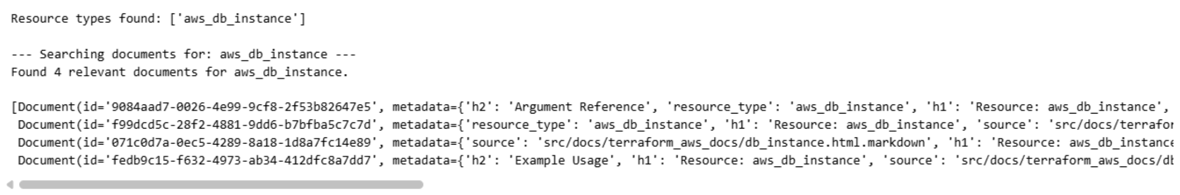
Embeddings are generated with `google/embeddinggemma-300m`, a model pre-trained on code and technical documentation. A dual-prompt configuration improves asymmetric search: queries use `prompt_name="query"`; documents use `prompt_name="document"`. GPU acceleration and batching (`batch_size=32`) reduced the embedding time.

```
embeddings = HuggingFaceEmbeddings(
    model_name="google/embeddinggemma-300m",
    model_kwargs={"device": device},
    query_encode_kwargs={"prompt_name": "query", "batch_size": 32},
    encode_kwargs={"prompt_name": "document", "batch_size": 32}
)
```

Vector Store

ChromaDB is used as the persistent vector backend. It builds an HNSW index for approximate nearest-neighbor search and persists to `./chroma_store`. Subsequent sessions load the store directly, avoiding re-embedding and reducing startup latency from minutes to seconds.

The vector store provides high-precision, resource-scoped retrieval for Terraform AWS documentation. During translation, the pipeline can inject top-ranked passages—filtered by `resource_type` and reranked as needed—into prompts, ensuring that the LLM's decisions (both in the classification and generation stages) are grounded in the exact semantics of the referenced AWS resource.



```
Resource types found: ['aws_db_instance']

--- Searching documents for: aws_db_instance ---
Found 4 relevant documents for aws_db_instance.

[Document(id='9084aad7-0026-4e99-9cf8-2f53b82647e5', metadata={'h2': 'Argument Reference', 'resource_type': 'aws_db_instance', 'h1': 'Resource: aws_db_instance',
 Document(id='f99dcd5c-28f2-4881-9dd6-b7bfba5c7c7d', metadata={'resource_type': 'aws_db_instance', 'h1': 'Resource: aws_db_instance', 'source': 'src/docs/terrafor
 Document(id='071c0d7a-0ec5-4289-8a18-1d8a7fc14e89', metadata={'source': 'src/docs/terraform_aws_docs/db_instance.html.markdown', 'h1': 'Resource: aws_db_instanc
 Document(id='fedb9c15-f632-4973-ab34-412dfc8a7dd7', metadata={'h2': 'Example Usage', 'h1': 'Resource: aws_db_instance', 'source': 'src/docs/terraform_aws_docs/d
```

Figure 7.3: RAG retrieval debug output for `aws_db_instance`: after extracting the resource type, the system finds 4 relevant Terraform AWS docs and returns the top chunks with hierarchical metadata `(h1/h2/h3)`, `resource_type`, and `source path`, ready for LLM prompting.

7.2.3 Intervention 12 - LLM-Based Classification Chain

The implementation of the LLM-based classification chain translated the constraint-based reasoning architecture into a functional pipeline capable of mapping arbitrary Terraform resources to semantically appropriate TOSCA types. The implementation comprised three interconnected components: dynamic Pydantic model generation for structured output enforcement, intelligent prompt construction with multisource context injection, and LLM invocation with validation.

Dynamic Structured Output Schema Generation

A critical design challenge was constraining the LLM to select from the exact set of valid TOSCA types present in the Simple Profile without hardcoding type lists that would become stale as the specification evolved. The implementation solved this through runtime schema generation: the `create_pydantic_model_for_tosca_types()` function consumed the 99 unique types extracted during vector store construction (Section 7.2.2) and dynamically constructed a `ToscaTypeSelection` Pydantic model with three fields—`category` (e.g., `"node_types"`), `resource_name` (e.g., `"Compute"`), and `reasoning` (technical justification). This approach operationalized DP8 (Structured Output) by ensuring that the classification schema remained synchronized with the actual specification content, rather than relying on manually maintained enumerations.

The Pydantic model served two purposes: (1) it provided explicit type hints and field descriptions that LangChain injected into the LLM system prompt, guiding the output structure, and (2) it enabled automatic validation and parsing of the LLM response, raising exceptions if the model returned malformed JSON or omitted required fields. This validation layer prevents downstream generation failures caused by classification errors, which is an essential robustness guarantee for the exploratory prototype.

```
class ToscaTypeSelection(BaseModel)
    category: str = Field(
        description="The category of the selected TOSCA type (e.g.,
         ↪  'node_types')."
    )
    resource_name: str = Field(
        description="The name of the selected TOSCA type (e.g.,
         ↪  'Compute')."
    )
    reasoning: str = Field(
        description="A brief explanation for why this specific
         ↪  TOSCA type was chosen."
    )
```

Multi-Source Context Assembly and Prompt Engineering

The classification prompt architecture implemented the grounded classification principle articulated in the design phase, providing the LLM with three critical information sources. First, the complete enumeration of valid TOSCA types was serialized as formatted JSON and injected into the prompt via the `{valid_choices}`. This explicit enumeration, combined with the instruction "You must choose from the list," established a hard constraint boundary for the LLM's decision space.

Second, the original Terraform resource code block was inserted via `{input_terraform}`, which supplied concrete syntactic and semantic details about the resource being classified. The prompt employed HCL syntax highlighting (` ```hcl `) to signal the code context to the LLM, leveraging its code-specialized training to parse resource types, argument names, and structural patterns.

Third, and most critically, the prompt incorporated the retrieved documentation context via the `{extra_context}` placeholder. This context was populated by querying the vector store (from Intervention 11) with a semantic search: given a Terraform resource block, the system extracted the resource type (e.g., `aws_instance`), queried ChromaDB with `where={"resource_type": "aws_instance"}`, retrieved the top-4 semantically relevant chunks via cosine similarity, and concatenated their text content. This retrieval-augmented approach ensured that the LLM's classification decision was informed by authoritative AWS provider documentation describing the resource's purpose, capabilities, and typical use cases—precisely the semantic knowledge required to map provider-specific constructs to vendor-neutral TOSCA abstractions.

The prompt engineering strategy emphasized role-based instruction: the system message established the LLM as "a TOSCA and Terraform expert", priming it to reason about infrastructure semantics rather than generate generic text. The human message employed structured information presentation—valid choices in JSON blocks, documentation in plain text blocks, and Terraform code in HCL blocks—exploiting the LLM's ability to parse and reason over multi-format inputs.

LLM Configuration and Invocation

The implementation employed Gemini 2.5 Flash as the classification LLM, a strategic choice motivated by three factors: (1) native structured output support via LangChain's `with_structured_output()` method, eliminating the need for custom JSON parsing logic; (2) generous context window (1,048,576 input tokens), accommodating the complete type enumeration plus extensive documentation chunks without truncation; and (3) low latency (2-4 seconds per classification), enabling rapid iteration during exploratory development. The model's maximum output token limit (65,535) far exceeded the classification requirements ($\sim$200 tokens per structured response), ensuring that response truncation never occurred.

The `classification_chain` object constructed via LangChain's pipe operator (`|`) composed the prompt template and structured the LLM into a single invocable unit. Execution proceeded through three stages: (1) the prompt template populated placeholders with actual values (type list, context, Terraform code), (2) the resulting string was transmitted to the Gemini API, and (3) the structured output wrapper parsed the JSON response into a validated `ToscaTypeSelection` instance. This functional composition pattern enables a clean separation between prompt logic and model invocation, facilitating prompt engineering experiments without modifying the LLM configuration.

7.2.4 Intervention 13 - LLM-Based TOSCA Generation Chain

The implementation of the TOSCA generation chain operationalizes the inheritance-aware template synthesis architecture, transforming validated type selections and enriched schema contexts into specification-compliant YAML node templates. The implementation comprised three interconnected components: comprehensive Neo4j schema retrieval with type detail aggregation, structured context formatting for LLM consumption, and constraint-enforced YAML generation, as described below.

Inheritance-Aware Schema Retrieval from Neo4j

The critical innovation in the generation chain was multi-level schema aggregation that provided the LLM with complete visibility into the TOSCA type's full capabilities. The `get_type_info_from_graph()` function implements a single complex Cipher query that: (1) traverses the entire inheritance chain from the selected type to its root ancestor via recursive `-[:DERIVED_FROM*]-` pattern matching; (2) collects all properties, attributes, capabilities, and requirements defined at each inheritance level; (3) deduplicates inherited elements by name while preserving provenance metadata (`inherited_from` field); and (4) returns a consolidated data structure containing both direct and inherited schema elements. For example, querying `Database` (which derives from `DBMS` → `SoftwareComponent` → `Root`) returns properties from all four levels, enabling the LLM to leverage inherited properties such as `private_address` (from `Root`) alongside database-specific properties such as `port` (from `DBMS`).

A complementary function, `get_capability_and_requirement_type_details()`, implements type-detailed enrichment by executing secondary queries for each `CapabilityType` and `RelationshipType` referenced in the node's capabilities and requirements. This resolved the semantic gap, where base schema retrieval returned capability names (e.g., `host: Container`) but not the internal structure of the capability type. The enrichment queries retrieved capability type properties, relationship type properties, and valid capability constraints, providing the LLM with complete nested definitions. This enabled the correct usage of complex TOSCA structures, such as requirements with typed relationship properties, a pattern that frequently causes generation errors in preliminary tests without type enrichment.

Structured Context Formatting for LLM Prompts

The `build_complete_context_with_types()` function transformed the Neo4j query results into a hierarchically formatted text representation optimized for LLM comprehension. Rather than injecting raw JSON, the formatter produces indented, human-readable text documenting the inheritance chain as a visual arrow sequence (`Database -> DBMS -> SoftwareComponent -> Root`), all properties with type annotations and inheritance provenance, capability definitions with nested type property schemas, and requirement specifications with detailed relationship type structures. This formatting strategy balances machine parseability (consistent indentation, structured sections) with semantic clarity (natural language descriptions, explicit inheritance annotations).

Constraint-Enforced YAML Generation

The generation prompt implemented explicit constraint instructions operationalizing DP3 (Metadata Preservation) and MR3 (Verifiability). Instruction (3) explicitly forbade hallucinated properties: "Only use properties and capabilities that exist in the TOSCA type definition provided." Instruction (6) mandated metadata fallback: "For all original Terraform properties that are provider-specific...include them in the node template's metadata section." This dual-target strategy balanced semantic abstraction (mapping `instance_type` to `num_cpus`/`mem_size`) with fidelity preservation (placing `ami`, `availability_zone` in metadata). The implementation employed Gemini 2.5 Pro (rather than Flash) for generation, trading increased latency for improved reasoning capability over complex, nested schemas. The generation chain composed a `PromptTemplate`, the LLM invocation, and a `StrOutputParser` via LangChain's pipe operator, producing pure YAML output without markdown formatting artifacts.

7.3 Evaluation of the Gamma Artifact

Following the implementation of the AI-augmented pipeline, the ADR team conducted a qualitative feasibility assessment to address preliminary insights for RQ2 and RQ3. Unlike the production-oriented Beta evaluation, this assessment prioritized the identification of capability boundaries and failure modes over quantitative performance metrics.

7.3.1 Evaluation Approach

Given the exploratory nature of Cycle 3 and the limited time and resources available for AI experimentation, the evaluation adopted a qualitative feasibility assessment methodology rather than rigorous quantitative benchmarking. The primary objective was to answer foundational questions: "Can AI-augmented translation work for basic resources?" and "What are the failure boundaries?" rather than demonstrating production readiness or systematic superiority over the deterministic approaches. This approach aligns with the ADR principle of guided emergence, where artifact evaluation informs the understanding of technical boundaries and generates design knowledge for future iterations.

Test Case Selection Strategy

A purposive sample of 9 Terraform resources was selected to probe the AI pipeline's capabilities across diverse semantic categories and abstraction challenges:

- **Compute:** `aws_instance` (EC2 virtual machine)
- **Networking:** `aws_vpc` (virtual private cloud), `aws_subnet` (network segment)
- **Storage:** `aws_s3_bucket` (object storage), `aws_ebs_volume` (block storage)
- **Database:** `aws_db_instance` (managed relational database service)
- **Security:** `aws_security_group` (firewall rules)

The exclusion of other resources reflected a deliberate scope limitation imposed by the AI pipeline's single-resource processing design. Resources that cannot exist meaningfully in isolation are deemed untestable within the current architecture, acknowledging that a comprehensive evaluation would require topology-aware reasoning capabilities beyond the prototype scope. Each testable resource was defined as an isolated Terraform block extracted from official AWS provider documentation examples, ensuring that the inputs represented canonical usage patterns. Multi-resource configurations were explicitly excluded to maintain the evaluation focus on per-resource classification and generation accuracy rather than relationship inference (which the pipeline does not attempt).

Evaluation Criteria and Success Definitions

The evaluation of the AI-driven translation pipeline was conducted as a qualitative feasibility study, focusing on generating design knowledge rather than quantitative production metrics. The assessment was based on four criteria. First, Structural Validity was a binary success/failure test ensuring that the AI produced parseable outputs: a valid `ToscaTypeSelection` Pydantic object for classification and syntactically correct YAML for generation, with a 100% success threshold. An observed exception involved occasional namespace prefix inconsistencies, where the LLM generated fully qualified type names (e.g., `tosca.nodes.Compute`) instead of short-form references (`Compute`) expected by the Simple Profile convention. While this technically violates the intended output format, the generated templates remained semantically valid—removing the `tosca.nodes.` prefix produced correct specification-compliant node templates. This failure mode was categorized as a minor formatting issue rather than a fundamental translation error, indicating that prompt refinement could enforce consistent naming conventions.

Second, Semantic Appropriateness was a qualitative judgment by the ADR team on whether the AI's architectural abstractions were reasonable, such as correctly mapping `aws_instance` to `Compute` and ensuring that property mappings were coherent. Third, Metadata Preservation verified that all original Terraform arguments were retained, either as mapped TOSCA properties or within a structured `metadata` block.

The principal objective was Failure Mode Documentation, which involved systematically identifying and categorizing issues such as non-determinism, semantic divergences from existing deterministic systems, namespace prefix inconsistencies, and hallucinations not grounded in source documentation.

This study was intentionally constrained to define its scope as an exploratory research. Key non-goals included avoiding quantitative performance metrics (e.g., precision/recall), direct system comparisons, and full TOSCA compilation validations. The evaluation was limited to specific Gemini models and a single documentation corpus, focusing on identifying the boundaries of AI capabilities and understanding its failure modes. The process involved a standardized protocol for preparing Terraform inputs, executing the classification and generation chains with context from a Neo4j graph, and manually inspecting the outputs against the defined criteria, with all results and issues documented in structured notes.

7.3.2 Evaluation Results and Discussions

In its successful cases, the AI pipeline demonstrated that AI-augmented translation is technically feasible for straightforward, standalone resources. It successfully generated syntactically valid and semantically reasonable TOSCA node templates for 7 out of 9 testable cases, achieving a 78% success rate.

Classification accuracy is a key area of evaluation. All tested resources yielded structurally valid `ToscaTypeSelection` objects, meeting the 100% Pydantic schema compliance requirement. However, the semantic correctness of these classifications varies.

A number of classifications were deemed correct and appropriate, including `aws_instance` → `Compute`, `aws_s3_bucket` → `Storage.ObjectStorage`, `aws_vpc` → `Network`, and `aws_subnet` → `Network`. The mapping of `aws_ebs_volume` to `Storage.BlockStorage` is also considered a reasonable abstraction.

However, some issues were identified. The classification of `aws_db_instance` was inconsistent and exhibited non-deterministic behavior across different runs. Additionally, the mapping for `aws_security_group` to `Network` was considered approximate, as it differed from the deterministic `Root` mapping used by the Beta system.

Regarding the quality of property mapping, the generated templates showed evidence of semantic understanding that went beyond mere, literal transcription.

The translation of the `aws_instance` resource serves as a clear example of this capability, where the AI correctly maps Terraform arguments to meaningful TOSCA properties:

```yaml
web_server:
  type: Compute
  properties:
    name: web_server
    num_cpus: 2        # Inferred from instance_type via documentation
    mem_size: 1 GiB     # Correctly formatted scalar-unit
  metadata:
    provider: aws
    instance_type: t3.micro  # Original value preserved
    ami: ami-0c55b159cbfafe1f0
    availability_zone: us-east-1a
```

The LLM demonstrated three critical capabilities in this translation.

1. **Semantic inference**: Abstracted `instance_type: t3.micro` to `num_cpus` and `mem_size` by consulting retrieved AWS documentation describing instance specifications, going beyond syntactic pattern matching.

2. **Type formatting**: Correctly generated TOSCA scalar-unit format (`1 GiB` rather than raw integer `1024`) conforming to Simple Profile data type conventions.

3. **Metadata preservation**: Retained all provider-specific attributes (`ami`, `availability_zone`, `instance_type`) in the structured `metadata` section, operationalizing DP3 (Metadata Preservation).

Metadata Preservation: All successful test cases included comprehensive `metadata` sections containing original Terraform attributes, confirming the LLM understood the dual-target instruction (map when possible, preserve in metadata otherwise).

Cross-Resource Context Absence: As designed, the pipeline processed resources independently. Testing `aws_vpc` + `aws_subnet` produced two correctly classified node templates (`Network` types), but no relationship connected them, despite the subnet's implicit VPC dependency. This confirmed the anticipated limitation: relationship inference requires either deterministic graph analysis (as in Beta) or a separate AI reasoning module with full topology context, both of which are beyond the scope of Cycle 3.

Identified Limitations and Failure Modes

The evaluation identified several key limitations and failure modes that are inherent to the AI-augmented approach.

1. Non-Deterministic Classification with Semantic Ambiguity A primary issue was the inconsistent classification of the `aws_db_instance` resource, which was classified differently across the three identical runs.

- **Run 1**: `DBMS` (node type representing database management system)
- **Run 2**: `Database` (node type representing database instance)
- **Run 3**: `DBMS` again

This non-determinism stems from two root causes. The first was Retrieval Variance, where the vector store search algorithm returned slightly different documentation chunks for identical queries, influencing the LLM's classification choice. The second was a Prompt Design Constraint, which forced the selection of a single type, preventing the LLM from capturing the resource's composite nature (a `Database` hosted on a `DBMS`). This contrasts with the deterministic mapper's ability to decompose the resource into two connected nodes.

The key implication is that AI-based classification struggles to guarantee reproducibility, which is a critical requirement for production systems.

2. Semantic Divergence from Deterministic Baseline The classification of `aws_security_group` highlights a semantic interpretation difference between the AI and a rule-based system.

- **Beta (Deterministic)**: Mapped to `Root`, arguing the Simple Profile lacks a native security group abstraction.
- **Gamma (AI)**: Classified as `Network`, reasoning that security groups manage network access.

While both classifications are defensible, this divergence shows that semantic mapping involves judgment calls without a single "correct" answer. It has been observed that different mapping philosophies (e.g., conservative fallback versus semantic approximation) can produce valid yet incompatible models, challenging the idea of objective correctness in translation.

3. Context-Dependent Resource Testability Several resources, such as `aws_route_table` and `aws_internet_gateway`, were excluded from the testing because they are inherently relational and lack semantic purpose in isolation. This reveals a fundamental limitation of single-resource evaluation, as many Terraform resources cannot be translated independently. This implies that the 78% success rate of this study is applicable only to the subset of resources that represent standalone components, and comprehensive translation requires topology-aware reasoning.

4. Absence of Relationship Inference As a known design limitation, the AI pipeline did not attempt to infer the relationships between resources. For instance, when processing an `aws_instance` and its associated `aws_subnet`, two independent node templates are generated without a connecting relationship, such as `HostedOn`, despite the explicit reference in the Terraform code. This contrasts with Beta's deterministic graph-based methods for reconstructing topologies.

This limitations directly stems from Gamma's architectural scope: the prototype deliberately omits DP4 (Infer Relationships from Multiple Context Cues) and DP5 (Mirror Hierarchical Structure), which are the core principles of the deterministic Beta framework. Relationship inference requires analyzing cross-resource dependencies through implicit cues (e.g., Terraform reference expressions like `subnet_id = aws_subnet.main.id`) and reconstructing dependency graphs—capabilities that demand either symbolic graph traversal (as implemented in Beta's `ResourceGraphAnalyzer`) or a specialized, topology-aware LLM chain operating on complete infrastructure contexts rather than isolated resources.

The absence of DP4/DP5 in Gamma does not constitute a failure but rather reflects a strategic design choice: validating AI-augmented *node template generation* (MR1, MR3) independently before attempting the significantly harder problem of AI-based topology synthesis.

CHAPTER 8

Discussion

This chapter critically reflects on the research journey from problem formulation through three Action Design Research cycles, evaluating whether the proposed meta-requirements and design principles successfully addressed the core challenge of systematic IaC-to-TOSCA translation. Section 8.1 revisits each meta-requirement and design principle in light of empirical evidence from the Alpha, Beta, and Gamma artifacts, identifying where the initial formulations held firm and where practical experience demanded refinement. Section 8.2 assesses artifact maturity using an adapted Capability Maturity Model, tracing the progression from an ad-hoc proof-of-concept to a production-ready framework. Section 8.3 synthesizes the observed trade-offs between deterministic and AI-augmented approaches, moving beyond "which is better" to articulate complementary strengths. Section 8.4 distills generalizable lessons for practitioners and researchers, and Section 8.5 transparently examines threats to validity. Finally, Section 8.6 positions this study within the broader landscape of IaC portability, TOSCA ecosystem development, and AI-assisted code generation research.

8.1 Reflection on Meta-Requirements and Design Principles

The meta-requirements (MRs) and design principles (DPs) established in Chapter 4 served as the theoretical foundation guiding all interventions across the three BIE cycles. This section systematically revisits each MR and DP to assess their validity, relevance, and completeness in light of practical implementation experiences. For each meta-requirement, we examined whether it captured genuine organizational needs, how well it was addressed by the implemented interventions, and whether the cycle outcomes necessitated reformulation or extension. Similarly, each design principle was evaluated for its effectiveness in operationalizing the corresponding MRs, identifying cases where initial formulations proved insufficient or where new principles emerged from practice. Table 8.1 presents the final refined set of meta-requirements after this reflective analysis.

8.1.1 Meta-Requirements

The initial meta-requirements were formulated during the problem formulation stage based on expert discussions, literature analysis, and identification of systemic challenges in the IaC ecosystem (Section 4.3). After completing three BIE cycles involving the design, implementation, and evaluation of Alpha, Beta, and Gamma artifacts, these requirements can now be reassessed for relevance, completeness, and accuracy. This subsection examines each MR, discussing how practical experience validated, challenged, or extended the initial formulations.

MR1 - Semantic Fidelity and Intent Preservation

The requirement that translations must preserve architectural intent beyond mere syntactic conversion proved fundamental throughout all cycles. Beta's multi-tier application test (Section 6.3.2) empirically demonstrated that the correct abstraction of `aws_instance` to `Compute`, combined with accurate relationship inference (load balancer $\rightarrow$ target group $\rightarrow$ instances), successfully captured the intent of the three-tier design pattern. The ADR team's manual inspection confirmed that both explicit dependencies (`depends_on` declarations) and implicit relationships (symbolic references such as `subnet_id = aws_subnet.main.id`) were correctly translated into TOSCA topology graphs.

However, cycle experiences have revealed that semantic fidelity has inherent limits. The TOSCA Simple Profile expressiveness gap, discussed extensively in Section 6.3.2, forces compromises where provider-specific constructs lack semantic equivalents. For instance, `aws_security_group` was mapped to `Root` in Beta's deterministic pipeline, whereas Gamma's AI-augmented approach classified it as `Network`. Both mappings are defensible, illustrating that "correct" semantic abstraction often involves judgment calls rather than objective truths. This observation does not invalidate MR1 but contextualizes it: semantic fidelity must be understood as "best possible abstraction given target standard constraints" rather than "perfect equivalence."

The initial formulation emphasized topology preservation but underspecified the metadata preservation strategy. Cycle 2 established that lossless translation (MR4) is essential for semantic fidelity; provider-specific details such as AMI IDs and availability zones must be retained even when they lack TOSCA property equivalents. This interdependency between MR1 and MR4 was not explicit in the original formulation but became clear through the implementation. Overall, MR1 remains valid and is confirmed as the cornerstone requirement, with the understanding that its fulfillment is constrained by the target standard expressiveness.

MR2 - Extensibility and Maintainability

The architectural transformation from Alpha's monolithic prototype to Beta's plugin-based framework (Chapter 6) provided the most dramatic empirical validation of any meta-requirement. Alpha's evaluation (Section 5.3.3) definitively demonstrated that a monolithic design creates an architectural dead-end: adding support for Ansible would have required a near-complete rewrite, while adding even a single new AWS resource type (`aws_s3_bucket`) demanded manual editing of centralized mapping dictionaries embedded in the core logic.

Beta's modular architecture addresses these limitations through protocol-based abstraction (Intervention 6), reusable base classes (Intervention 7), and registry-based mapper dispatch (Intervention 8). The evaluation confirmed that adding new resource types now requires only implementing a `SingleResourceMapper` subclass and registering it—approximately 450 lines of isolated code with no core-framework modifications. This represents a fundamental improvement in the maintainability.

However, a critical gap emerged during Beta evaluation: architectural extensibility was validated through design review rather than empirical multi-plugin implementation (Section 6.3.2). The ADR team conducted structured walkthroughs to demonstrate how an Ansible plugin would be implemented, confirming that protocol contracts provide sufficient scaffolding. However, the Terraform plugin remains the sole working instantiation within the scope of this study. This introduces risk; hidden coupling or insufficient abstraction may only surface when implementing a second heterogeneous plugin (e.g., YAML-based Ansible vs. HCL-based Terraform; imperative tasks vs. declarative resources).

Reflecting on this gap, a refinement to MR2 is proposed: MR2b - The framework should demonstrate extensibility through architectural review AND at least two working plugin implementations from different IaC paradigms. While Beta successfully achieved "architectural readiness for extension," true validation of extensibility principles requires stress testing protocol contracts across heterogeneous implementations. This refinement represents design knowledge that emerged from practice, as single-plugin implementation cannot fully validate multi-paradigm abstraction claims.

MR3 - Verifiability and Specification Compliance

This requirement achieved unambiguous validation across all test phases. Every TOSCA 2.0 template generated by Beta was successfully compiled with the extended Puccini processor without syntax errors, providing objective evidence of specification compliance. The 100% success rate across diverse scenarios—from single-resource translations to the 40-resource multitiers application—confirms that the framework reliably produces standard-compliant outputs.

The Fluent Builder API (intervention 5) was instrumental in achieving this success.

By embedding TOSCA 2.0 validation rules within Pydantic models, the builder prevents the construction of malformed templates at build time rather than discovering errors during external validation. This "correctness by construction" approach eliminated the entire class of syntax errors that plagued Alpha's string-based generation, where mistakes were only caught during Puccini compilation.

One nuance emerged during the Gamma evaluation: syntactic correctness does not guarantee semantic correctness. Templates can be Puccini-valid yet semantically questionable—for example, the AI pipeline's `aws_security_group` → `Network` classification passes all structural validation but represents a different semantic interpretation than Beta's deterministic `Root` mapping. This distinction was implicit in the original MR3 formulation, which focused on "syntactic correctness and semantic validity according to the specification." In practice, TOSCA specifications define structural rules (i.e., what relationships are allowed) but not semantic appropriateness (i.e., which type best captures intent). This observation does not require reformulating MR3 but highlights that verifiability addresses form, not meaning; semantic fidelity (MR1) remains a separate concern.

MR4 - Abstraction–Specificity Balance

The lossless translation principle was successfully operationalized through structured metadata blocks that preserve all provider-specific attributes lacking direct TOSCA equivalents. Beta's implementation (Section 6.2.4) demonstrates this for complex resources: RDS instances retain engine versions, backup windows, and parameter group references in metadata, even though Simple Profile's `DBMS` type cannot express these properties.

However, cycle experiences revealed that the "balance" in this requirement is context-dependent and involves trade-offs that were not specified in the original formulation. Over-preservation of metadata reduces portability; a TOSCA template laden with AWS-specific artifacts cannot be trivially deployed to Azure without manually adapting the metadata. Conversely, over-abstraction sacrifices deployability; a template without AMI IDs or instance types cannot be instantiated on AWS without external configuration injection. The optimal balance depends on the use case: green-field TOSCA adoption favors abstraction, and migration scenarios favor preservation.

This observation suggests a refinement: MR4b - The framework should support configurable abstraction levels via profiles (minimal/standard/comprehensive metadata). Users can select the degree of provider-specificity to retain based on their portability versus fidelity priorities. This capability was not implemented in Beta but represents the natural evolution of the lossless translation principle, informed by practical experience.

MR	Meta-Requirement	Last changed during
MR1	The translation process must preserve semantic fidelity and architectural intent beyond syntactic equivalence.	Problem Formulation
MR1a	Both explicit and implicit dependencies in provider-specific IaC must be captured and reflected as TOSCA relationships to maintain design intent.	Problem Formulation
MR1b	Semantic fidelity should be interpreted as best-possible abstraction within target-standard constraints, recognizing the expressiveness limits of TOSCA 2.0.	Reflection
MR2	The framework should be extensible and maintainable, allowing new IaC technologies and resource types to be integrated without refactoring the core.	Problem Formulation
MR2a	Architectural extensibility must be validated through architectural review and at least two working plugin implementations from different IaC paradigms.	Reflection
MR3	The generated TOSCA 2.0 templates must be verifiable and specification-compliant, ensuring syntactic correctness and internal consistency.	Problem Formulation
MR3a	"Correctness by construction" should be enforced through model-based validation embedded in the Fluent Builder API.	Building & Evaluation (Beta)
MR4	The IaC-to-TOSCA translation must achieve a configurable abstraction–specificity balance, preserving provider metadata where needed for semantic traceability.	Problem Formulation
MR4a	Provider-specific attributes without TOSCA equivalents shall be retained in structured metadata blocks to ensure lossless translation.	Building & Evaluation (Beta)
MR4b	The framework should support configurable abstraction profiles (minimal / standard / comprehensive) to let users control portability–fidelity trade-offs.	Reflection

Table 8.1: Meta-Requirements after reflection based on ADR experiences

8.1.2 Design Principles

The design principles (DPs) articulated in Section 4.4 provided prescriptive guidance for constructing artifacts that would satisfy the meta-requirements. Unlike MRs, which define "what" the framework must achieve, DPs specify "how" to build it—concrete architectural and implementation strategies grounded in both theoretical foundations and expert knowledge. After three BIE cycles involving iterative design, implementation, and evaluation, these principles can be critically examined for their effectiveness, completeness, and generalizability. This subsection reflects on each DP, assessing whether it successfully operationalized its corresponding MRs, identifying gaps exposed during implementation, and proposing refinements based on the empirical evidence.

DP1 - Separation of Technology-Specific Concerns

Beta's protocol-based abstraction comprehensively validated this principle. Alpha's monolithic design created an architectural dead-end, where adding Ansible support would require a near-complete rewrite. Beta's `PhasePlugin`, `SourceFileParser`, and `ResourceMapper` contracts enabled the full isolation of Terraform-specific logic, allowing new AWS resource types to be added with zero core framework modifications. However, a critical gap emerged: validation occurred through single-plugin implementation plus architectural review, not through empirical multi-plugin development. Structured walkthroughs confirmed theoretical extensibility, but hidden assumptions may only surface when implementing heterogeneous plug-ins (e.g., Ansible's imperative tasks versus Terraform's declarative resources).

Refinement proposed: DP1b - Design plugin contracts through the implementation of at least two heterogeneous plugins from different IaC paradigms to empirically validate abstraction completeness. While Beta operationalized separation-of-concerns, true validation requires stress-testing protocols across fundamentally different IaC models. This design knowledge emerged from reflecting on the limitations of a single plugin.

DP2 - Prioritize Semantic Mapping over Syntactic Translation

This principle proved central to achieving MR1 and was consistently applied across mapping interventions. Beta's translation of `aws_instance` to `Compute` rather than `AwsInstance` exemplifies semantic abstraction—describing architectural role over provider implementation. Multi-tier application tests confirmed the preservation of intent.

However, the `aws_security_group` case revealed a subtle challenge: Beta mapped it to `Root` (lacking a Simple Profile equivalent), whereas Gamma classified it as `Network` (network access control semantics). Both mappings are defensible and Puccini-valid, illustrating that semantic abstraction involves judgment calls without an objective ground truth. Deterministic systems encode one expert interpretation and do not discover "the" correct mapping.

Refinement proposed: DP2b - Document semantic mapping rationale to enable alternative interpretations and facilitate collaborative refinement. Capture reasoning behind type selection (why `Root` not `Network`?) as structured documentation, enabling reviewers to contest decisions with full context and identify remapping candidates when the Simple Profile evolves.

DP3 - Preserve Provider-Specific Details in a Traceable Form

Beta systematically preserved Terraform attributes lacking TOSCA equivalents in structured metadata, achieving lossless translation (MR4). Complex resources such as RDS instances retain dozens of AWS-specific configurations—backup windows, parameter groups, and engine options—that Simple Profile's `DBMS` cannot express. The evaluation confirmed comprehensive traceability.

A practical tension emerged: comprehensive metadata reduces portability—AWS-saturated templates cannot be trivially deployed to Azure without manual adaptation. Orchestrators often ignore metadata, raising questions regarding its utility. However, metadata enables reverse translation, supports auditing, and provides human context, even when machines ignore it.

The principle remains valid as formulated, but the implementation revealed that it represents one spectrum point. Organizations prioritizing portability might prefer minimal metadata, whereas those prioritizing fidelity might prefer comprehensive preservation. This informed MR4b (configurable abstraction profiles)metadata strategy should be configurable, not all-or-nothing. The principle correctly operationalizes lossless translation but requires a flexible implementation.

DP4 - Infer Relationships from Multiple Context Cues

Beta successfully implemented this through three strategies: explicit `depends_on`, symbolic references, and pattern knowledge. The multi-tier test validated this, with 58 edges correctly inferred across 40 resources. The implementation revealed an effectiveness hierarchy: symbolic reference extraction (e.g., `subnet_id = aws_subnet.public.id`) proved to be the most reliable; explicit `depends_on` covered only manual constraints; heuristic patterns were brittle.

Gamma's complete failure in relationship inference exposes a critical assumption: DP4 requires a topology-wide context. Single-resource LLM prompts cannot identify such dependencies. Symbolic parsing requires the analysis of expressions across resources, and pattern recognition requires cross-resource comparison. The principle remains valid but requires architectural support for graph-aware reasoning, whether deterministic traversal (Beta) or multi-turn LLM conversations (unexplored). This highlights that relationship inference cannot be decomposed into isolated per-resource operations, informing future AI-augmented architecture design.

DP5 - Mirror the Hierarchical Structure of the TOSCA Model

The Fluent Builder API validated this principle, achieving MR3 (verifiability) and MR2 (maintainability). Builder composition—`ServiceTemplateBuilder` coordinating nested `NodeTemplateBuilder`, `RequirementBuilder`, `CapabilityBuilder`—directly mirrors TOSCA structure. Beta evaluation confirmed syntax error elimination; every template passed the Puccini compilation on the first attempt. "Correctness by construction" succeeded.

An unexpected trade-off emerged: a steeper learning curve. The ADR team members noted that the fluent interface required more cognitive investment than string formatting. For developers accustomed to templates (Jinja2), builders felt heavy-weight for simple cases. This trade-off represents an inherent choice, not a design flaw; production systems prioritize correctness over learnability. The principle requires no revision; organizations choose based on the context (rapid prototyping versus robust validation). Beta demonstrates that when correctness is paramount, mirroring the TOSCA structure internally is highly effective.

DP6 - Model Domain Knowledge as a Structured, Queryable Knowledge Representation

Gamma's Neo4j knowledge graph validated this principle for AI-augmented translation. Graph representation of TOSCA Simple Profile—node types with `DERIVED_FROM` inheritance, properties via `HAS_PROPERTY` relationships—enabled semantic queries impossible with flat documentation. The `get_type_with_full_context()` query aggregates inherited properties across entire type hierarchies, providing LLMs with comprehensive schemas that prevent the hallucination of non-existent properties.

This structured knowledge was essential for constraint-based generation (DP8). However, scalability remains unvalidated—only Simple Profile (234 nodes) was ingested; the full TOSCA 2.0 specification with custom organizational profiles might stress graph query performance. Moreover, knowledge graph construction requires manual YAML parsing and schema design, lacking automation for standard updates.

Refinement proposed: DP6b - Knowledge graphs should support versioning for standard evolution tracking. As TOSCA evolves, maintaining historical versions enables compatibility analysis and migration path generation. The core of the principle, that is, the queryable structured representation, is validated, but production systems require version management.

DP7 - Enable Multi-Source Contextual Retrieval with Relevance Prioritization

Gamma's dual-source RAG (TOSCA graph + Terraform documentation vector store) demonstrated this principle's value. Semantic search with metadata filters (`resource_type == "aws_instance"`) plus cross-encoder re-ranking provided contextually relevant documentation that grounded LLM decisions. The classification accuracy improved over the raw LLM baseline (not quantified but qualitatively observed).

However, retrieval variance emerged as a fundamental limitation: identical queries returned slightly different documentation chunks across runs, causing non-deterministic classification (`aws_db_instance` flip-flopped between `DBMS` and `Database`). This stemmed from the approximate nearest-neighbor search algorithms and chunk ordering instability.

Refinement proposed: Implement deterministic retrieval ordering (e.g., metadata filters + fixed k + sorted by embedding distance) before reranking to eliminate the variance. While multi-source RAG enhances context quality, production systems require reproducibility and consistency. The principle correctly identifies the need for diverse knowledge sources but underspecified determinism requirements. Temperature=0 and fixed seeds proved insufficient; the retrieval layer itself requires deterministic guarantees.

DP8 - Constrain Translation Outputs to Validated Domain Schemas

Pydantic-based structured output successfully prevented type hallucination—LLMs could not generate non-existent TOSCA types when constrained by `ToscaTypeSelection` schema populated with Simple Profile enumeration. All Gamma classifications produced schema-valid objects. This validated the "constrained generation over free-form text" approach for domain-specific code generation.

However, namespace prefix inconsistencies emerged: LLMs occasionally generated `tosca.nodes.Compute` instead of `Compute`, technically violating Simple Profile conventions despite semantic correctness. Post-processing could strip prefixes, but this indicated that schema constraints address structure, not format.

Refinement proposed: Combine schema validation with regular expression-based format enforcement for string fields. Pydantic validators can enforce naming patterns beyond the type checks. The principle correctly prioritizes constraint-based generation but should explicitly include format validation and schema compliance. This represents a minor technical gap rather than a conceptual flaw, validated through Gamma's otherwise successful hallucination prevention.

Initial Design Principle	MRs	Interventions	Eventual Design Principle
DP1 – Separation of Technology-Specific Concerns (Plugin Architecture)	MR2	6, 8	Maintain strict separation of provider-specific logic through protocol-based plugin contracts. Refined as DP1b: Validate abstraction completeness by implementing at least two heterogeneous plugins (e.g., Terraform, Ansible).
DP2 – Prioritize Semantic Mapping over Syntactic Translation	MR1	2, 5	Preserve architectural intent rather than literal syntax during translation. Refined as DP2b: Document semantic mapping rationale to support collaborative reinterpretation and version evolution.
DP3 – Preserve Provider-Specific Details in a Traceable Form	MR4	4, 7	Maintain lossless translation by storing provider-specific attributes in structured metadata. Confirmed as valid; informs MR4b (configurable abstraction profiles).
DP4 – Infer Relationships from Multiple Context Cues	MR1, MR3	3, 9	Infer TOSCA relationships from explicit dependencies, symbolic references, and heuristic patterns. Confirmed, but requires graph-aware reasoning for AI-based pipelines.
DP5 – Mirror the Hierarchical Structure of the TOSCA Model	MR2, MR3	5	Implement builder composition that mirrors TOSCA's nested hierarchy, enforcing "correctness by construction." Confirmed without modification.
DP6 – Model Domain Knowledge as a Structured, Queryable Representation	MR1, MR3	9	Represent TOSCA and IaC semantics in a graph-based knowledge base to enable semantic queries. Refined as DP6b: Support versioning to track standard evolution and ensure compatibility.
DP7 – Enable Multi-Source Contextual Retrieval with Relevance Prioritization	MR1, MR3	9	Combine TOSCA knowledge graphs and IaC documentation in RAG-based retrieval. Refined: enforce deterministic retrieval ordering for reproducible AI reasoning.
DP8 – Constrain Translation Outputs to Validated Domain Schemas	MR3	9	Use schema-constrained generation (Pydantic) to ensure syntactic correctness. Refined: combine schema validation with regex-based format enforcement to guarantee consistency.

Table 8.2: Design Principles: initial formulation versus final reflection

8.2 Lesson Learned

This section distills generalizable insights derived from the three BIE cycles, moving beyond the specific technical contributions of Alpha, Beta, and Gamma artifacts to extract design knowledge applicable to broader contexts. Following the ADR principle of formalization of learning (Stage 4), these lessons synthesize theoretical understanding with practical experience, addressing both methodological aspects of conducting design research in infrastructure automation domains and substantive findings about IaC-to-TOSCA translation challenges. Each lesson is grounded in concrete observations from the research process—whether from implementation struggles, evaluation surprises, or stakeholder feedback—and articulated as actionable guidance for future researchers and practitioners working on similar portability and standardization issues.

Lesson 1: TOSCA Simple Profile Expressiveness is the Bottleneck. The quality of reverse-engineered models is fundamentally constrained not by the sophistication of the mapping tool but by the expressiveness of the target standard itself. During testing, our deterministic mapper frequently defaulted to the generic `Root` type for critical AWS constructs, such as security groups and target groups, as the TOSCA Simple Profile lacks equivalent abstractions. This indicates that investing in advanced classification algorithms yields diminishing returns when the bottleneck is the limited taxonomy of the target. A perfect classifier still produces a semantically weak model when forced to map a complex resource, such as `aws_lb_target_group`, to a generic `LoadBalancer`.

Consequently, the most effective path forward is to advocate for community-driven TOSCA extensions to better represent modern cloud patterns. Introducing types such as a `NetworkPolicy` for firewall rules or a `LoadBalancerPool` for target groups is essential. Until the standard evolves, any generated model will remain semantically shallow, limiting its utility for intent-based orchestration. This illustrates a core challenge for any standard: balancing vendor-neutral generalization with specialized abstractions required by contemporary infrastructure.

Lesson 2: Architectural Extensibility Must Be Empirically Validated Through Multiple Heterogeneous Implementations The validation of the extensibility of the plugin architecture, while successful in theory through architectural reviews, was only empirically tested with a single Terraform plugin. This creates a significant risk of hidden coupling, where design choices may be unconsciously influenced by Terraform-specific patterns. For example, the protocol's SourceFileParser.parse() method returns a dict[str, Any], a structure that naturally fits Terraform's output but may be unsuitable for paradigms such as Ansible's list-based playbooks.

To genuinely validate the claim of being technology-agnostic, future work must implement at least two plugins from fundamentally different IaC paradigms, such as declarative Terraform and imperative Ansible. The architecture can only be considered truly extensible if both plug-ins can be integrated without requiring modifications to the core protocols. This highlights a crucial methodological insight for research claiming generalizability: theoretical designs are inadequate. Empirical validation requires the implementation of multiple diverse client extensions to ensure that an abstraction is not merely a name.

Lesson 3: Terraform's `plan -json` is a Pragmatic Compromise with Semantic Trade-offs. The decision to use Terraform's evaluated plan output instead of parsing raw HCL files was a pragmatic choice that ensured semantic fidelity by offloading the complexity of variable interpolation and module expansion. This allows the translation to operate on the final deployment intent.

However, this strategy introduces significant trade-offs. This resulted in a loss of source-level traceability, as the plan's JSON output did not link resources back to their origin. tf files, which complicates debugging. It also created an operational dependency on the Terraform CLI within the translation environment and limited the applicability of the pattern to other IaC tools that may lack a similar machine-readable output.

While alternative HCL parsing libraries were considered, they were rejected because they only provided unevaluated syntax trees, which would require reimplementing Terraform's runtime logic and risk semantic divergence.

The conclusion is that for research prioritizing correctness, delegating the evaluation to the source tool is a justified approach. However, a production-grade solution would likely require a hybrid model that uses HCL parsing for source provenance while leveraging the evaluated plan for semantic accuracy. This insight broadly applies to translation frameworks for any DSL with complex evaluation semantics.

Lesson 4: LocalStack Enables Rapid Iteration but Cannot Validate Production Semantic Fidelity. The integration of LocalStack was crucial for development velocity, enabling cost-free reproducible testing, fast iteration cycles, and CI/CD integration. This allowed for the safe and rapid development of the mappers.

However, it has significant limitations. The free tier of LocalStack had service coverage gaps, notably lacking support for key components such as Application Load Balancers, which forced the implementation of a less complete plan-only fallback mode. Furthermore, emulation fidelity was not perfect, with potential differences in computed attributes and behaviors compared to real AWS, creating a risk of false confidence, where a successful local deployment might still fail in production.

This directly impacted the validity of the study. The claim of "semantic fidelity" could only be fully validated for the subset of services compatible with LocalStack, as the fallback mode could not resolve dynamic dependencies in more complex scenarios.

Therefore, the recommendation for future production-grade work is to complement LocalStack testing with ephemeral deployment in a real cloud environment. Although LocalStack is invaluable for development and initial validation, it cannot replace testing in a production-like environment when making strong claims about semantic correctness.

Lesson 5: AI-Augmented Translation Requires Topology-Aware Reasoning, Not Document Retrieval. Although the AI-augmented pipeline successfully classified individual resources, it completely failed at relationship inference, which is a critical aspect of semantic fidelity. When processing connected resources such as an `aws_instance` and an `aws_subnet`, the system generates two disconnected node templates, ignoring explicit references in the Terraform code.

The root cause is an architectural mismatch: the RAG pipeline processes resources in isolation, providing the LLM with documentation for a single resource type at a time. It lacks the graph-level context necessary to interpret cross-resource dependencies. This is a fundamental limitation, as a topology without connecting relationships is architecturally meaningless. In contrast, the deterministic Beta system succeeded because it treated the entire infrastructure plan as a graph, allowing the analysis and reconstruction of these connections.

Potential solutions involve more complex AI architectures, such as multiturn conversations or hybrid symbolic-neural approaches. However, the core lesson is that RAG, while effective in augmenting generation with facts, is not suited for the relational reasoning inherent in infrastructure translation. This task is fundamentally a graph transformation problem that requires LLMs to be integrated with graph-aware processing rather than being used as a substitute for deterministic graph analysis.

Lesson 6: Non-Determinism is Unacceptable for Infrastructure Translation. The evaluation revealed that the AI pipeline exhibited critical non-deterministic behavior, as observed when the classification of `aws_db_instance` alternated between `DBMS` and `Database` across identical runs. This instability was traced to a combination of retrieval variance from the vector search algorithm and the inherent stochasticity of LLM APIs, which is particularly pronounced when dealing with semantically ambiguous terms.

This lack of reproducibility is a disqualifying flaw in production IaC systems, which require identical outputs from identical inputs for auditing, compliance, and operational predictability. Efforts to mitigate this variance through technical means were unsuccessful, suggesting that the issue is fundamental to current LLM-based architectures rather than a simple implementation bug.

Therefore, a fully autonomous translation pipeline based solely on LLM- and RAG-based approaches remains infeasible with current technology. The observed stochasticity and retrieval variance indicate that, while these systems can effectively support semantic reasoning, they cannot yet guarantee deterministic and reproducible behavior, which is a fundamental requirement for production-grade IaC translation. A more viable direction is the human-in-the-loop paradigm, in which the AI proposes candidate mappings with associated confidence scores, enabling expert verification and correction.

8.3 Threats to Validity

Following the guidelines of Othmane et al. for empirical software engineering research, this section systematically examines potential threats to the validity of the findings presented in this thesis [66]. Threats to validity represent factors that could undermine the credibility of research conclusions or limit their generalizability beyond the specific context of this study. We organized these threats according to the established taxonomy: internal validity (whether observed effects genuinely resulted from the interventions), external validity (generalizability to other contexts), construct validity (whether measurements accurately capture intended concepts), and conclusion validity (whether inferences from data are warranted). Transparent acknowledgment of these limitations is essential for situating the contributions of this work appropriately and guiding future research that can address these gaps. Rather than weakening the research, the explicit identification of threats demonstrates methodological rigor and provides a roadmap for validation studies that can strengthen the evidence base for IaC-to-TOSCA translation frameworks. ma

8.3.1 Internal Validity

Internal validity concerns whether the observed outcomes can be confidently attributed to the designed interventions rather than confounding factors, measurement artifacts, or researcher bias. For this ADR study, internal validity threats center on the evaluation methodology, test case representativeness, and environmental fidelity.

Threat 1: Evaluation Bias and Proximity to Implementation. A significant threat to internal validity arises from evaluation bias because the primary researcher was responsible for both implementing the system and evaluating its output. This proximity creates a risk of confirmation bias, especially in the subjective assessment of "semantic appropriateness," where the researcher might unconsciously favor interpretations that align with their design decisions.

To mitigate this bias, an objective external validation strategy was employed. All generated TOSCA templates were compiled using Puccini, an independent and automated correctness checker. Puccini functions as a "ground truth" validator, deterministically catching any structural or syntactic errors. This provided an impartial check on the output's validity, operating entirely outside the researcher's control and reducing the impact of subjective judgment on the evaluation results of the model.

Threat 2: Limited ADR Team Size and Practitioner Involvement. The second threat to validity is the limited size of the research team, which consisted of the researcher and two academic supervisors, without direct involvement from end-user practitioners, such as DevOps engineers. This deviates from typical ADR studies that emphasize the need for broad stakeholder collaboration.

The team's structure was justified by the highly specialized nature of the research, which prioritized deep TOSCA domain expertise over broad representation of the research. The connection of a supervisor to the official OASIS TOSCA Technical Committee was intended to serve as a proxy for the practitioner perspectives.

However, residual risks remain. The absence of direct feedback from DevOps practitioners means that operational concerns, such as deployment workflows and CI/CD integration, were designed based on anticipated needs derived from expert judgment rather than from observed pain points in real-world environments. This potentially limits the validity of claims regarding the practical utility and usability of artifacts.

Threat 3: Test Case Selection and Coverage. The third threat to validity concerns the test case selection, which, while designed to cover common infrastructure patterns, may not represent the full diversity of real-world Terraform configurations. The evaluation relied on canonical examples and a researcher-constructed application that introduced several biases.

The selection focused on well-documented resources, potentially overestimating success for more obscure services, and avoided complex Terraform edge cases, such as dynamic blocks or `for_each` with nested modules. Furthermore, the primary multi-tier test was constrained by resources supported by LocalStack's free tier, limiting its scope.

Although this was partially mitigated by designing the 40-resource application to include realistic complexity and interdependencies, a significant residual risk remains. No systematic coverage analysis was performed to quantify the percentage of the Terraform AWS provider tested. Consequently, the findings support claims about handling "common infrastructure patterns" but cannot be generalized to comprehensive AWS coverage. More rigorous validation would require stratified sampling across service categories and testing with real-world open-source modules.

Threat 4: Temporal Validity and Specification Evolution. The fourth threat concerns temporal validity, as the research is anchored to the 2024-2025 versions of TOSCA 2.0 and the Terraform AWS provider. As both standards evolve, there is a significant risk of obsolescence, where future updates could invalidate the design principles or implementation patterns developed in this study, such as the strategy for parsing Terraform's plan output.

To mitigate this, the system was designed with abstraction. A plug-in architecture isolates version-specific logic behind stable interfaces, theoretically allowing the system to adapt to schema changes without a full redesign.

However, a residual risk remains. While this modularity can handle incremental updates, major paradigm shifts, such as a fundamental change in TOSCA's modeling constructs or Terraform replacing its plan introspection mechanism, could still require substantial rework. The framework's ability to adapt to such large-scale changes remains unvalidated, posing a threat to the long-term validity of its extensibility.

8.3.2 External Validity

External validity concerns the extent to which findings and design knowledge derived from this research can be generalized beyond the specific technological context, organizational setting, and methodological constraints of the study. Threats to external validity challenge the applicability of the proposed framework, meta-requirements, and design principles to other IaC technologies, cloud providers or organizational environments.

Threat 1: Single IaC Technology Implementation. The most significant threat to the generalizability of the framework is that its plugin architecture was only empirically validated with a single Terraform implementation. The claim that it can support heterogeneous IaC paradigms remains theoretical, as no second plug-in for a different technology, such as Ansible, has been developed.

This creates a substantial risk that the design contains hidden assumptions based on Terraform's declarative, state-based model. These assumptions may complicate adaptation to fundamentally different paradigms, such as Ansible's imperative, task-based approach. For instance, data parsing protocols are optimized for Terraform's structure and may not suit others.

Although architectural walkthroughs suggest theoretical extensibility, this reasoning cannot replace empirical validation. Without implementing a second plugin from a contrasting paradigm, the framework's claim of being genuinely technology-agnostic is not fully substantiated and risks being a "Terraform abstraction in disguise."

Threat 2: Single Cloud Provider Focus. The second major threat to generalizability is the exclusive focus on AWS, which questions whether the translation patterns are applicable to other providers, such as Azure or GCP. The framework may contain embedded assumptions based on AWS's specific service granularity, resource naming conventions, and unique constructs that have no direct cross-cloud equivalents.

Although the architecture was intentionally designed to separate provider-specific mapping logic from generic, provider-agnostic topology inference, suggesting that the core algorithms could be generalized, a significant risk remained. The quality of any semantic translation is fundamentally constrained by the expressiveness of the TOSCA Simple Profile. The abstraction gaps already identified for AWS services, such as for security groups, are likely to be even more severe when attempting to model the unique and complex services of other cloud providers, limiting the practical extensibility of the framework.

Threat 3: TOSCA Simple Profile as Target Standard. The third threat to generalizability is the exclusive reliance on the TOSCA Simple Profile as the target standard. This profile is notably sparse, providing only a generic set of types that lack the specificity needed to accurately model many modern cloud infrastructure concepts.

This limitation directly impacts research, as it forces the translation process to make significant semantic compromises. Complex cloud resources, which have no direct equivalent in the bare-metal profile, must be mapped to overly generic types, causing a loss of critical architectural detail.

Consequently, the study's findings on translation challenges are fundamentally bound to the expressive poverty of this specific profile. The difficulties encountered may not represent a universal translation problem but rather reflect the inadequacy of the chosen target standard for representing complex real-world systems.

Threat 4: Evaluation Methodology Limitations. The external validity of this study was constrained by the evaluation methodology, which may not reflect real-world usage. The reliance on researcher-designed synthetic test cases, rather than complex production codes, means that the framework was not tested against the messiness of legacy infrastructure.

Furthermore, the study was a snapshot in time and did not include a longitudinal evaluation to assess long-term maintainability against the evolving standards. A crucial limitation is the absence of usability studies involving the target user population, such as DevOps engineers.

Consequently, while the research demonstrates technical feasibility and architectural soundness, its findings cannot validate claims regarding practical adoption, usability, or successful integration into real-world workflows. However, these aspects remain speculative without field deployment and user feedback.

8.3.3 Construct Validity

Construct validity examines whether the measurements and operationalizations used in the research accurately capture the theoretical concepts they intend to represent. For this study, construct validity concerns center on how abstract meta-requirements were translated into concrete evaluation criteria and whether those criteria genuinely assess the intended qualities of the translation framework.

Threat 1: Operationalization of "Semantic Fidelity". A key threat to construct validity is the operationalization of MR1 (Semantic Fidelity), which was assessed qualitatively rather than quantitatively. The evaluation relied on the subjective judgment of the research team to determine if the translations were "appropriate," without a formal definition of semantic equivalence.

However, this approach has significant limitations. The existence of multiple defensible mappings for the same resource, such as for `aws_security_group`, demonstrates that there is no single objective "ground truth." The research lacks quantitative metrics from graph theory or information retrieval (e.g., precision/recall) to measure semantic similarity and lacks formal inter-rater reliability analysis to quantify the agreement between evaluators.

Consequently, the claim of preserving semantic intent rests on face validity—it appears correct to experts—but lacks construct validity. This introduces a significant risk that the findings reflect the evaluators' biases rather than an objective measure of semantic equivalence.

Threat 2: "Extensibility" Definition and Validation. The second threat to construct validity concerns the definition of MR2 (extensibility). This research validated this concept through an architectural reviewusing structured walkthroughs of a hypothetical Ansible pluginrather than by implementing a second working plugin.

This approach raises validity issues. This study adopted a broad interpretation of extensibility, defining it as having an architecture with properties that support extension (such as loose coupling). However, this "construct under-representation" fails to measure the practical aspects of integration, such as API ergonomics or debugging challenges, which would only surface during an actual implementation.

While some horizontal extensibility was shown by adding new mappers within the Terraform plug-in, the more crucial vertical extensibility (adding a new technology) was not empirically proven. Therefore, the claim of "validated extensibility" must be qualified as being only architecturally validated, not practically demonstrated.

Threat 3: "Verifiability" Scope and Limitations The third threat to construct validity lies in the scope of "verifiability" (MR3). This was operationalized through the Puccini compilation, which successfully validated that the generated TOSCA was syntactically and structurally correct according to the specification. This provides strong validity for the formal compliance.

However, a significant construct validity gap exists because Puccini cannot assess semantic correctness. It cannot determine whether the chosen TOSCA node type is an appropriate architectural representation of the source resource. For example, Puccini would validate mapping a caching cluster to a DBMS type because it is structurally compliant, despite being a poor semantic fit.

Therefore, the strong claims of this study regarding verifiability are only valid for specification compliance and do not extend to semantic appropriateness. Conflating these two distinct concepts risks construct confounding, as automated verification and manual semantic assessment measure different things.

Threat 4: "Metadata Preservation" as Operationalization of Lossless Translation. The fourth threat to construct validity arises from how "lossless translation" (MR4) was operationalized. This study defined this as preserving all unmapped source attributes in a structured metadata block.

However, this approach raises a critical question: Does simply storing data constitute a "lossless" translation if that data are not semantically interpretable or actionable by a standard orchestrator? The current method achieves syntactic preservation (the bytes are retained) but not necessarily semantic preservation (the meaning is lost to downstream tools).

The claim of being "lossless" is further weakened as the research did not test reverse translation (TOSCA back to Terraform) to prove that the original code can be perfectly reconstructed. Therefore, the framework validates only a narrow definition of losslessness, and it is unclear whether this satisfies the true intent of the requirement, where the preserved information must remain useful.

Threat 5: AI-Augmented Approach Success Criteria. The final threat to construct validity lies in the evaluation of the AI-augmented approach. The research relied on qualitative, post-hoc criteria such as "semantic appropriateness," which were assessed through manual inspection rather than a formal, predefined rubric.

However, this methodology has significant limitations. The reported 78% success rate is difficult to interpret, as "success" was not rigorously defined. Critically, the evaluation lacked any baseline comparison, making it impossible to determine whether the performance of the AI model was meaningful or whether simpler methods could achieve similar results.

While this exploratory approach was a deliberate choice to identify capability boundaries, it means that the construct of "successful AI translation" is not well-defined. Consequently, the findings demonstrate the technical feasibility of the approach but do not constitute a validated performance. Stronger claims would require formal success criteria, standardized testing, and comparison with baselines.

8.3.4 Conclusion Validity

Conclusion validity concerns whether the inferences drawn from empirical observations are logically warranted and whether the evidence supports the claims made. Unlike the previous validity dimensions, conclusion validity focuses on the strength of reasoning connecting data to conclusions, rather than measurement quality or generalizability.

Threat 1: Qualitative Nature of AI Evaluation. The qualitative nature of the AI evaluation is a primary threat to the validity of the AI evaluation. The assessment of the gamma artifact relied on the manual inspection of only nine test cases, without statistical analysis or formal performance metrics. Therefore, the reported "78% success rate" is an anecdotal observation, not a rigorously measured result.

This small, non-statistical sample size severely limits the inferences that can be drawn from it. The evidence supports directional conclusions; for example, AI-augmented translation is technically feasible for simple resources. However, it cannot substantiate any quantitative claims regarding accuracy or performance. While this qualitative approach was appropriate for the study's exploratory goals, the conclusions must be carefully qualified as demonstrating possibilities rather than validated effectiveness or production readiness.

Threat 2: Causal Attribution in Cycle Evaluations. Another threat to conclusion validity is the difficulty in causal attribution within the research cycles. Because multiple changes, such as architectural refactoring and API improvements, are often implemented simultaneously, it is challenging to attribute specific outcomes, such as increased maintainability," to a single intervention.

This risk was mitigated by a staged evaluation approach, in which interventions were implemented and assessed incrementally within each cycle to help isolate their individual effects.

However, a residual risk remains. Some improvements are likely to arise from the interaction effects between multiple design principles. Disentangling the individual contribution of each change is difficult, as the research methodology did not allow for the factorial experimental design required for precise causal attribution.

Threat 3: Generalization from Limited Evidence. The final threat to conclusion validity is the risk of over-generalizing broad principles from the limited evidence gathered. For example, the conclusion that the expressiveness of the TOSCA Simple Profile is the primary bottleneck is an inductive leap based solely on the experience of mapping AWS resources.

This threat is mitigated by the transparent structure of this research, which explicitly separates direct observations from broader implications and recommendations. This makes inferential leaps clear to the reader. The study also carefully scopes its conclusions, such as attributing the AI's failure in topology inference to a specific architectural choice rather than making a sweeping claim about the AI's capabilities.

This approach means that while strongly supported conclusions relate to proven architectural patterns, more tentative claims about AI or multi-cloud applicability are appropriately qualified as requiring further validation.

CHAPTER 9

Conclusions

This chapter concludes the research by synthesizing the findings from the three Action Design Research cycles. This section provides direct answers to the research questions formulated in the Introduction, culminating in a response to the main research question concerning the systematic reverse engineering of IaC into TOSCA 2.0. This chapter outlines the primary research contributions, acknowledges the study's limitations, and offers recommendations for future academic research and industrial practice. It serves to formalize the design knowledge generated and situate the findings within the broader context of cloud infrastructure portability and its standardization.

9.1 Answers to Research Questions

This section answers the research questions formulated in Section 1.4, culminating in an answer to the main research question.

RQ$_1$. *What are the design principles for a deterministic, rule-based mapping from Terraform to TOSCA 2.0?*

Regarding the first research question on the design principles for a deterministic, rule-based mapping, this research identified and validated five core principles. These were instantiated in the Beta artifact of Cycle 2, which successfully addressed the architectural failures of the initial prototype design. The foundational principle is the Separation of Technology-Specific Concerns (DP1), realized through a modular, protocol-based plug-in architecture that proved essential for maintainability and future extension. The translation itself was guided by the principle of prioritizing semantic mapping over syntactic translation (DP2), which ensured that provider-specific resources were mapped to their abstract architectural roles, thereby preserving the design intent. To achieve a lossless translation, the principle of preserving provider-specific details (DP3) was implemented by retaining all non-mappable attributes in structured metadata blocks. Furthermore, the system was designed to Infer Relationships from Multiple Context Cues (DP4), successfully reconstructing complex topologies by analyzing explicit dependencies, implicit symbolic references, and semantic patterns. Finally, the principle of mirroring the hierarchical structure of the TOSCA model (DP5) was realized via a Fluent Builder API, which enforced "correctness by construction" and eliminated syntax errors.

RQ$_2$. *To what extent can Large Language Models augmented with RAG support automated IaC-to-TOSCA translation? A preliminary investigation of feasibility and limitations.*

The second research question asked to what extent Large Language Models (LLMs) with RAG could support this translation. The exploratory investigation in Cycle 3 demonstrated that this approach is promising but limited in scope. The AI-augmented pipeline succeeded in classifying individual standalone resources and generating their TOSCA node templates with reasonable semantic accuracy.

This was enabled by a RAG architecture that grounded the LLM in authoritative knowledge from a Neo4j graph of the TOSCA specification and a vector store of the Terraform documentation. However, this approach revealed two critical failures. First, it was entirely unable to perform relationship inference because its single-resource focus lacked the necessary topology-wide context. Second, the system exhibited non-determinism, classifying the same resource differently across identical runs, which is a disqualifying flaw for production systems that demand reproducibility.

🔍 **RQ$_3$.** *What are the observed trade-offs between deterministic rule-based mapping and AI-augmented approaches in the context of IaC-to-TOSCA translation?*

This leads to the third research question concerning the trade-offs between the two paradigms: the deterministic approach, embodied by the Beta artifact, offers high precision, verifiability, and reproducibility. It excels at the complex, graph-based task of relationship inference, which is critical for architectural fidelity.

Its main drawback is the significant manual engineering effort required to create and maintain the mappers, which poses a scalability challenge. In contrast, the AI-augmented approach offers flexibility and has the potential to reduce the effort required to support new resources. However, it is currently unreliable for topology reconstruction, is prone to non-determinism, and its "black-box" nature complicates verification. A clear trade-off exists between the reliability of deterministic systems and the potential efficiency of AI assistance.

🔍 **MRQ.** *How can provider-specific IaC files be systematically and accurately reverse-engineered into the vendor-neutral TOSCA 2.0 standard?*

Ultimately, this research sought to determine how provider-specific IaC files can be systematically and accurately reverse-engineered into the TOSCA 2.0 standard. The findings indicate that the most effective method is a modular deterministic framework founded on the design principles validated in this study. The system architecture must be plug-in-based to manage the fragmented IaC landscape. Accuracy and semantic fidelity are achieved by mapping architectural roles, preserving provider-specific metadata, and performing robust graph-based relationship inference.

Verifiability is best guaranteed through a "correctness by construction" approach using a Fluent Builder API. While a fully autonomous AI translation pipeline is currently infeasible, the most viable path forward appears to be a hybrid model, where a reliable deterministic core is complemented by AI-driven tools to assist and accelerate the development of new resource mappers.

9.2 Research Contributions

This study makes several contributions to both prescriptive design knowledge and practical tools for cloud infrastructure management. These contributions address a critical gap at the intersection of IaC practices and vendor-neutral orchestration.

The primary theoretical contribution is a validated set of meta-requirements and design principles that constitute a formal methodology for reverse engineering IaC. This framework offers prescriptive guidance on systematically translating provider-specific definitions into abstract, portable models while addressing key challenges, such as semantic intent preservation, architectural extensibility, and specification compliance. This study fills a notable gap in the literature, which has largely focused on forward engineering or horizontal IaC translation rather than the "vertical" reverse engineering to a higher level of abstraction explored in this thesis. Furthermore, this study provides a novel empirical comparison of two distinct translation paradigms: a deterministic rule-based system and an exploratory AI-augmented approach. By detailing their respective strengths, weaknesses, and operational trade-offs, this study offers a nuanced understanding of their complementary roles in the context of IaC transformation.

On a practical level, this study delivers three tangible artifacts. The most significant is Beta, a well-architected and extensible proof-of-concept framework for translating Terraform configurations into specification-compliant TOSCA 2.0 templates. Its plugin-based architecture and Fluent Builder API serve as a robust foundation for future extensions to other IaC tools and to cloud providers. Second, this study made a direct contribution to the open-source community by extending the Puccini compiler to support the TOSCA 2.0 standard. This foundational effort not only enabled the validation of this research but also strengthened the broader TOSCA ecosystem by providing the community with an essential and up-to-date tool. This contribution was reviewed and merged into the official project repository, providing an external validation of its quality and utility. Finally, the exploratory Gamma artifact serves as an architectural blueprint for developing AI-assisted translation tools, demonstrating a novel application of the Neo4j Knowledge Graph for representing TOSCA specifications within a dual-source RAG pipeline.

9.3 Limitations

While this research provides a validated framework for IaC reverse engineering, its conclusions must be considered in light of several limitations.

First, the scope of the empirical validation was constrained. The framework's plugin architecture was practically implemented for only a single IaC technology (Terraform) and a single cloud provider (AWS). Although the architecture was designed for extensibility, its applicability to other paradigms, such as imperative Ansibleor other cloud providers, remains theoretical. Without empirical validation across multiple heterogeneous technologies, hidden assumptions tied to Terraform's declarative model may exist.

Second, the quality of the translation is inherently limited by the expressiveness of the target standards. The TOSCA Simple Profile lacks specific high-fidelity representations for many modern cloud-native constructs, such as security groups and load balancer target groups. This "expressiveness gap" forced the translation to map complex resources to overly generic types, resulting in a loss of semantic precision in the translation. This is not a failure of translation logic but a fundamental constraint of the target standard itself.

Third, the testing environment introduced potential fidelity gaps in the results. The heavy reliance on LocalStack for development and testing, while crucial for rapid iteration, meant that the framework was not consistently validated against a live AWS environment. Gaps in LocalStack's service coverage and potential emulation inaccuracies create a risk that translations validated locally may not behave identically in production, particularly for complex or newly released services.

Finally, the evaluation of the AI-augmented approach was strictly exploratory and qualitative in nature. The assessment was conducted on a small, curated set of isolated resources and was not subjected to rigorous quantitative benchmarks. Therefore, while the findings demonstrate technical feasibility and identify key challenges such as non-determinism and failure in relationship inference, they cannot support definitive claims about the performance or production readiness of the AI-based method.

9.4 Academic Recommendations for Future Work

The findings and limitations of this thesis open several promising avenues for future academic research. Building on the foundational framework established in this study, the following areas warrant further investigation.

First, the most critical next step is to extend the empirical validation of the proposed deterministic framework. This involves implementing a second plugin for an IaC technology with a different paradigm, such as the imperative task-based model of Ansible. Such an effort would rigorously test the technology-agnostic claims of the protocol-based architecture and reveal any hidden assumptions associated with Terraform's declarative nature. Similarly, extending the framework to support another major cloud provider, such as Microsoft Azure or Google Cloud Platform, would further validate the design principles and provide richer data on the expressiveness gaps within the TOSCA standard.

Second, future research should aim to advance the capabilities of AI-augmented translations. The primary challenge is the inability of the current prototype to perform relationship inference. A key research direction is the development of a topology-aware AI reasoning model that can process an entire IaC plan as a coherent graph rather than as isolated resources. This could involve graph neural networks or multiturn LLM agents. Furthermore, given the observed non-determinism, a more pragmatic line of inquiry would be to design and evaluate a human-in-the-loop system for this purpose. Such a tool would leverage AI to suggest mappings and generate draft mappers, which a human expert would then review, refine, and approve, thereby combining AI's efficiency with human reliability.

Third, there is a significant opportunity to contribute to the evolution of the TOSCA ecosystem in the future. Based on the semantic gaps identified in this study, researchers could formulate and propose formal extensions to the TOSCA Simple Profile to better represent modern cloud-native constructs, such as security groups, IAM policies, and load balancer target groups. Another challenging and valuable research area is "round-trip" engineering, which investigates the reverse translation from a modified TOSCA model back into provider-specific IaC, enabling true bidirectional infrastructure portability.

9.5 Industrial Recommendations

Based on the findings of this study, several practical recommendations can be offered to industry practitioners, cloud architects, and organizations seeking to navigate the complexities of multi-cloud infrastructure and vendor lock-in.

First, organizations should adopt a phased, deterministic-first approach when pursuing portable infrastructure. Rather than attempting a high-risk, "big-bang" rewrite of existing IaC, a more pragmatic strategy is to use a deterministic translation framework, such as the one prototyped in the Beta artifact, to gradually convert proven Terraform modules into a repository of vendor-neutral TOSCA templates. This builds an architectural baseline for portability without disrupting current operations and avoids the unreliability of fully autonomous AI solutions.

Second, when developing internal infrastructure tools, it is crucial to prioritize modular, plug-in-based architectures from the outset. The experience of the Alpha artifact serves as a cautionary tale: a monolithic design, while initially simple, inevitably creates a maintenance and extensibility dead end in today's heterogeneous cloud environments. A modular architecture is essential for the long-term viability.

Third, practitioners and organizations are encouraged to actively engage with and contribute to open standard communities, such as OASIS TOSCA. This study identified the limited expressiveness of the TOSCA Simple Profile as a primary bottleneck for creating high-fidelity translations. Industry participation is vital for evolving these standards to better reflect the realities of modern cloud-native patterns, thereby making them more practical and valuable for everyone.

Finally, organizations should use AI as an assistant rather than an oracle. The current state of technology, as observed in the Gamma prototype, is not mature enough for fully autonomous IaC translation owing to issues with non-determinism and critical failures in topological reasoning. Therefore, AI tools should be leveraged to accelerate development, for example, by generating draft mappers for new resources; however, the final validation and integration must remain within a deterministic, auditable system managed by human experts.

Bibliography

[1] I. Kumara, M. Garriga, A. U. Romeu, D. Di Nucci, F. Palomba, D. A. Tamburri, and W.-J. van den Heuvel, "The do's and don'ts of infrastructure code: A systematic gray literature review," *Information and Software Technology*, vol. 137, p. 106593, 2021. (Cited on pages 2, 17 and 18)

[2] J. Alonso, L. Orue-Echevarria, V. Casola, A. I. Torre, M. Huarte, E. Osaba, and J. L. Lobo, "Understanding the challenges and novel architectural models of multi-cloud native applications–a systematic literature review," *Journal of Cloud Computing*, vol. 12, no. 1, p. 6, 2023. (Cited on pages 2, 18 and 43)

[3] Precedence Research. (2025) Multi-cloud management market size, share and trends 2025 to 2034. [Online]. Available: https://www.precedenceresearch.com/multi-cloud-management-market (Cited on page 2)

[4] J. Opara-Martins, R. Sahandi, and F. Tian, "Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective," *Journal of Cloud Computing*, vol. 5, no. 1, p. 4, 2016. (Cited on pages 3, 43 and 54)

[5] P. Lipton, D. Palma, M. Rutkowski, and D. A. Tamburri, "Tosca solves big problems in the cloud and beyond!" *IEEE cloud computing*, 2018. (Cited on pages 3, 30 and 55)

[6] T. Metsch, M. Viktorsson, A. Hoban, M. Vitali, R. Iyer, and E. Elmroth, "Intent-driven orchestration: Enforcing service level objectives for cloud native deployments," *SN Computer Science*, vol. 4, no. 3, p. 268, 2023. (Cited on pages 3, 26 and 34)

[7] Intel Labs. Intent-driven orchestration: Simplifying cloud and edge deployments. Intel Corporation. [Online]. Available: https://www.intel.com/content/www/us/en/research/news/intent-driven-orchestration.html (Cited on pages 3 and 34)

[8] D. A. Tamburri, W.-J. Van den Heuvel, C. Lauwers, P. Lipton, D. Palma, and M. Rutkowski, "Tosca-based intent modelling: goal-modelling for infrastructure-as-code," *Sics software-Intensive cyber-Physical systems*, vol. 34, no. 2, pp. 163–172, 2019. (Cited on pages 4, 27 and 31)

[9] S. Bhatia and C. Gabhane, *Reverse Engineering with Terraform: An Introduction to Infrastructure Automation, Integration, and Scalability Using Terraform*. Springer, 2024. (Cited on pages 6 and 28)

[10] S. Yeung, "A comparative study of rule-based, machine learning and large language model approaches in automated writing evaluation (awe)," in *Proceedings of the 15th International Learning Analytics and Knowledge Conference*, 2025, pp. 984–991. (Cited on page 6)

[11] S. Rädler, L. Berardinelli, K. Winter, A. Rahimi, and S. Rinderle-Ma, "Bridging mde and ai: a systematic review of domain-specific languages and model-driven practices in ai software systems engineering," *Software and Systems Modeling*, vol. 24, no. 2, pp. 445–469, 2025. (Cited on page 8)

[12] K. G. Srivatsa, S. Mukhopadhyay, G. Katrapati, and M. Shrivastava, "A survey of using large language models for generating infrastructure as code," *arXiv preprint arXiv:2404.00227*, 2024. (Cited on pages 9, 25 and 57)

[13] D. Ahuja, "Terraformization: Revolutionizing enterprise it strategy," *Global Journal of Engineering and Technology Advances*, vol. 23, pp. 296–306, 04 2025. (Cited on page 10)

[14] M. Saraswat and R. Tripathi, “Cloud computing: Comparison and analysis of cloud service providers-aws, microsoft and google,” in *2020 9th international conference system modeling and advancement in research trends (SMART)*. IEEE, 2020, pp. 281–285. (Cited on page 10)

[15] M. K. Sein, O. Henfridsson, S. Purao, M. Rossi, and R. Lindgren, “Action design research,” *MIS quarterly*, pp. 37–56, 2011. (Cited on pages 13, 37, 39, 40, 43, 49 and 50)

[16] A. Haj-Bolouri, S. Purao, M. Rossi, and L. Bernhardsson, “Action design research in practice: Lessons and concerns.” in *ECIS*, 2018, p. 131. (Cited on page 13)

[17] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, “Adoption, support, and challenges of infrastructure-as-code: Insights from industry,” in *2019 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 2019, pp. 580–589. (Cited on pages 17, 21, 24 and 54)

[18] A. Mikkelsen, T.-M. Grønli, and R. Kazman, “Immutable infrastructure calls for immutable architecture,” 2019. (Cited on page 17)

[19] N. Koneru, “Infrastructure as code (iac) for enterprise applications: A comparative study of terraform and cloudformation,” *American Journal of Technology*, vol. 4, no. 1, pp. 1–28, 2025. (Cited on page 18)

[20] D. Sokolowski, D. Spielmann, and G. Salvaneschi, “Automated infrastructure as code program testing,” *IEEE Transactions on Software Engineering*, vol. 50, no. 6, pp. 1585–1599, 2024. (Cited on page 18)

[21] M. Orzechowski, B. Balis, K. Pawlik, M. Pawlik, and M. Malawski, “Transparent deployment of scientific workflows across clouds-kubernetes approach,” in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 2018, pp. 9–10. (Cited on page 19)

[22] A. Verdet, M. Hamdaqa, L. D. Silva, and F. Khomh, “Assessing the adoption of security policies by developers in terraform across different cloud providers,” *Empirical Software Engineering*, vol. 30, no. 3, p. 74, 2025. (Cited on page 19)

[23] T. Binz, G. Breiter, F. Leyman, and T. Spatzier, “Portable cloud services using tosca,” *IEEE Internet Computing*, vol. 16, no. 3, pp. 80–85, 2012. (Cited on pages 20 and 30)

[24] C. Lauwers and C. Curescu, “Tosca version 2.0,” OASIS, Tech. Rep., Jul. 2025, last revised or approved by the OASIS TOSCA TC on 22 July 2025. [Online]. Available: https://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.html (Cited on pages 21, 22, 30, 55 and 56)

[25] J. Bellendorf and Z. Á. Mann, “Specification of cloud topologies and orchestration using tosca: a survey,” *Computing*, vol. 102, no. 8, pp. 1793–1815, 2020. (Cited on pages 21, 30, 31 and 55)

[26] A. Luzar, S. Stanovnik, and M. Cankar, “Examination and comparison of tosca orchestration tools,” in *European Conference on Software Architecture*. Springer, 2020, pp. 247–259. (Cited on pages 21, 30 and 55)

[27] M. Wurster, U. Breitenbücher, L. Harzenetter, F. Leymann, J. Soldani, and V. Yussupov, “Tosca light: Bridging the gap between the tosca specification and production-ready deployment technologies.” in *CLOSER*, 2020, pp. 216–226. (Cited on page 21)

[28] H. Koziolek, R. Hark, N. Eskandani, P. S. Nguyen, and P. Rodriguez, “Tosca for microservice deployment in distributed control systems: Experiences and lessons learned,” in *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2023, pp. 11–21. (Cited on page 21)

[29] T. Fellner, “Cross-vendor variability management for cloud systems using the tosca dsl/author tobias fellner, bsc,” 2024. (Cited on pages 22 and 54)

[30] E. J. Chikofsky and J. H. Cross, “Reverse engineering and design recovery: A taxonomy,” *IEEE software*, vol. 7, no. 1, pp. 13–17, 2002. (Cited on page 23)

[31] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza, “Software documentation issues unveiled,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1199–1210. (Cited on page 23)

[32] C. Artho and A. Biere, “Combined static and dynamic analysis,” *Electronic Notes in Theoretical Computer Science*, vol. 131, pp. 3–14, 2005. (Cited on page 23)

[33] A. Singhal and G. Shlok, “Reverse engineering,” *International Journal of Computer Applications*, vol. 975, p. 8887, 2014. (Cited on page 23)

[34] G.-P. Drosos, T. Sotiropoulos, G. Alexopoulos, D. Mitropoulos, and Z. Su, “When your infrastructure is a buggy program: Understanding faults in infrastructure as code ecosystems,” *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 2490–2520, 2024. (Cited on page 24)

[35] C. Raibulet, F. A. Fontana, and M. Zanoni, “Model-driven reverse engineering approaches: A systematic literature review,” *Ieee Access*, vol. 5, pp. 14 516–14 542, 2017. (Cited on page 24)

[36] H. A. Siala, K. Lano, and H. Alfraihi, “Model-driven approaches for reverse engineering—a systematic literature review,” *IEEE Access*, vol. 12, pp. 62 558–62 580, 2024. (Cited on page 24)

[37] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021. (Cited on page 25)

[38] E. Kitzelmann, “Inductive programming: A survey of program synthesis techniques,” in *International workshop on approaches and applications of inductive programming*. Springer, 2009, pp. 50–73. (Cited on page 25)

[39] Y. Dong, X. Jiang, J. Qian, T. Wang, K. Zhang, Z. Jin, and G. Li, “A survey on code generation with llm-based agents,” *arXiv preprint arXiv:2508.00083*, 2025. (Cited on page 25)

[40] X. Gu, M. Chen, Y. Lin, Y. Hu, H. Zhang, C. Wan, Z. Wei, Y. Xu, and J. Wang, “On the effectiveness of large language models in domain-specific code generation,” *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 3, pp. 1–22, 2025. (Cited on page 25)

[41] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, H. Wang, and H. Wang, "Retrieval-augmented generation for large language models: A survey," *arXiv preprint arXiv:2312.10997*, vol. 2, no. 1, 2023. (Cited on page 26)

[42] A. Clemm, L. Ciavaglia, L. Z. Granville, and J. Tantsura, "Intent-based networking-concepts and definitions," 2022. (Cited on pages 26, 27 and 34)

[43] S. Minhas, R. Jaswal, A. Sharma, and S. Singla, "Revolutionizing networking: A comprehensive overview of intent-based networking," in *2024 International Conference on Emerging Innovations and Advanced Computing (INNOCOMP)*. IEEE, 2024, pp. 463–468. (Cited on pages 26 and 34)

[44] Firefly, "Codification — firefly docs," https://docs.firefly.ai/detailed-guides/codification, 2025, accessed: 2025-10-07. (Cited on page 28)

[45] Pulumi, "Convert code to pulumi," https://www.pulumi.com/docs/iac/adopting-pulumi/converters/, 2025, accessed: 2025-10-07. (Cited on page 29)

[46] Y.-M. Hung, S.-C. Chien, and Y.-Y. Hsu, "Orchestration of nfv virtual applications based on tosca data models," in *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 2017, pp. 219–222. (Cited on page 30)

[47] K. Kaur, V. Mangat, and K. Kumar, "A study of openstack networking and auto-scaling using heat orchestration template," in *Intelligent Computing and Communication Systems*. Springer, 2021, pp. 169–176. (Cited on page 30)

[48] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 164–175. (Cited on page 32)

[49] I. Kumara, Z. Vasileiou, G. Meditskos, D. A. Tamburri, W.-J. Van Den Heuvel, A. Karakostas, S. Vrochidis, and I. Kompatsiaris, "Towards semantic detection of smells in cloud infrastructure code," in *Proceedings of the 10th International Conference on Web Intelligence, Mining and Semantics*, 2020, pp. 63–67. (Cited on page 32)

[50] E. Low, C. Cheh, and B. Chen, "Repairing infrastructure-as-code using large language models," in *2024 IEEE Secure Development Conference (SecDev)*. IEEE, 2024, pp. 20–27. (Cited on page 33)

[51] H. Hong, S. Lee, D. Ryu, and J. Baik, "Code smell-guided prompting for llm-based defect prediction in ansible scripts," *Journal of Web Engineering*, vol. 23, no. 8, pp. 1107–1126, 2024. (Cited on page 33)

[52] Y. Du, Y.-F. Ma, Z. Xie, and M. Li, "Beyond lexical consistency: Preserving semantic consistency for program translation," in *2023 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2023, pp. 91–100. (Cited on page 33)

[53] L. Pasquale, A. Sabetta, M. d'Amorim, P. Hegedűs, M. T. Mirakhorli, H. Okhravi, M. Payer, A. Rashid, J. C. Santos, J. M. Spring *et al.*, "Challenges to using large language models in code generation and repair," *IEEE Security & Privacy*, vol. 23, no. 2, pp. 81–88, 2025. (Cited on page 33)

[54] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, "Lost in translation: A study of bugs introduced by large language models while translating code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13. (Cited on page 33)

[55] T. Tavanxhiu, "Evaluating the action design research methodology for requirement analysis, modeling, and engineering in information system design," in *LIMEN 2023/9–Leadership, Innovation, Management and Economics: Integrated Politics of Research-CONFERENCE PROCEEDINGS*. Udruženje ekonomista i menadžera Balkana, pp. 115–123. (Cited on page 37)

[56] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén *et al.*, *Experimentation in software engineering*. Springer, 2012, vol. 236. (Cited on page 51)

[57] A. V. Indukuri, "Infrastructure as code: A paradigm shifts in cloud resource management and deployment automation," 2025. (Cited on page 53)

[58] I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez, "Model-based dsl frameworks," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006, pp. 602–616. (Cited on page 56)

[59] D. Jones and S. Gregor, "The anatomy of a design theory," *Journal of the Association for Information Systems*, vol. 8, no. 5, p. 1, 2007. (Cited on page 62)

[60] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005. (Cited on pages 68 and 72)

[61] J. Hunt, *Protocols, Polymorphism and Descriptors*. Cham: Springer International Publishing, 2019, pp. 311–323. [Online]. Available: https://doi.org/10.1007/978-3-030-20290-3_27 (Cited on page 85)

[62] M. M. Kandé and A. Strohmeier, "On the role of multi-dimensional separation of concerns in software architecture," in *Prooceedings OOPSLA workshop on Advanced Separation of Concerns*, 2000. (Cited on page 85)

[63] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. (Cited on pages 86, 101 and 104)

[64] K. Hawick, "Fluent interfaces and domain-specific languages for graph generation and network analysis calculations," in *Proc. Int. Conf. on Software Engineering (SE'13)*, 2013, pp. 752–759. (Cited on page 87)

[65] H. Singh and S. I. Hassan, "Effect of solid design principles on quality of software: An empirical assessment," *International Journal of Scientific & Engineering Research*, vol. 6, no. 4, pp. 1321–1324, 2015. (Cited on page 89)

[66] L. ben Othmane, M. G. Jaatun, and E. Weippl, *Empirical research for software security: foundations and experience*. CRC Press, 2017. (Cited on page 154)

Ringraziamenti

Desidero innanzitutto esprimere la mia gratitudine ai miei relatori della Jheronimus Academy of Data Science (JADS) e dell'Università di Salerno. Ringrazio il professore Tamburri, il professor Palomba e il dottor Fossati per la preziosa guida, la costante disponibilità e il fondamentale supporto fornito durante l'intero percorso di stesura di questo elaborato. La mia gratitudine si estende a tutta la comunità del JADS, che mi ha accolto con professionalità, offrendomi un ambiente di studio vivace e stimolante.

Il mio ringraziamento più profondo e sentito va al cuore di tutto, al mio punto di partenza e di ritorno: la mia famiglia. A mia madre e a mio padre, che con il loro amore incondizionato e i loro sacrifici hanno costruito le fondamenta indistruttibili di ogni mia scelta. Non mi avete fatto mancare mai niente, permettendomi di dedicarmi a questo percorso con tranquillità. Ma soprattutto, mi avete dato l'opportunità di vivere un'esperienza come l'Erasmus, che si è rivelata una botta di vitalità e di stimoli incredibile, cambiandomi profondamente. Questo traguardo è il frutto del vostro sostegno. Con un affetto che riempie il cuore, ringrazio i miei nonni, i miei zii e cugini. Siete stati la mia tifoseria personale, il mio porto sicuro; il vostro abbraccio, anche a distanza, è sempre arrivato forte e chiaro, dandomi la certezza di non essere mai solo.

Un capitolo a parte, quello più colorato e irrinunciabile, lo dedico a voi, amici del gruppo *WannaBrawl*. Siete entrati nella mia vita da pochi anni, ma l'avete cambiata profondamente, diventandone una colonna portante. Mi avete aiutato quando ne avevo più bisogno, e spero di aver fatto lo stesso per voi, in quello scambio reciproco e sincero che è il cuore della nostra amicizia. Mi avete insegnato che la vita è molto più di una scadenza e a ricordarmi, di tanto in tanto, di "sentire l'odore dei pini" lungo il tragitto. Sono eternamente grato per avermi accolto nel vostro gruppo. La vostra amicizia è il regalo più bello degli ultimi anni. Grazie per essere esattamente come siete.

Un pensiero speciale va ai miei coinquilini dell'Erasmus, con cui ho condiviso mesi intensi e indimenticabili. Grazie per la compagnia, le risate e il sostegno reciproco che hanno reso quell'esperienza unica e irripetibile.

Infine, un ringraziamento va a tutti i compagni e le persone incontrate durante il percorso universitario: la vostra presenza, in aula o a distanza, ha reso le lezioni più leggere e lo studio un'esperienza condivisa e meno solitaria.

A tutti voi, grazie di cuore.
Senza il vostro supporto, questo traguardo non avrebbe avuto lo stesso significato.