Platform-Independent Cloud Computing Infrastructure Definition Model for Scalable Microservices Architectures

Camilo Ernesto Vargas Romero

Universidad Nacional de Colombia
Facultad de Ingeniería, Departamento de Ingeniería de Sistemas e Industrial
Bogotá, D. C., Colombia
2024

Platform-Independent Cloud Computing Infrastructure Definition Model for Scalable Microservices Architectures

Camilo Ernesto Vargas Romero

Thesis presented as a partial requirement for the degree of:
Master in Systems Engineering and Computer Science
«Magíster en Ingeniería - Ingeniería de Sistemas y Computación»

Advised by:
Jeisson Andrés Vergara Vargas, M.Sc

Research Field:
Software Engineering - Software Architecture
Research Group:
ColSWE - Software Engineering

Universidad Nacional de Colombia
Facultad de Ingeniería, Departamento de Ingeniería de Sistemas e Industrial
Bogotá, D. C., Colombia
2024

Dedicated to my family.

”Education is the most powerful weapon which you can use to change the world.”

- Nelson Mandela

Acknowledgments

I deeply appreciate my family for their unwavering support, commitment, and understanding throughout the development of this research project. Their support was crucial during moments when I was absent and preoccupied, and their presence has been an indispensable pillar in my academic journey.

I express my total gratitude to the Universidad Nacional de Colombia, which has consistently driven me towards excellence and quality. Being part of its academic community is and will continue to be a source of pride for me. Contributing to software engineering, the Colswe research group, and software architecture through this research not only enriches my experience but also creates opportunities for other students and researchers to significantly advance the academic and intellectual development of our alma mater.

I want to highlight and extend special thanks to my advisor, Professor Jeisson, who not only guided, advised, and accompanied me throughout the development of this research project but also has been a fundamental pillar in my professional and academic growth. His constant support and willingness to share his knowledge have been invaluable, and I am eternally grateful for his unwavering commitment to quality.

Finally, my acknowledgment to the authors of the cited documents, who provided the solid foundations that support the ideas presented in this research project. Their contribution is essential to the construction of knowledge and the robustness of this research.

Title in English

Platform-Independent Cloud Computing Infrastructure Definition Model for Scalable Microservices Architectures

Título en español

Modelo de definición de infraestructura de computación en la nube independiente de la plataforma para arquitecturas de microservicios escalables

Abstract

Software systems play a fundamental role in modern everyday life, and cloud computing has significantly expanded their reach while simultaneously reducing complexities and costs associated with their development and deployment. In this context, software architecture emerges as an essential component to ensure an optimal structure according to needs, defining innovative styles such as microservices architectures, which consist of a group of small, highly specialized services. When considering the implementation of these systems in the cloud, the challenge arises of defining the appropriate cloud computing infrastructure, especially regarding the configuration of scalability properties, a task often characterized by its complexity and duration. To address this complexity, an extension of the Architecture Description Language (ADL) Sarch has been undertaken, allowing the definition of scalability properties. Additionally, model-to-model transformation rules have been established to facilitate the transition from an architecture model to an infrastructure-as-code schema, specifically configured to address the scalability quality attribute. This innovative approach aims to simplify the task of configuring the necessary infrastructure, accelerating and automating the process to enable a faster and more efficient implementation of advanced microservices architectures in the dynamic environment of cloud computing.

Keywords: Software Architecture, Microservice Architecture, Infraestructure as Code, Model-Driven Engineering, Architectural Style, Cloud Computing, Architectural Description Language

Resumen

Los sistemas de software desempeñan un papel fundamental en la vida cotidiana moderna, la computación en la nube ha ampliado significativamente su alcance, al mismo tiempo que ha reducido complejidades y costos asociados con su desarrollo y despliegue. En este contexto, la arquitectura de software emerge como un componente esencial para garantizar una estructura óptima segun las necesidades, definiendo estilos innovadores como las arquitecturas basadas en microservicios, que están compuestas por un grupo de pequeños servicios altamente especializados. Al considerar la implementación de estos sistemas en la nube, surge el desafío de definir la infraestructura computacionl de nube adecuada, especialmente en lo que respecta a la configuración de propiedades de escalabilidad, una tarea que con frecuencia se caracteriza por su complejidad y duración. Para abordar esta complejidad, se ha llevado a cabo una extensión del Lenguaje de Descripción de Arquitectura (ADL) Sarch, permitiendo la definición de propiedades de escalabilidad. Además, se han establecido reglas de transformación modelo a modelo que facilitan la transición desde un modelo de arquitectura hacia un esquema de infraestructura como código, específicamente configurado para abordar el atributo de calidad de escalabilidad. Este enfoque innovador busca simplificar la tarea de configurar la infraestructura necesaria, acelerando y automatizando el proceso para permitir una implementación más rápida y eficiente de arquitecturas avanzadas en microservicion en el dinámico entorno de la computación en la nube.

Palabras clave: Arquitectura de Software, Arquitectura de Microservicios, Infraestructura como Código, Ingeniería Dirigida por Modelos, Estilo Arquitectónico, Computación en la Nube, Lenguaje de Descripción de Arquitectura

Esta tesis de maestría se sustentó el 10 de Abril de 2024 a las 09:00 a.m., y fue evaluada por los siguientes jurados:

Felipe Restrepo Calle (Phd.)
Profesor Asociado
Universidad Nacional de Colombia

Jairo Hernan Aponte Melo (Phd.)
Profesor Asociado
Universidad Nacional de Colombia

Contents

List of Figures

List of Tables

1 Introduction

With the increasing demand for different kind of software systems, the adoption of cloud computing has become a popular and efficient solution for computational infrastructure services. This market expansion has resulted in larger and more diverse data centers, incorporating various specialized hardware systems [8, 16]. However, this growth presents challenges for development teams and software architects tasked with creating and maintaining cloud infrastructures within distributed architectures, such as microservices architectures. Addressing critical aspects like predicting performance, monitoring, and resource provisioning decisions is imperative to ensure the development of high-quality software systems [36].

In this context, architects must make precise design and configuration decisions to effectively manage cloud infrastructure resources and capitalize on the scalability and flexibility benefits it offers. Scalability, a crucial quality attribute in software architecture, plays a pivotal role in handling the increasing number of users. Scalability refers to a software system's ability to adapt to changes in workload, whether in terms of requests or connected users [29]. This quality attribute greatly benefits from the elastic capabilities provided by cloud infrastructure.

This research aims to design a model that employs a set of model-to-model transformations to convert a microservices architecture design into an Infrastructure as Code (IaC) scheme, ensuring scalability and platform independence. Such a solution would significantly assist infrastructure teams and software architects in simplifying the task of configuring and managing cloud infrastructure.

To achieve this goal, extensive research was conducted, leading to the development of an approach that enables the automatic and accurate translation of components and interactions from a microservices architecture (MSA) design into a cloud infrastructure configuration. Model-to-model transformation techniques were explored, and a set of rules and patterns were established to efficiently and scalably define infrastructure. Additionally, existing methodologies and tools in the field of Infrastructure as Code were investigated and adapted to suit the specific needs of microservices architectures.

It is anticipated that this research will make a substantial contribution to the field of software architecture and cloud infrastructure by providing development teams and software architects with a powerful tool for designing and managing scalable infrastructures efficiently. By facilitating the configuration and management of cloud infrastructure, the elastic capabilities of cloud computing can be fully leveraged, ensuring high-quality software systems that can effectively adapt to changes in demand and business requirements.

1.1 Problem Statement

Despite having solutions for orchestration and abstraction in cloud computing infrastructure provisioning, especially in microservices architectures [25, 11], users still find themselves relying on manual configurations. Orchestration, involving the automated coordination of components or services to achieve specific goals, should ideally eliminate the need for manual intervention. However, this situation persists when scalability adjustments are required, either directly on the provider's platform or within infrastructure-as-code schemes. The utilization of these approaches entails a steep learning curve, demands expertise in handling these tools, and poses a higher risk of human errors.

Providing software architects with a model that allows them to define the cloud infrastructure of their systems would not only afford them greater flexibility but also increased control. This would enable agile adaptation to changing demands and the swift configuration of quality attributes, such as scalability, based on the specific needs of the system. Properly configuring cloud infrastructure and having a model to adjust scalability are valid considerations, given that swift decision-making regarding cloud resource configuration and provisioning is critical for the operation of modern software systems [36].

Addressing the stated problem demands further research and proposals related to the modeling of cloud computational infrastructure that ensures consistency with MSA (Microservices Architecture) and enables adjustments to the scalability configuration of cloud components and services. Possessing a model that abstracts the definition of cloud computational infrastructure and facilitates scalability configuration would allow organizations to reduce the time, resources, and costs required for executing a cloud infrastructure configuration and definition process.

The research question that is answered with the present work is the following:

How can you abstract and automate the provisioning of cloud computing infrastructure, allowing scalability configuration, and ensuring consistency with the design of a microservices-based software system?

1.2 Objetives

1.2.1 General Objetive

To build a model that allows defining elements of cloud computing infrastructure which a software system with microservices architecture will be deployed, guaranteeing scalability and platform-independence.

1.2.2 Specific Objetives

- To extend an architecture description language that supports the design of microservices architectures in order to allow the definition of scalability properties.
- To define a set of model-to-model transformations that allows generating an infrastructure as code schema from an instance of the architecture description language.
- To implement a software tool to automate the execution of the defined transformation set.
- To validate the proposed model through a case study in order to deploy a software system with microservices architecture and guarantee its scalability.

1.3 Contribution

The contributions of this thesis are defined in de following items:

- A domain-specific language extension for managing scalability properties in microservices architectures: an extension enhances an Architecture Description Language (ADL), called Sarch, to define scalability properties in microservices-based architectures.
- A set of model-to-model transformations: a collection of model-to-model transformation rules from MSA to IaC, employing Model-Driven Engineering techniques and leveraging architectural views.
- A tool for generating IaC schemas: an automated tool for executing model-to-model transformations, designed to enable end-users to apply this research in a productive environment.

1.4 Outline of the Thesis

Figure **1-1** visually depicts the purpose of this thesis. The development of this thesis is outlined in the following sections of this document: Chapter 1 introduces the research work developed in the thesis. Chapter 2, provides background information, mainly Software Architecture, Cloud Computing and Model-Driven Engineering. Chapter 3, describe the related works, Chapter 4 describe the model proposed in this thesis. Chapter 5, present details about the implemented transformation tool. Chapter 6, an evaluation of IaC schema generated from case study. Chapter 7, present the conclusions and future work.

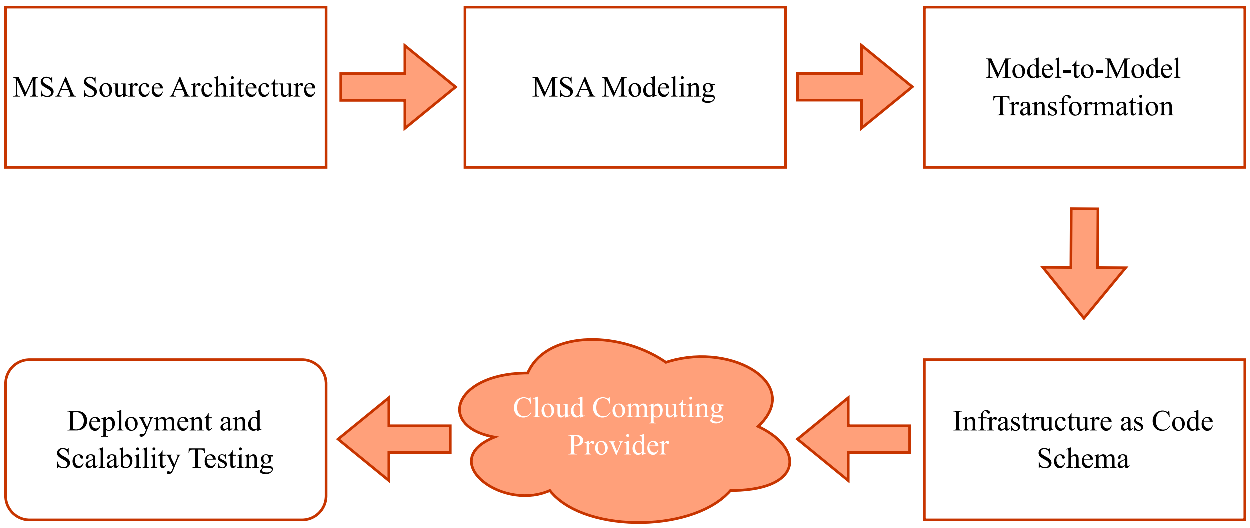

Figure 1-1: Graphical representation of the thesis proposal.

2 Background

The primary objective of this chapter is to provide a comprehensive overview of the most relevant topics driving the development of this research. In this regard, the intention is to offer a thorough and encompassing perspective that includes the most significant aspects related to the study. By doing so, the aim is to capture the essence and motivation underlying this investigative work.

2.1 Software Architecture

Software Architecture is a branch of software engineering that focuses on the architectural design of software systems [9]. It is defined as the set of key design decisions that constitute a system [21]. Furthermore, the software architecture of a system provides guidance for the major design decisions of the elements and components that make up the system and its evolution [1]. Similarly, Rozanski et al. in [29] describe software architecture as the structure of the system's structures, encompassing software elements, their relationships, and their properties.

The architecture of a software system consists of a collection of related elements and their properties. Elements represent the computational units of a system, playing a role at runtime [9]. Relationships denote the interactions or connections between the elements of the system [9], and properties provide additional information about the behavior and capacity of an architecture, simultaneously representing visible or measurable behaviors of the elements of the system [22].

The architecture of software systems is based on pre-existing architectures with a defined set of characteristics known as architectural styles. An architectural style can be defined as a specification of the most common structures in software systems applicable to a specific context or system [22]. A distinctive feature of these styles is their agnosticism towards both technology and domain [9].

Architectural patterns are a generalization of common architectural structures in software systems, applicable to specific problems within a defined set of contexts [29]. These patterns provide a proven structure and approach to address common challenges in the structural design of software systems. Some representative architectural patterns include Event-Driven Architecture, Pipeline Architecture, Model-View-Controller (MVC), Layered Architecture, and Hexagonal Architecture.

The fundamental difference between styles and patterns lies in the fact that the latter originate from recurring design problems within specific contexts, addressing particular characteristics, while styles simply adapt to the context without addressing specific design issues [7]. Some of the most relevant architectural styles include client-server, publish-subscribe (pub/-sub), event-driven architecture, layered architecture, Representational State Transfer (REST), Service-Oriented Architecture (SOA), and microservices architecture [22].

In the exploration of architectural styles, it is essential to consider the varied nature of applications and, in particular, distinguish between monolithic systems and other more distributed architectures. In a monolithic system, the bulk of the user interface and business logic layers are consolidated within a central component. This monolithic component houses the majority of the system's functionality, and its implementation occurs as a unified entity [10].

2.1.1 Microservices Architecture

This architectural style arises in response to the inherent limitations of monolithic architectures, addressing challenges related to maintainability, scalability, and resilience [27]. Its distinguishing feature lies in the adoption of microservices, encapsulating sets of functionalities within a system, organized around specific business domains. These discrete services operate autonomously and independently [24], engaging in inter-service communication. This architecture offers several advantages, including enhanced maintainability and more efficient scalability.

Key attributes encompass:

- Distributed Architecture: This approach disperses components and capabilities throughout an interconnected network, unlike monolithic structures. It provides increased flexibility and agility in handling evolving system demands by distributing functions across various nodes.

- Lightweight Protocol-based Communication [28, 27]: Microservices communicate via agile protocols, enabling efficient information exchange critical for real-time cooperation among diverse services. This effective communication drives synchronization and collaboration, ensuring fluid data flow and prompt responses to requests.

- Scalability Efficiency: A notable feature is its adeptness at scaling efficiently. Microservices permits the selective scaling of high-demand services, avoiding unnecessary expenses in areas with lower demand, unlike monolithic systems that uniformly scale all components.

- Autonomy and Independence [27]: The microservices architecture nurtures autonomy and independence for each service. Each microservice operates as an autonomous, self-contained entity, amenable to independent development, deployment, and maintenance. This expedites development and deployment, enabling teams to work concurrently on distinct microservices without mutual interference.

Detriments include:

- Elevated Technical Complexity [27]: The microservices architecture introduces complexities, particularly for those with limited technical acumen. Communication between microservices can grow intricate, elevating latency and necessitating meticulous planning. The requirement for intricate infrastructure to orchestrate and monitor services can augment the intricacy, demanding an advanced grasp of technical nuances and dedicated resources [33].

- Management Complexity: Segmenting an software system into autonomous microservices necessitates efficient management and coordination. Overseeing multiple components, spanning deployment, monitoring, and updates, can evolve into a complex task, warranting additional tools and processes [33].

2.1.2 Performance and Scalability

In software architecture the quality attributes are properties that can be compared and measured [2], allowing to evaluate how well the most important non-functional requirements of a software system are met. Some of the most notable attributes are availability, deployability, energy efficiency, integrity, performance and scalability, security and testability. This research work will specifically focus on the quality attribute of performance and scalability.

The quality attributes of performance and scalability refers to a system's ability to handle varying workloads in terms of number of requests, transactions, or processes, or to handle more complex workloads [29]. There are several strategies or tactics that can be used to improve performance and scalability as a quality attribute in the architecture of a software system. According to Bass et al. [2], these tactics can be divided into two main groups: those that control resource demand and those that manage resources, see the figure **2-1**.

From this group of tactics, we will work on the resource management branch with the tactic of resource increment, which means that when a system reaches the limits of its hardware, the available resources must be scaled up. In [29], two different ways of scaling the available hardware are presented:

1. Replacing the existing one with a higher capacity or vertical scaling one.

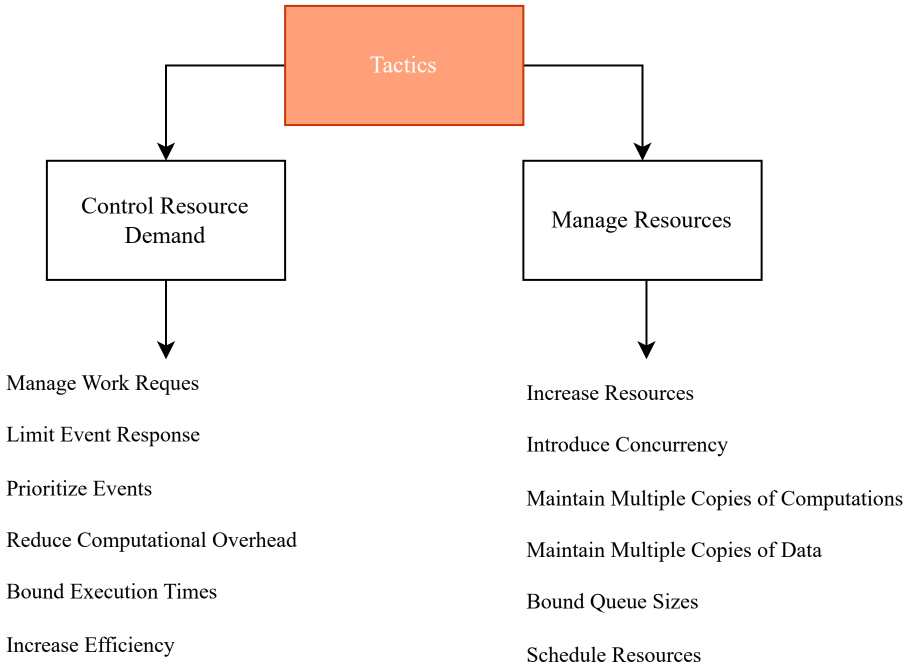

Figure 2-1: Performance tactics [2]

2. Adding more similar components or horizontal scaling.

These two scalability tactics are highlighted as useful for microservices architectures.

- Vertical scalability aims to add more resources to logical units (e.g., servers in a cluster) and add more resources to physical units (e.g., computer memory)[20] figure **2-2** .

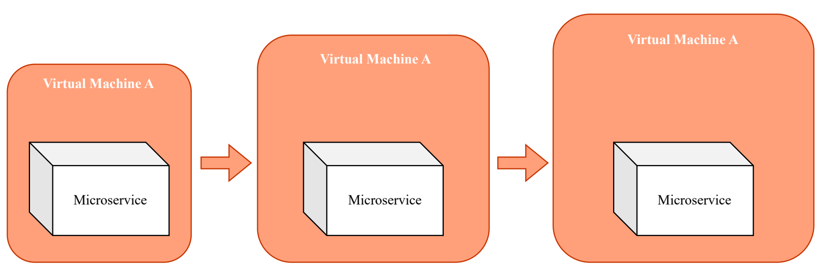

Figure 2-2: Vertical scalability

- Horizontal scalability aims to determine how many instances of each microservice are needed in execution, the architecture is capable of increasing the number of microservice instances to respond to different workloads, either reactively or proactively to achieve the desired QoS (quality of service) [20] figure **2-3**.

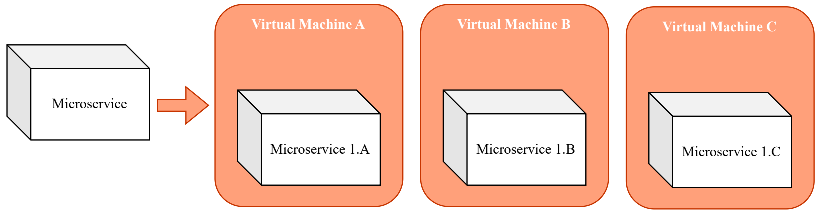

Figure 2-3: Horizontal scalability

2.2 Cloud Computing

Cloud computing, serving as a model for delivering computing services over the internet, can be categorized based on service domain and service types [26]. Concerning service domain, three main classifications exist: Public Cloud, where external entities provide computing resources to external users; Private Cloud, where enterprises establish and operate cloud services exclusively for their use; and Hybrid Cloud, which combines resources from both public and private clouds, allowing for secure network connections and flexible resource allocation [26]. These classifications offer insights into the diverse scope and functionalities within the realm of cloud computing.

In terms of service types, cloud computing encompasses Infrastructure as a Service (IaaS), providing users with virtualized computing resources without managing the physical infrastructure; Platform as a Service (PaaS), offering a comprehensive platform for application development and management; and Software as a Service (SaaS), delivering software services directly to end-users through the cloud, streamlining access and collaboration without local installations [26]. Each service type addresses specific needs, contributing to the overall efficiency and flexibility of cloud computing solutions.

In summary, the classification of cloud computing into service domains and service types offers a comprehensive perspective on available options, allowing organizations to choose strategies that align with their specific requirements.

There exist numerous cloud service providers and platforms in the market that offer public cloud services, with many providing comparable services, albeit with specific enhancements and focal points. Among the most utilized and distinguished platforms are AWS (Amazon Web Services), and GCP (Google Cloud Platform), Microsoft Azure, all of which will be the focus of scrutiny in this research project.

Amazon Web Services (AWS) is a globally distributed cloud services platform renowned for offering a comprehensive range of computing resources. These resources encompass storage, computing power, databases, machine learning, and more. AWS empowers organizations and developers to efficiently, scalably, and reliably host applications and run workloads without the need to manage physical resources. Its highly redundant and fault-tolerant infrastructure ensures high availability and resilience, making it a popular choice across businesses of varying sizes and industries for deploying IT infrastructures and storing data [17].

Google Cloud Platform (GCP), provided by Google, is a cloud services platform that furnishes a diverse array of computing resources and tools for storage, processing, analysis, and application development. With a globally distributed architecture and an interconnected network of data centers, GCP ensures high availability, performance, and scalability. Noteworthy features include cloud computing, data storage, databases, machine learning, Internet of Things (IoT), analytics, and more. GCP distinguishes itself through integration with leading Google technologies such as BigQuery, TensorFlow, and Kubernetes. Additionally, it offers flexible pricing options and billing models based on consumption, along with robust security and compliance tools to safeguard hosted data and applications [17].

Microsoft Azure, Microsoft's cloud services platform, presents a wide array of computing resources and tools for storage, processing, analysis, and application development. Similar to its counterparts, Azure boasts a globally distributed architecture and an interconnected network of data centers, ensuring high availability, performance, and scalability. Key services encompass cloud computing, data storage, databases, machine learning, Internet of Things (IoT), analytics, and more. Azure stands out for its seamless integration with Microsoft's technologies and tools, such as Azure Active Directory, Power BI, and Visual Studio. The platform also provides flexible pricing options and billing models based on consumption, coupled with robust security and compliance tools to uphold the protection of hosted data and applications [17].

2.3 Model-Driven Engineering

Model-Driven Engineering (MDE) is an approach that harnesses the advantages of modeling in software engineering. It comprises a set of concepts, notations, processes, rules, and tools, with the central focus on models and transformations [5]. Models serve to describe the reality of a system or situation and determine the details and scope for studying a problem. A model is composed of a metamodel, where a metamodel is a model that describes another model [5]. Transformations are a key component in MDE, allowing the definition of mappings between different models. Transformations occur from an input model to an output model. There are two types of transformation models [5]:

- Model-to-Model (M2M) transformations: In this kind of transformation, a model is utilized as the input parameters, and the outcome, following the transformation, becomes a different model.

- Model-to-Text (M2T) transformations: In this kind of transformation, a model is used as input parameters, and the result, post-transformation, is textual content.

These transformations are executed based on a set of specific transformation rules for each case study. Therefore, for every transformation, there are two phases: defining the transformation map for each element from one model to another (either a model or text), and automating the execution of these transformation rules. This ensures that the input produces a transformation as output [5]. In Model-Driven Engineering (MDE), models and transformations are represented using the advantages of modeling languages. A modeling language is a tool that allows the specification of system models in graphical or textual representations [5]. Domain-Specific Languages (DSLs) are languages designed for a specific domain, context, or company. They aid in simplifying tasks for individuals who need to describe things within that particular domain [5].

3 Related Work

This chapter presents a brief state of the art of the approaches, techniques, or alternative solutions that have approached the topic addressed in this research work from different viewpoints and techniques. In the literature, multiple previous works have been identified that have used model-driven engineering to manage cloud infrastructure resources based on their architecture, especially for microservice architectures.

This research work was based on the Systematic Literature Review (SLR) methodology developed by Kitchenham et al. [18], which provides a rigorous and structured approach to thoroughly analyze existing evidence in relevant literature. The adoption of this methodology aligns with the main objective of this study, which is to address the research question raised in Chapter 1.

In the initial stage, the scope and objectives of the review were outlined, focusing on the exploration of formal approaches to software architecture modeling, the definition of computational infrastructure in cloud environments, and the use of quality attributes in software architecture. To ensure the comprehensiveness of the search, systematic searches were conducted across various academic databases, including IEEE-Explore, Scopus, Science Direct, and Springer Link.The selection of studies was carried out by applying strict inclusion and exclusion criteria, which included the use of formal methods for software architecture modeling, explicit definition of computational infrastructure in the cloud, consideration of quality attributes in software architecture, and temporal relevance, prioritizing more recent studies.

With relevant studies identified, relevant data extraction was conducted, allowing for significant information retrieval on the approaches, techniques, and results presented in the reviewed literature. In the following chapter, the results of this systematic review will be presented in detail, along with the conclusions derived from this comprehensive analysis.

3.1 Literature Review

Through SLR, several works related to cloud infrastructure resource management based on microservices architecture were extracted. A synthesis of some previous works was conducted, which from different perspectives aim to provide solutions for cloud infrastructure resource management. Most of these works employ Infrastructure as Code (IaC) schemes as a central technique for defining and configuring the required computational resources in the cloud.

3.2 Review about Microservice Architectural Style

In literature, addressing resource management in microservices environments has been achieved through the implementation of Infrastructure as Code (IaC) schemes. This technique stands out as a primary approach for defining and configuring computational resources in cloud environments; however, it is not the only one.

The innovative approach of a study in the field of multi-cloud resource management for microservices-based architectures is highlighted, as presented in [15]. This study proposes a multi-cloud management framework designed to effectively integrate the APIs of multiple cloud providers through an API called CSP Pool, significantly enhancing the instance provisioning process. Furthermore, a Multi-Cloud Manager and a Multi-Cloud Controller are implemented to facilitate efficient cloud resource management, while the CSP Pool API ensures stability and reliability by excluding unreliable APIs. Through a use case, it is demonstrated that the use of the CSP Pool API reduces provisioning time, enabling faster deployment of microservices in a multi-cloud environment, resulting in a smoother and more reliable user experience.

Various solutions have been explored in existing works, leveraging the advantages of Model-Driven Engineering (MDE). Proposed solutions encompass modeling tools, model-based frameworks, domain-specific languages, and architecture description languages. This unified approach underscores the relevance of MDE in efficiently managing resources in cloud infrastructures, showcasing the application of diverse tools and techniques to address this challenge from multiple perspectives.

An exemplary instance is the modeling tool ARGON (Infrastructure Modeling Tool for Cloud Provisioning) introduced in [30]. ARGON specifies the final state of provisioning infrastructure and generates Infrastructure as Code schemas, facilitating the management of provisioning tools in the DevOps community. Notably flexible and robust, as discussed in [32] and [31], ARGON introduces extensions to handle scenarios such as cloud migration. ARGON emerges as a valuable tool for this research, establishing a solid foundation in model-driven engineering and the management of Infrastructure as Code schemas.

In [4], the "Cloud Camp" framework is introduced as a fast and scalable provisioning system based on models. This framework adheres to the standard topology and orchestration specification TOSCA and distinguishes itself by incorporating domain-specific modeling for cloud specifications. While beneficial for microservices-based architectures, the document does not specify the work done or planned for specific quality attributes, leaving areas of research open for future studies.

3.3 ADLs for Cloud Infrastructure Provisioning

Architectural Description Languages (ADLs) emerge as invaluable instruments for conceiving, describing, and analyzing complex software architectures, be it in microservices-based environments or cloud infrastructures. Through ADLs, it is possible to formalize the definition of components, connectors, configurations, as well as the relevant properties and constraints that shape an architecture.

Among the ADLs, ACME [13] stands out as a prominent tool used to assess non-functional attributes in traditional architectural styles such as Client-Server and Peer-to-Peer. ACME uniquely defines four fundamental aspects of architecture: structure, properties, constraints, and the types and styles that delineate architectural classes and families.

Another intriguing ADL is D-SDL [11], focused on describing business domains and services in microservices-based systems. This modeling language proves valuable for designing scalable and distributed system architectures. As part of the European project Archware, π-ADL [25] targets the specification of dynamic and mobile architectures. π-ADL rigorously formalizes the structural and behavioral aspects of architectures, using the load balancer architectural pattern to address scalability characteristics.

Some works propose extensions to previous efforts, such as SYSADL [19], highlighting an ADL that extends SysML to express descriptions of software architectures, specializing in key elements like components, connectors, and configurations. Additionally, AADL [12], a modeling language dedicated to describing real-time system architectures and embedded systems, enables the evaluation of quality properties such as performance, responsiveness, and system security. It is also compatible with analysis and simulation tools.

An important work to highlight is Sarch is presented as an Architectural Description Language (ADL) with the primary purpose of facilitating the description of software architectures [34, 35]. Its foundation is based on a set of architectural perspectives suggested by Clements et al. in the Views Beyond Catalog (VB) [9], which address crucial elements to support the creation of a software system. The architectural perspectives integrated into Sarch encompass Data, Component and Connector (CC), Layered, and Deployment views. Sarch's noteworthy capability to describe any software architecture, regardless of the adopted architectural style, highlights its versatility in this domain. Sarch has been tailored to perform model-to-model transformations fromthe Service-oriented Architecture (SOA) to the Microservices Architecture (MSA) style, as elucidated in [3].

Table **3-1** provides a comparative summary of related works, supporting the selection of Sarch as the optimal solution to achieve the objectives of this research. The other analyzed works have been of great significance, serving as crucial starting points and guides to arrive at the proposed solutions in this study concerning the definition of cloud computing infrastructure for Microservices Architecture (MSA)-styled architectures.

Table 3-1: Comparison of ADLs found

Name	Allow to use MSA?	Other architecture patterns?	Define quality attributes?	Generate files with the architecture model?	Can it be extended?
Sarch[35]	Yes	Yes	Yes	Yes	Yes
μσADL[25]	Yes	Yes	No	No	No
ACME[13]	No	Yes	No	No	No
D-SDL[11]	Yes	No	No	No	No
π ADL[25]	Yes	Yes	No	No	No
SYSADL[19]	Yes	Yes	Yes	Yes	No
AADL[12]	No	No	No	No	No

In summary, ADLs stand out as powerful tools for the description, design, and analysis of complex software architectures. They not only enable the formalization of component specifications, relationships, and configurations but also facilitate the evaluation of quality attributes and relevant constraints. Despite their valuable contributions, a literature review did not identify an ADL or previous work specifically equipped to abstract and automate the provisioning of cloud computing infrastructure, allowing scalability configuration and ensuring consistency with the design of a microservices-based software system. This gap underscores the need to develop and explore approaches that address the problem outlined in this research work.

4 Model definition

This chapter presents a detailed exposition of the model employed in defining and executing the necessary transformations to achieve the anticipated results in this research. Specific components of the ADL Sarch are meticulously described, highlighting their relevance in generating the inputs required for Model-to-Model transformations. This process is crucial for yielding a final outcome of an Infrastructure as Code schema.

4.1 Definition

The aim of this model is to generate an Infrastructure as Code schema from an architectural description model of a microservices-based software system. Figure **4-1** provides a comprehensive overview of the model. It takes certain input variables and, following the execution of M2M transformations, produces specific outputs:

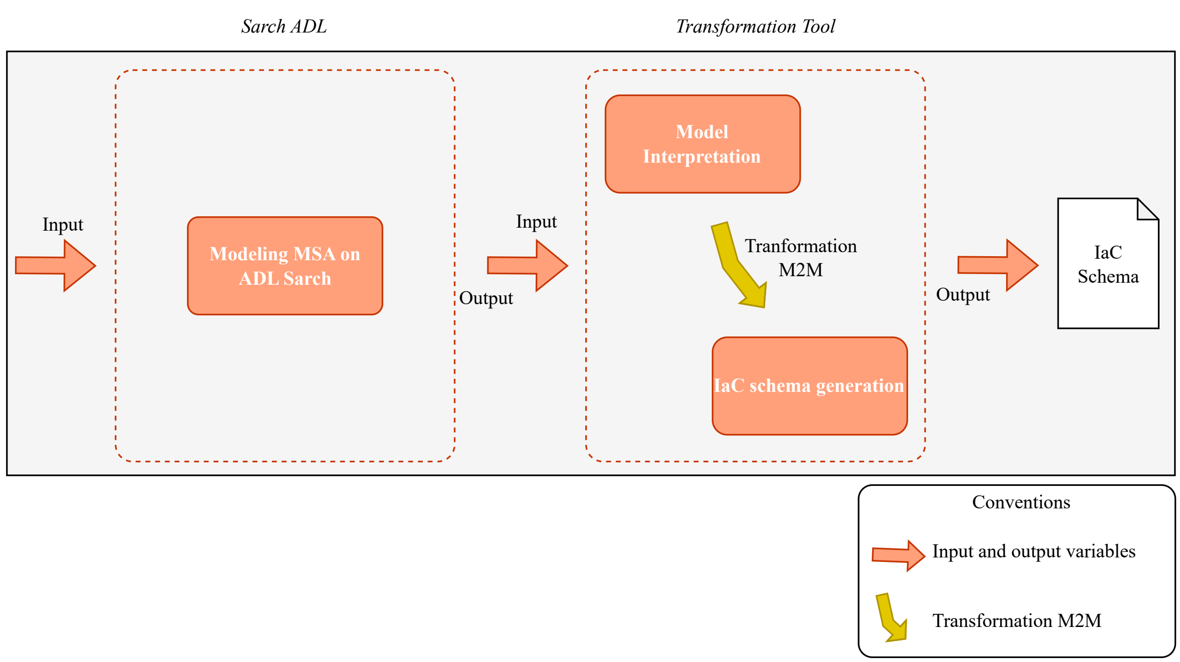

Figure 4-1: Graphical representation of the model.

The model's model-to-model transformations encompass an extension of the grammar of the previously described Sarch ADL, enabling the definition of specific scalability properties for

Microservices Architectures (MSA). These model-to-model transformations progress from MSA to an Infrastructure as Code schema, with the initial model articulated using the extended Sarch language.
As depicted in Figure **4-1**, the transformation model comprises the following steps:

- The architecture of a software system with an MSA style is delivered as an outcome.
- The system architecture is modeled in the component and connector view.
- The system architecture is modeled in the deployment view.
- The model generated from the Sarch language serves as input to a set of model-to-model transformations.
- Model-to-model transformations are executed to generate an Infrastructure as Code (IaC) schema as the output variable.

4.2 Model Inputs and Outputs

The input and output variables for this model are fundamental as they enable the recognition of the system's initial state and the outcomes of the model. Variables:

Variable	**Description**	**Function**
Component and Connector view	Presents elements that have a presence at runtime, such as components, processes, services, client objects, servers, and data stores. Groups components and connectors [9].	Input
Deployment view	View describing hardware and network infrastructure elements where software elements are located and their relationships [9].	Input
IaC Schema	Schemas for managing and provisioning computing infrastructure through code scripts related to the software system modeled in Sarch. The term IaC will be further elaborated in Chapter 5.	Output

Table 4-2: Definition of input and output model variables

The input variables in this model, represented by the component and connector view, as well as the deployment view, allow for the identification of the software system model intended for deployment in the cloud computing infrastructure of a selected cloud provider. On the other hand, the output variable, presented as the Infrastructure as Code (IaC) schema, provides the scope and outcomes of the model after the execution of model-to-model transformations.

4.3 Model Driven Engineering Approach

Beginning with the definition of the provided model and the specification of inputs and outputs, an approach of Model-Driven Engineering (MDE) is adopted, as described in [5]. This approach requires two fundamental elements: a Domain-Specific Language (DSL) and a set of transformation rules. The need for a context for modeling through the DSL is established, and in this case, the context corresponds to a Microservices Architecture (MSA) modeled through the Architecture Description Language (ADL) Sarch. Additionally, a goal is defined that will be achieved through transformations; in this research work, the objective is to generate a model representing an Infrastructure as Code (IaC) schema.

Thanks to the advantages offered by Sarch in terms of the breadth of architectural styles, technology, and versatility for modeling software architectures, this Architecture Description Language (ADL) is chosen to be used and extended within the framework of this research work, aiming to achieve the set objectives for solving the stated problem. In this manner, to generate model-to-model transformations resulting in an Infrastructure as Code (IaC) schema, the Component and Connector views of the Sarch ADL are used as input. The transformation functions take the elements from each view and their relationships to generate a set of modules or resources within the Infrastructure as Code schema. The following are key aspects of the Architecture Description Language (ADL) Sarch that are employed in the model-to-model transformations:

- An environmental element in the deployment view can be represented as a hardware element from two perspectives: first, as an inherent element of the cloud provider such as a node or dedicated service, and second, within a cluster as an instance of the same.

- An environmental element in the deployment view is an Infrastructure as Code (IaC) module or resource.

- Environmental elements in the deployment view may have horizontal scaling characteristics, detailed in Chapter 5. These will be represented in terms of the maximum and minimum number of replicas or the automatic scaling capacity of a cloud infrastructure cluster or instance.

- From the Component and Connector view, connections for the api_gateway and load_balancer services are extracted. These connections will be represented as services within a Kubernetes cluster, and such configurations will be incorporated into the final Infrastructure as Code (IaC) schema.

In Sarch, the Component and Connector (CC) view contains elements of various types, such as components, connectors, ports, etc., as needed. In this research work, components of a Microservices Architecture (MSA) are software elements (se), such as microservices, databases, load balancers, API gateways, storage, caches, and web applications. These software elements encompass a wide variety of MSA architecture cases and are essential for the model-to-model transformation. Additionally, these elements are used in the deployment view, as they are deployed in an environmental element (ee) that has the capacity to maintain them during runtime. These types of environmental elements (ee) cover a wide range of cases in MSA architectures. The table **4-3** presents the compatible components and connectors for the smooth flow of transformations, representing a selection of the most common components in microservices-based architectures [23].

Sarch Element	Element Type	Description
Component	microservice	Small, independent service.
	database	Data persistence system.
	loadbalancer	Distributes incoming network traffic.
	apigateway	Facilitates communication between microservices and external clients.
	storage	General-purpose storage system.
	cache	Stores frequently accessed data for quick retrieval.
	webapplication	User interface for web-based applications.
Connector	http	Hypertext Transfer Protocol connector.
	ftp	File Transfer Protocol connector.
	rest	Representational State Transfer connector.
	db_connector	Database connector.

Table 4-3: Supported Components and Connectors

In the deployment view, elements of various types are presented. For this research work, types such as container, cache service, storage service, load balancing service, cluster, relational database service, NoSQL database service, node, and API gateway are considered. Table **4-4** outlines the types of environmental elements and relationships that will be supported for the proper flow of transformations.

Search Element	Element Type	Description
Environmental Element Type	Container	An isolated environment for running applications.
	cache_service	A service that stores data in memory for quick access.
	storage_service	A service providing persistent storage for data.
	loadbalancer_service	A service that distributes workload among servers for efficiency.
	cluster	A set of interconnected nodes, e.g., like Kubernetes.
	relational_db_service	A service that manages structured data in a relational system.
	NoSQL_db_service	A service that manages unstructured or semi-structured databases.
	apigateway_service	A service acting as an intermediary for communication between applications and services.
	node	A compute service or virtual machine.
Relation	contains	Relation between two environmental elements.
	deployed_in	Relation between software elements and environmental elements.

Table 4-4: Supported Deployment Elements

4.4 Transformation Rules

The tables **4-5**, **4-6**, **4-7** and **4-8** and figures **4-2**, **4-3**, **4-4**, **4-5**, **4-6** and **4-7** present the definition of model-to-model transformation rules from an MSA model generated in SARCH, resulting in elements within an Infrastructure as Code (IaC) schema for two cloud service providers (AWS, GCP).

4.4.1 Stateless Environmental Elements Transformation Rules

The table **4-5** and figure **4-2** outline transformation rules for elements within a stateless software architecture. The included elements are microservices, web applications, and API gateways. These rules will guide the transformation process of the architecture, ensuring consistency and effectiveness within the context of stateless systems.

Element	Sarch	Terraform AWS	Terraform GCP
microservices web-application api-gateway	se(stateless_service) deploy_in ee(cluster)	terraform_-K8S_deployment	terraform_-K8S_deployment
	se(stateless_service) deploy_in ee(Node)	terraform_-EC2	terraform_-compute_-engine
	se(stateless_service) deploy_in ee(container) & ee(cluster) contains ee(container)	terraform_-K8S_deployment	terraform_-compute_-engine
	se(stateless_service) deploy_in ee(container) & ee(Node) contains ee(container)	terraform_-EC2	terraform_-compute_-engine

Table 4-5: Tranformation Rules State Less - Enviromental Element

The flow followed by the transformation rules defined in table **4-5** for stateless software elements is composed of a series of steps and conditions. This results in two types of elements in the Infrastructure as Code (IaC) schema: a virtual machine (VM) or a Kubernetes (K8s) deployment.

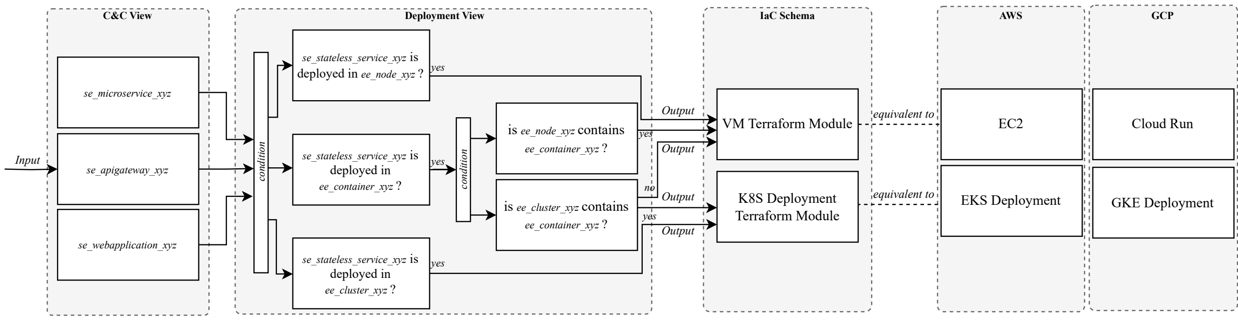

Figure 4-2: Stateless transformation diagram.

Steps:

1. In the workflow, the model of a microservices-based software architecture (MSA) is introduced. Software elements (se) or components in the Components and Connectors (C&C) view that are stateless, such as microservices, API gateways, and web applications, are extracted.

2. The first condition checked is the environment element (ee) in which the software element (se) will be deployed. It is possible to deploy these software elements in three types of environment elements: a node, a container, or a cluster.

3. If the software element (se) is deployed on a node-type environment element (ee), the result will be an Infrastructure as Code (IaC) module referencing a virtual machine (VM) based on the selected provider.

4. If the software element (se) is deployed on a container-type environment element (ee), a second condition is validated:
   - If the container is contained in a node, the result will be an IaC module referencing a virtual machine (VM) based on the selected provider.
   - If the container is contained within a cluster, the result will be an IaC module referencing a Kubernetes (K8S) deployment based on the selected provider.

5. If the software element (se) is deployed on a cluster-type environment element (ee), the result will be an IaC module referencing a Kubernetes (K8s) deployment based on the selected provider.

The modules in the IaC schema generated in this workflow reference either a virtual machine (VM) or a Kubernetes (K8s) deployment, which will be configured and tailored to the selected provider. For AWS, an EC2 module and an EKS deployment will be used, while for GCP, a Cloud Run module and a GKE deployment will be employed.

4.4.2 Api Gateway Element Transformation Rules

While the Api Gateway is encompassed in the transformations of stateless software elements, two additional rules are added in table **4-6** and figure **4-3** to achieve a broader coverage of possible scenarios. In this case, the transformation results in an Api Gateway service module and a Kubernetes (K8s) Ingress module.

Element	Sarch	Terraform AWS	Terraform GCP
api gateway	se(api_gateway) deploy_in ee(api_gateway_service)	terraform_-API_Gateway	terraform_-GCP_AG
	se(api_gateway) deploy_in ee(cluster)	terraform_-K8S_ingress	terraform_-K8S_ingress
	se(api_gateway) deploy_in ee(api_gateway_service) & ee(cluster) contains ee(api_gateway_service)	terraform_-K8S_ingress	terraform_-K8S_ingress

Table 4-6: Tranformation Rules Api Gateway - Enviromental Element

The flow followed by the transformation rules defined in table **4-6** consists of a series of steps and conditions, resulting in two types of elements in the Infrastructure as Code (IaC) schema: an Api Gateway service module or a Kubernetes (K8s) Ingress module based on the selected provider.

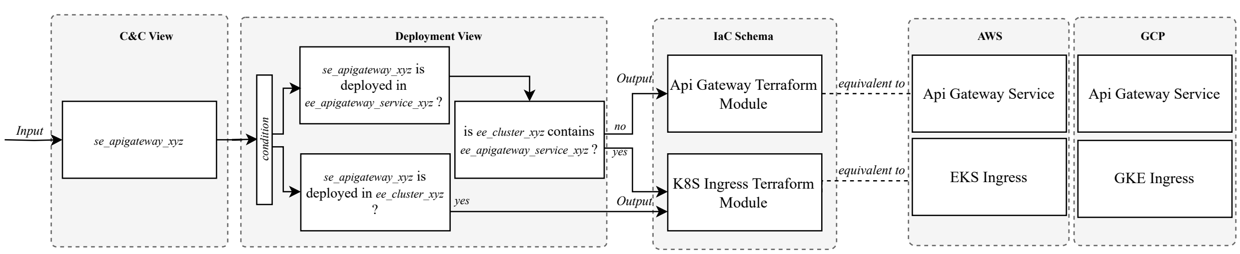

Figure 4-3: Api Gateway transformation diagram.

Steps:

1. In the workflow, the model of a microservices-based software architecture (MSA) is introduced. Software elements (se) or components in the Components and Connectors (C&C) view are extracted. If the software element (se) is of type apigateway, it is possible to deploy this type of element in two types of environment elements, a cluster or a provider-specific api gateway service.

2. If the software element (se) is deployed in an environment element of type api gateway service, the following condition is validated:

- If the api gateway service is contained in a cluster, the result will be an IaC module referencing a Kubernetes (K8s) Ingress based on the selected provider.
- If the api gateway service is not contained in a cluster, the result will be an IaC module referencing a provider-specific Api Gateway service.

3. If the software element (se) is deployed in an environment element of type cluster, the result is an IaC module referencing a Kubernetes (K8s) Ingress.

The modules generated in this flow reference an AWS and GCP-specific Api Gateway service, and an EKS Ingress or a GKE Ingress service based on the selected provider.

4.4.3 Load-balancer Environmental Element Transformation Rules

In the table **4-7** and figure **4-4**, detailed transformation rules are provided specifically for the load balancer environment element. These guidelines aim to ensure the effective deployment and functionality of the load balancer in the targeted environment, optimizing its performance and contribution to the overall system architecture.

Element	**Sarch**	**Terraform AWS**	**Terraform GCP**
Load Balancer	se(loadBalancer) deploy_in ee(load_balancer_service)	terraform_-AWS_LB	terraform_-GCP_LB
	se(loadBalancer) deploy_in ee(load_balancer_service) & ee(cluster) contains ee(load_balancer_service)	terraform_-K8S_service	terraform_-K8S_service
	se(loadBalancer) deploy_in ee(container) & ee(Node) contains ee(container)	terraform_-EC2	terraform_-compute_engine

Table 4-7: Transformation Rules Load Balancer - Environmental Element

The rules defined in table **4-7** follow a flow that consists of steps and conditions, resulting in three types of elements in the Infrastructure as Code (IaC) schema: a provider-specific load balancer module, a Kubernetes (K8s) service, and a virtual machine (VM) module.

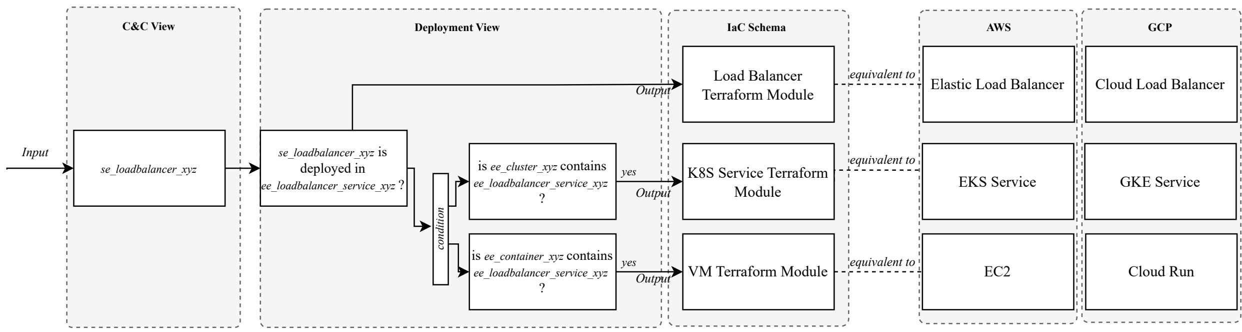

Figure 4-4: Load Balancer transformation diagram.

Steps:

1. In the workflow, the model of a microservices-based software architecture (MSA) is introduced. Software elements (se) or components in the Components and Connectors (C&C) view are extracted. If the software element (se) is of type Load Balancer, a condition is validated.

2. If the software element (se) is deployed in a load balancer service and is not contained within any other environment element, the result is a Load Balancer service IaC module based on the selected provider. If the environment element (ee) is contained within another environment element (ee), two additional conditions are validated:

   - If the environment element (ee) is contained within a cluster-type environment element (ee), the result is an IaC module referencing a K8s service.
   - If the environment element (ee) is contained within a container-type environment element (ee), the result is an IaC module referencing a virtual machine from the provider.

The modules generated in this flow reference an AWS and GCP Load Balancer service, a K8s service module representing EKS for AWS, and GKE for the GCP provider. Additionally, a virtual machine module represents an EC2 instance for AWS and Cloud Run for GCP.

4.4.4 Storage Environmental Elements Transformation Rules

In the table **4-8** and figures **4-5**, **4-6**, **4-7** and **4-8**, detailed transformation rules are outlined for storage environment elements, encompassing cache, databases, and storage components. These rules provide specific guidance on how to handle the transformation of each of these crucial elements within the architectural context.

Relational Database:

Element	Sarch	Terraform AWS	Terraform GCP
Relational database	se(relational database) deploy_in ee(relational_-database_service)	terraform_-RDS	terraform_-CloudSQL
NoSQL database	se(NoSQL database) deploy_in ee(nosql_database_-service)	terraform_-Dy-namoDB	terraform_-DataStore
storage	se(storage) deploy_in (storage_service)	terraform_-S3	terraform_-cloudStor-age
cache	se(cache) deploy_in (cache_service)	terraform_-Elastic-Cache	terraform_-MemoryS-tore

Table 4-8: Tranformation Rules Storage - Enviromental Element

The rules defined in table **4-8** for components of relational database type follow a flow with a series of steps and conditions, resulting in an Infrastructure as Code (IaC) module that references a database service based on the selected provider.

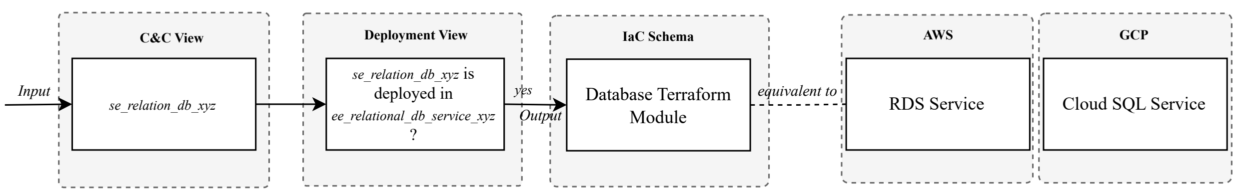

Figure 4-5: Relational Database transformation diagram.

Steps:

1. In the workflow, the model of a microservices-based software architecture (MSA) is introduced. Software elements (se) or components in the Components and Connectors (C&C) view are extracted. If the software element (se) is of type relational database, a condition is validated.

2. If the software element (se) is deployed in an environment element (ee) of type relational database, the result is a module of a relational database in the Infrastructure as Code (IaC) schema based on the selected provider.

The database module generated in this flow references an RDS service for AWS or a Cloud SQL service for GCP, depending on the selection.

NoSQL Database:

In table **4-8**, transformation rules for NoSQL database components are presented. These rules follow a flow of steps and conditions that result in an IaC module configured and tailored to the selected provider.

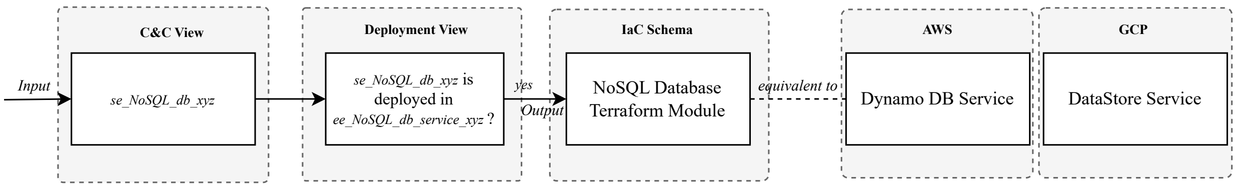

Figure 4-6: NoSQL Database transformation diagram.

Steps:

1. If the software element (se) in the Components and Connectors view is of type NoSQL database, a condition is validated.

2. If the software element (se) is deployed in an environment element (ee) of type NoSQL database service, the result is an IaC module referencing a NoSQL database service.

The generated NoSQL database module in this flow references the DynamoDB service for AWS or the Datastore service for GCP, depending on the selection.

Storage

Within the rules defined in table **4-8**, those related to storage-type components can be found. These transformations follow a flow of steps and conditions that result in a configured and tailored IaC module based on the selected provider.

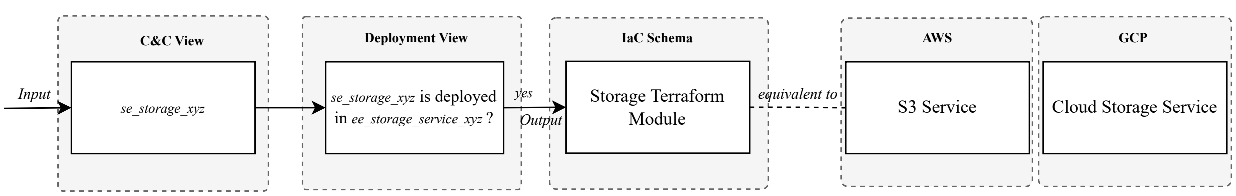

Figure 4-7: Storage transformation diagram.

Steps:

1. In the workflow, the model of a microservices-based software architecture (MSA) is introduced. Software elements (se) or components in the Components and Connectors (C&C) view are extracted. If the software element (se) is of type storage, a condition is validated.

2. The software element (se) is deployed in an environment element (ee) of type storage service, and the result of the flow is an IaC module configuring storage based on the selected provider.

The IaC module for storage references an S3 service in AWS and a Cloud Storage service in GCP.

Cache

In the last section of table **4-8**, rules for cache-type components are defined. These rules follow a flow of steps and conditions that enable the generation of an IaC module to define and configure the infrastructure required to support this component.

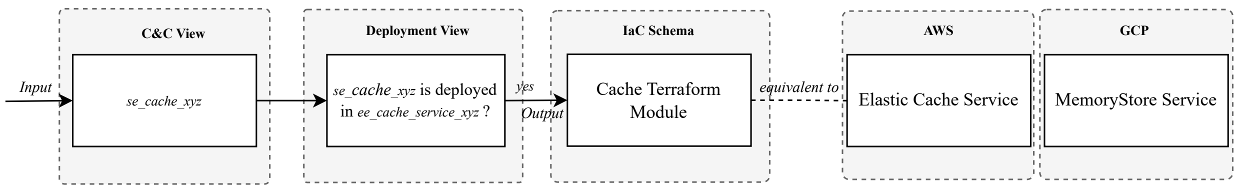

Figure 4-8: Cache transformation diagram.

Steps:

1. In the workflow, the model of a microservices-based software architecture (MSA) is introduced. Software elements (se) or components in the Components and Connectors (C&C) view are extracted. If the software element (se) is of type cache, a condition is validated.
2. If the software element (se) is deployed in an environment element (ee) of type cache service, the result is an IaC module referencing a cache service based on the selected provider.

The cache service module generated in the IaC schema in this flow references Elastic Cache services for AWS and Memory Store Service for GCP.

5 Transformation Tool

In this chapter, a detailed exposition is provided on the implementation of a tool designed to automate the execution of the model-to-model transformations discussed in previous chapters. This tool serves as a central component and operates in harmony with the presented model. It functions as the entry point for input variables, executes model-specific transformations, and yields the corresponding output variables. This tool plays a crucial role in the model workflow, facilitating the efficient integration of input data and ensuring coherence in the transformation process.

5.1 Infrastructure as Code (IaC)

Infrastructure tools such as Terraform, Chef, and Puppet play a pivotal role in the efficient management and configuration of cloud environments. Widely recognized and powerful, these solutions empower infrastructure teams and software architects to define and automate the configuration of resources and services in scalable cloud environments. Each of these tools offers distinctive approaches and features, providing flexibility for the implementation and administration of infrastructures. Their popularity and versatility position them as valuable choices for the effective orchestration of cloud resources and the facilitation of infrastructure operations.

Chef[1] and Puppet[2] are configuration tools specifically designed to handle software on existing computational resources. They provide a robust framework for automating tasks related to infrastructure [14].

On the other hand, Cloud Formation[3] and Heat are tools that also allow the representation of infrastructure as code. However, their focus is more directed towards specific cloud providers. Cloud Formation is used to provision resources in AWS, while Heat operates exclusively in the OpenStack API environment [14]. While valuable, these tools may have limitations in terms of portability and flexibility across different cloud providers.
In the context of Terraform, developed by HashiCorp, it stands out for its ability to interact with various cloud providers such as AWS, GCP, AZURE, among others. This interoperability is

[1]https://www.chef.io/
[2]https://puppet.com/
[3]https://aws.amazon.com/cloudformation/

achieved through the use of plugins called providers, allowing communication with the underlying APIs of different providers. Terraform offers the flexibility to configure the desired provider easily using the "provider" block type in the configuration file. Moreover, Terraform not only supports multiple providers but also facilitates simultaneous work with several of them, providing capabilities for multi-cloud provisioning [14].

In the context of this research work, the objective is to generate infrastructure-as-code schemas for a group of cloud providers. That's why the use of Terraform and its capabilities is well-suited to meet the requirements.

5.2 Kubernetes

Abbreviated as K8s, Kubernetes stands out for its exceptional scalability and deployment capabilities [6]. Designed to efficiently manage applications in distributed environments, Kubernetes offers a modular and extensible architecture that enables organizations to adapt to the increasing demands of their workloads. One of Kubernetes' key properties is its automatic horizontal scalability. By utilizing mechanisms like dynamic creation and elimination of container replicas, Kubernetes can automatically adjust the amount of resources allocated to an application based on workload. This allows for optimal resource utilization and ensures that applications can efficiently handle sudden increases in demand without service disruptions.

Furthermore, Kubernetes offers significant advantages over other container orchestration services. Its broad community of developers and users, backed by the Cloud Native Computing Foundation (CNCF), ensures a steady pace of innovation and a strong knowledge base. Kubernetes' portability is also noteworthy, as it can be deployed in various environments, including public and private clouds, and even on-premises setups, enabling businesses to avoid vendor lock-in and maintain greater control over their infrastructure. Kubernetes' automation capabilities also accelerate the application deployment and operation process, reducing development and deployment time, and facilitating the adoption of DevOps practices.

In the context of this research, some components specific to Kubernetes (k8s) are highlighted, such as:

- **Pod:** It is the fundamental deployment unit that encapsulates one or more related containers, sharing resources and network space. Pods enable consistent application management, facilitate communication between containers, and allow scalability and availability by being easily replicable and replaceable as per application needs.

- **Node:** Refers to an individual machine within the cluster's computing infrastructure. Nodes host and run containers, serving as the execution environment for pods, which are the basic deployment units for applications. Kubernetes manages nodes and allocates resources like CPU and memory for the scheduled pods, ensuring load balancing and performance across the cluster.

- **Deployment:** Is a declarative resource that defines the desired state of a set of identical pods, allowing for easy management of application updates and scaling. It ensures the specified number of pod replicas are running and handles rolling updates or rollbacks, making application maintenance more controlled and automated.
- **Service:** Is an abstraction that provides a stable network endpoint for accessing a dynamic set of pods. It enables load balancing and discovery of pods, allowing applications within the cluster to communicate with each other or external entities consistently, even as pods are created or replaced.

Kubernetes (K8s) offers effective strategies for managing its scalability properties, leveraging tools like the Horizontal Pod Autoscaler (HPA) and the Cluster Autoscaler.

HPA

The Horizontal Pod Autoscaler (HPA) is a dynamic scaling feature within Kubernetes designed to optimize resource utilization and enhance application performance. HPA automatically adjusts the number of pod replicas in a deployment or replica set based on real-time metrics such as CPU utilization or custom-defined metrics. When the observed metrics exceed or fall below specific thresholds, HPA triggers scaling actions. It can increase the number of replicas to meet higher demand or decrease them during periods of lower load. This elasticity ensures that applications efficiently handle varying levels of traffic, avoiding overprovisioning of resources, reducing operational costs, and maintaining consistent responsiveness. This behavior is illustrated in the image **5-1**.

Cluster Autoscaler

The Cluster Autoscaler is a key component in Kubernetes that automates the scaling of the entire cluster's capacity. It dynamically adjusts the number of worker nodes in response to changes in resource demands. If certain pods cannot be scheduled due to resource constraints, the Cluster Autoscaler initiates node scaling to accommodate the pending workloads. Similarly, if nodes are underutilized, the autoscaler can remove unnecessary nodes, optimizing resource allocation and cost efficiency. This ensures that the cluster size aligns with the application's requirements, providing seamless scalability and efficient resource utilization. This behavior is illustrated in the image **5-2**.

Thanks to the advantages that Kubernetes (K8s) offers in terms of scalability and its affinity with Microservices Architecture (MSA), its implementation will be considered to apply the scalability

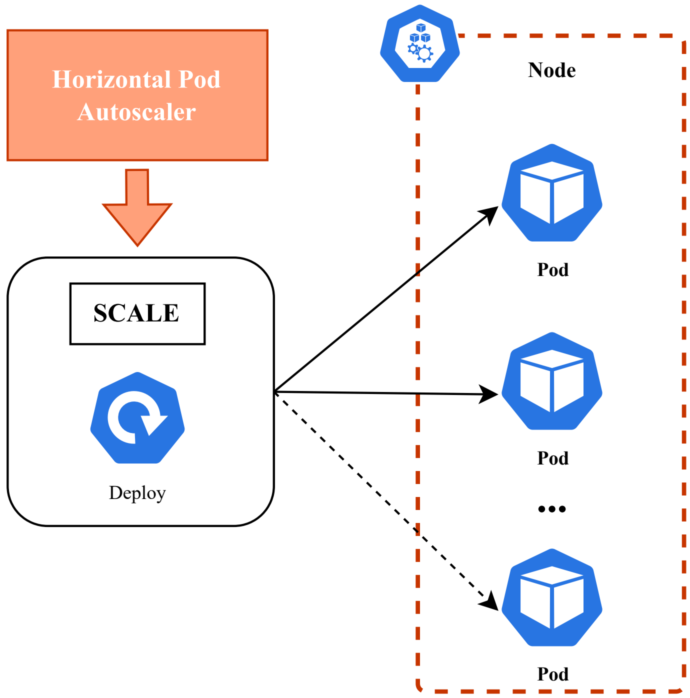

Figure 5-1: Diagrama del Horizontal Pod Autoscaler

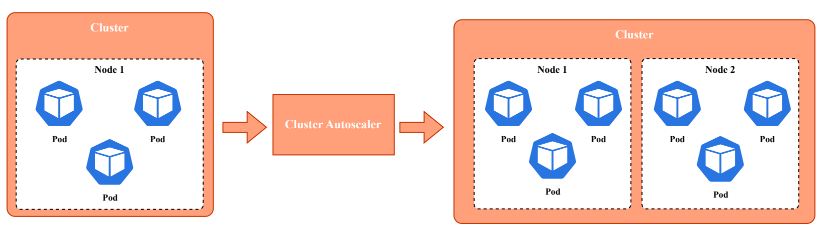

Figure 5-2: Cluster Autoscaler Diagram

properties recently added to Sarch. This type of cloud infrastructure will be especially used for systems with MSA that require deployment in a cluster. Additionally, Kubernetes' advantages in multicloud environments make it an ideal environment to achieve the goals set in this research.

5.3 Implementation

A graphical interface-equipped tool has been developed to facilitate model input. This instrument allows the selection of a specific set of cloud service providers. Depending on the chosen provider, the tool generates a set of files containing an Infrastructure as Code (IaC) schema. The implementation of this tool was carried out using the JavaScript programming language for general purposes, within the NodeJS framework environment.The resulting graphical interface has been carefully designed to be user-friendly and intuitive for the end users, who are understood as members of development or engineering teams aiming to define cloud infrastructure for their software systems. To see the tool in action, a graphical representation is provided in Figure **5-3**, where the two alternatives for loading the architecture model of a microservices-based software system are presented.

Figure 5-3: Transformation Tool Interface

Figure **5-4** illustrates the option to choose from several available cloud service providers. For each of these providers, it is possible to generate an Infrastructure as Code (IaC) schema.

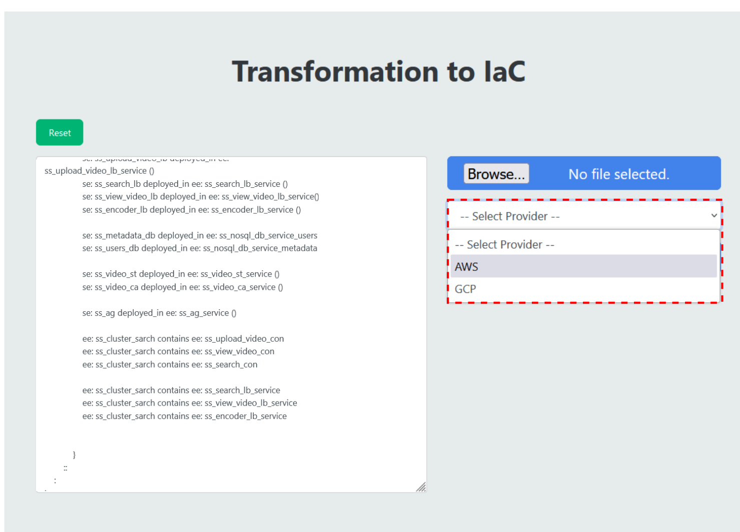

Figure 5-4: Transformation Tool Interface - Provider Selection

The key technology that enabled the extension of SARCH is Langium4. Langium not only acts as the extension component but also serves as the backend of the tool, automating the execution

4https://langium.org/docs/

of model-to-model transformations. It is an open-source tool designed for language generation, compatible with the Language Server Protocol. Langium is developed in TypeScript and runs in the NodeJS framework.

The developed transformation tool consists of two components: a frontend and a backend. This separation allows for independent management of the logic of each component and leverages the capabilities of Langium to automate the execution of transformations, defined in previous chapters, in the backend component.

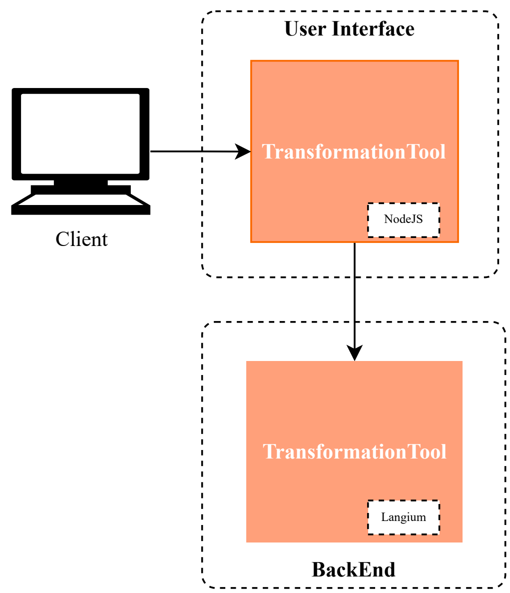

Figure 5-5: Transformation Tool Diagram

In the backend component, transformation rules are defined in TypeScript language to allow input of a model of an MSA-modeled system in Sarch and generate an Infrastructure as Code (IaC) schema as a result. These transformation rules are meticulously developed and supported by the Abstract Syntax Tree (AST) created by Langium from the Sarch ADL.

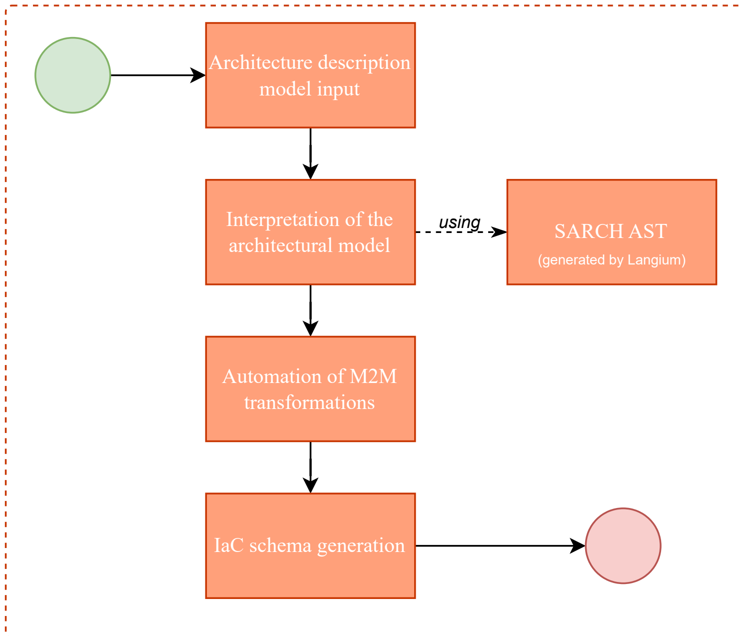

Figure 5-6: Transformation Tool Flow Diagram

Figure **5-6** details the implemented flow to automate the transformation from MSA to IaC, following these steps:

1. Architecture description model input: The system user inputs a file or plain text containing the architectural model generated from the Sarch ADL.

2. Model interpretation: Using the Sarch AST, the input model is interpreted to identify elements and relationships from the Components and Connectors view, as well as the Deployment view.

3. Execution of transformations: In this stage, transformation rules defined in previous chapters are executed.

4. Generation of IaC schema: The IaC files generated in the previous step are grouped and delivered as the final result of the flow.

5.4 Sarch Extension

The Sarch ADL has been extended to encompass aspects related to the scalability quality attribute for Microservices Architecture (MSA). In the definition of its deployment views, parameters have been added to specify the number of minimum and maximum replicas, along with the auto-scalability attribute.

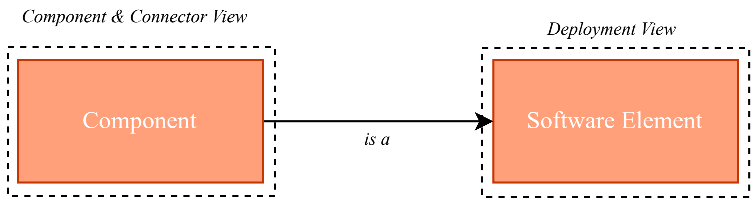

Figure 5-7: Software Element on Sarch

The parameters are added to the software elements, which represent components of the components and connectors view as depicted in figure **5-7**.

Listing 5.1: Definición de SoftwareElement

```
SoftwareElement :
'software_element ' '(' componentElement =[ ComponentElement :ID] ')'
name=ID ('('
(' min_replicas :' minReplicas =INT)? # minimum replica number
(' max_replicas :' maxReplicas =INT)? # maximum replica number
('autoscaling ' podAutoscaler =ID)? # Autoscaling parameter
') ')?
;
```

- min_replicas: This parameter, presented in Listing 5.1, allows defining the minimum number of replicas that will be present in the infrastructure for a software element. The elements possessing this characteristic are detailed in table **5-1**.

- max_replicas: This parameter, presented in Listing 5.1 enables the definition of the maximum number of replicas that the cloud infrastructure can accommodate for a software element. The elements with this characteristic are detailed in table **5-1**.

- autoscaling: This attribute, presented in Listing 5.1 represents a property that enables auto-scaling the infrastructure for a software element. It requires the definition of both minimum and maximum replicas; if not defined, default values will be assumed. The elements with this characteristic are detailed in table **5-1**.

Parameters related to scalability are added to the environment elements, which represent an infrastructure element in which a software component will be deployed as depicted in figure **5-8**.This will enable the configuration of these characteristics in the resulting Infrastructure as Code (IaC) schema, providing scalability advantages to the cloud infrastructure.

Software Element	min_replicas	max_replicas	autoscaling
microservice	Yes	Yes	Yes
api gateway	Yes	Yes	Yes
web application	Yes	Yes	Yes

Table 5-1: Software elements scalability properties.

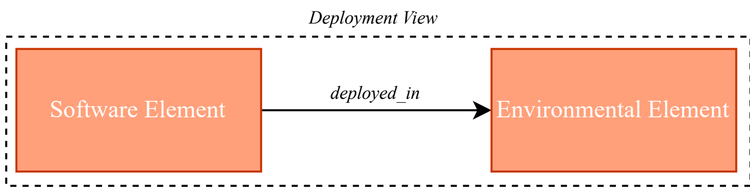

Figure 5-8: Software Element on Sarch

```
EnvironmentalElement :
  'environmental_element' '('
    environmentalElementType = [EnvironmentalElementType :ID] ')' name = ID
    '('
      ('operating_system' operatingSystem = [OperatingSystem :ID])?
      ('cpu' cpu = INT )?
      ('ram' (ram = INT StorageUnit = StorageUnit ))?
      ('storage_capacity' (storage = INT StorageUnit = StorageUnit ))?
      ('autoscaling' clusterAutscaler = ID)? # Autoscaling parameter
      ('min_replicas :' minReplicas = INT)? # minimum replica number
      ('max_replicas :' maxReplicas = INT)? # maximum replica number
    ') '?
  ;
```

Table **5-2** compiles the attributes that can be assigned based on the type of environment element being modeled. This is due to the fact that not all parameters are applicable to every environment element available in a Microservices Architecture (MSA) infrastructure.

The resulting Infrastructure as Code (IaC) schema is anticipated to align seamlessly with the specific needs and requisites of the software system delineated in the Sarch model. This comprehensive IaC blueprint will encapsulate fundamental configurations essential for deploying the software system, with a primary focus on optimizing the attributes of performance and scalability. By prioritizing these key quality attributes, the IaC schema aims to provide an infrastructure foundation that not only meets the functional requirements of the software but also ensures robust performance and the ability to seamlessly scale in response to varying workloads. The detailed specifications within the IaC will contribute to a streamlined and efficient deployment

Environmental Elements	Min Replicas	Max Replicas	Autoscaling	Storage Capacity	Cpu	Ram	Operating System
cluster	Yes	Yes	Yes	No	No	No	No
storage_service	No	No	No	Yes	No	No	No
cache_service	No	No	No	Yes	No	No	No
container	No	No	No	No	Yes	Yes	No
virtual_machine	No	No	No	Yes	Yes	Yes	Yes

Table 5-2: Environmental elements properties.

process, fostering an environment conducive to the system's optimal functionality and adaptability.

6 Evaluation

In this chapter, we present the results obtained during the validation process of the model's functionality, including the execution of model-to-model transformations and the developed tool, as introduced in earlier chapters. The validation is conducted through the analysis of a specific case study, which is modeled and deployed on two cloud service providers.

6.1 Case Study

The case study developed for the evaluation phase of this research represents a software system with a microservices architecture simulating a video streaming environment. This is because it is a robust software system, composed of various microservices, an API gateway, load balancers, and different types of storage—a perfect case study to cover a wide variety of software elements inherent to a microservices-based architecture. This particular case study comprises five microservices, each performing specific functions, along with five load balancers to ensure an equitable distribution of requests. Additionally, it integrates two databases for efficient data storage, a storage system to manage multimedia resources, a cache system to optimize performance, an API Gateway to centrally handle requests, and finally, a web application acting as a user interface for interacting with the system as a whole.

This complex and comprehensive infrastructure allows for a holistic evaluation of the capabilities and performance of the proposed model under simulated conditions of a video streaming system. Each component plays a crucial role in the operation of the overall system, providing a faithful representation of the challenges and requirements that a similar environment might face in practice.

6.2 Sarch MSA-Architecture Model as Input

A reference implementation was developed for the case study, from which the input for the model presented in Chapter 4 was derived. The modeling of the case study was performed using the Sarch tool, employing the components and connectors views, as well as the deployment view. Figure **6-1** displays the Components and Connectors (C&C) view, while Figure **6-3** presents the deployment view.

6.2.1 Component and Connector View

During this phase, the Sarch ADL is employed to model the case study using the Components and Connectors view. Figure **6-1** illustrates the Components and Connectors view of the case study, depicting the architecture's components and their relationships through connectors. Figure **6-2** provides the Sarch grammar and detailed modeling of the Components and Connectors view for the case study.

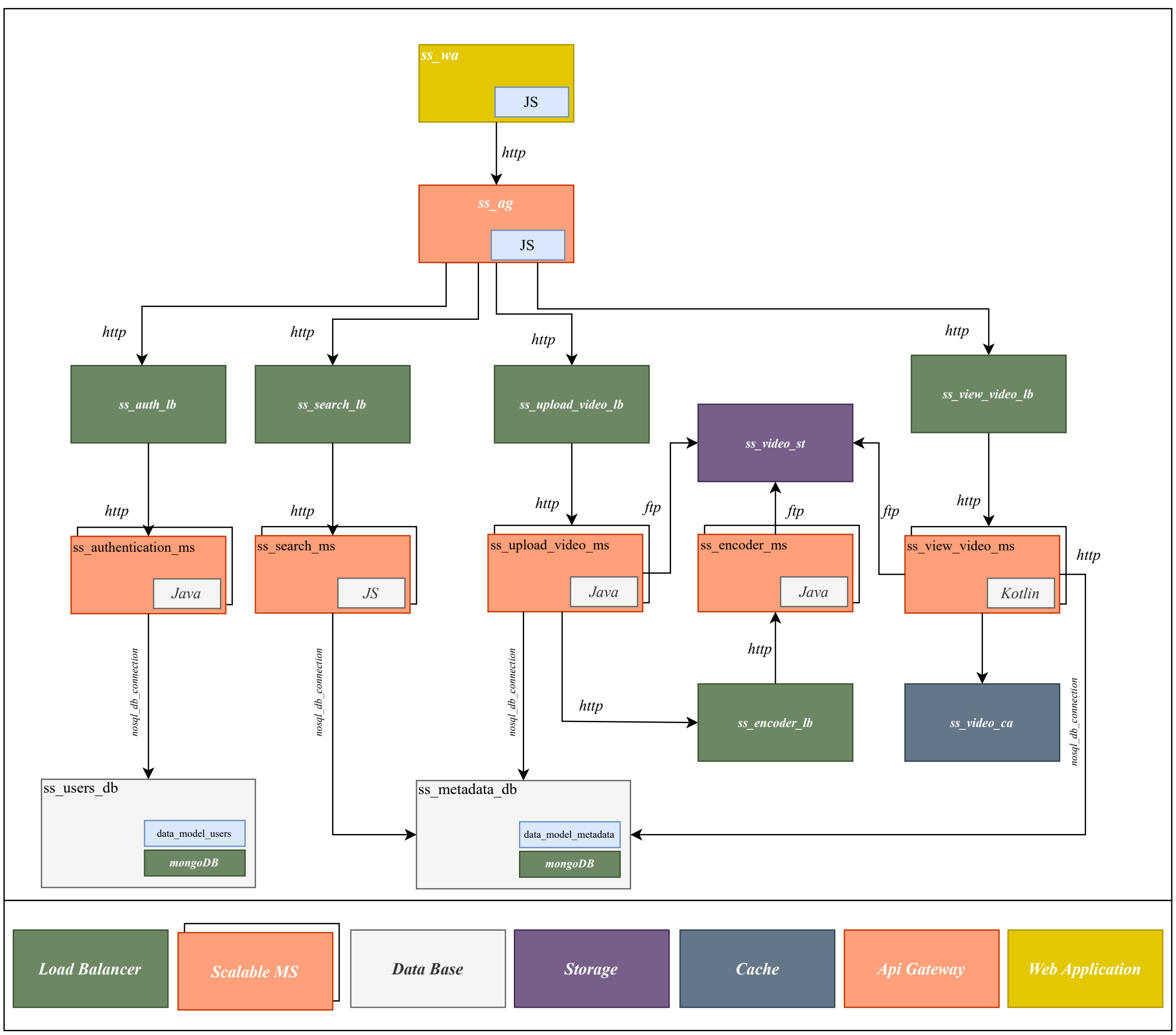

Figure 6-1: Case study C&C diagram

```
// Grammar
'component_and_connector_view' '::'
   'elements' '{'
     'types' '->'
        'components' ':' ComponentTypeList
        'connectors' ':' ConnectorTypeList
        'ports' ':' PortTypeList
        'roles' ':' RoleTypeList
        (' attributes ' '->'
            programming_languages' ':' ProgrammingLanguageList
            'db_types' ':' DbTypeList
        )?
    (componentAndConnectorElement += ComponentAndConnectorElement)+
'}'
' relations ' '{'
        (componentAndConnectorRelation += ComponentAndConnectorRelation)*
    '}'
' :: '

ComponentAndConnectorElement:
  ComponentElement | ConnectorElement;

ComponentElement:
    'component' type=ComponentType name=ID ('('
        ('programming_language' programmingLanguage=[ProgrammingLanguage:ID])?
        ('db_type' dbType=[DbType:ID])?
    ')' ) ?;

ConnectorElement:
    'connector' connectorType=[ConnectorType:ID] name=ID;
```

```
component_and_connector_view ::
    elements {
       types ->
        components: microservice, loadbalancer, apigateway, storage, cache,
                webapplication, database
        connectors: http, rest, db_connector
        ports: client, server
        roles: invokes_services, provides_services, write, reader, publisher,
                subscriber
        attributes ->
            programming_languages: java, js, kotlin
            db_types: no_sql
        component webapplication ss_wa (programming_language js)
        component apigateway ss_ag
        component loadbalancer ss_auth_lb
        component loadbalancer ss_search_lb
        component loadbalancer ss_upload_video_lb
        component loadbalancer ss_encoder_lb
        component loadbalancer ss_view_video_lb
        component microservice ss_authentication_ms (programming_language java)
        component microservice ss_search_ms (programming_language js)
        component microservice ss_upload_video_ms (programming_language java)
        component microservice ss_encoder_ms (programming_language java)
        component microservice ss_view_video_ms (programming_language kotlin)
        component database ss_users_db
        component database ss_metadata_db
        component storage ss_video_st
        component cache ss_video_ca

        connector http http_connector
        connector db_connector db_nosql
        connector db_connector ftp
    }
    relations {
        attachment http_connector: ss_wa, ss_ag
        attachment http_connector: ss_ag, ss_auth_lb
        attachment http_connector: ss_ag, ss_search_lb
        attachment http_connector: ss_ag, ss_upload_video_lb
        attachment http_connector: ss_ag, ss_view_video_lb
        attachment http_connector: ss_auth_lb, ss_authentication_ms
        attachment http_connector: ss_search_lb, ss_search_ms
        attachment http_connector: ss_upload_video_lb, ss_upload_video_ms
        attachment http_connector: ss_encoder_lb, ss_encoder_ms
        attachment http_connector: ss_view_video_lb, ss_view_video_lb

        attachment db_nosql: ss_authentication_ms, ss_users_db
        attachment db_nosql: ss_search_ms, ss_metadata_db
        attachment db_nosql: ss_upload_video_ms, ss_metadata_db
        attachment db_nosql: ss_view_video_ms, ss_metadata_db

        attachment ftp: ss_upload_video_ms, ss_video_st
        attachment ftp: ss_encoder_ms, ss_video_st
        attachment ftp: ss_view_video_ms, ss_video_st
        attachment ftp: ss_view_video_ms, ss_video_ca
    }::
```

Figure 6-2: Case Study C&C View - Sarch Grammar

This architecture model serves as a fundamental input for the execution of model-to-model transformations. Here is where the software components or elements to be deployed in the infrastructure are defined, along with their relationships and connectors.

6.2.2 Deployment View

In this phase, the Sarch ADL plays a crucial role in modeling the case study through the Deployment view. Figure **6-3** provides a graphical representation of the deployment view. Figure **6-4** showcases the specific Sarch grammar designed for the Deployment view, providing a structured framework for the representation of elements and relationships. Figures **6-4** and **6-5** present the modeling of the case study.

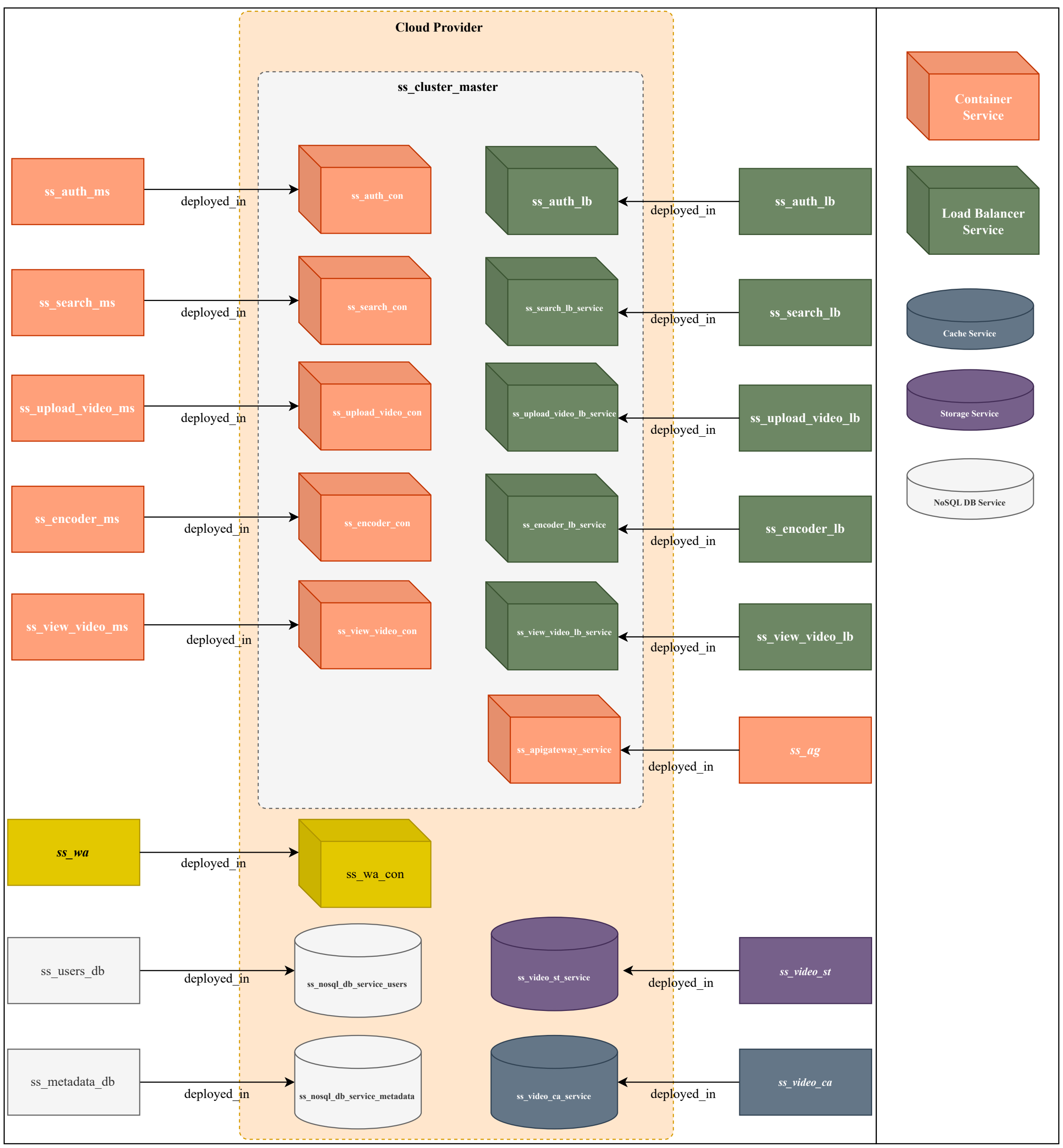

Figure 6-3: Case study Deployment diagram

```
// Sarch Grammar
'deployment_view' ':: '
    'types' '->'
        'environmental_elements' ':' EnvironmentalElementTypeList
    (' attributes ' '->'
        'operating_systems' ':' OperatingSystemList
    )?
    elements' '{'
        (deploymentElement += DeploymentElement)+
    '}'
    ' relations ' '{'
        (deploymentRelation += DeploymentRelation)+
    '}'
' :: '
DeploymentElement:
    SoftwareElement | EnvironmentalElement;
DeploymentElement:
    SoftwareElement | EnvironmentalElement ;
StorageUnit:
    name= ('gB' | 'mB' | 'kB');
SoftwareElement:
    'software_element' '(' componentElement=[ComponentElement:ID] ')' name=ID ('('
    (' min_replicas :' minReplicas=INT)?
    (' max_replicas :' maxReplicas=INT)?
    (' autoscaling ' podAutoscaler=ID)?
')' ) ?;
EnvironmentalElement:
    'environmental_element' '('
            environmentalElementType=[EnvironmentalElementType:ID] ')' name=ID ('('
        ('operating_system' operatingSystem=[OperatingSystem:ID])?
        ('cpu' cpu=INT )?
        ('ram' (ram=INT StorageUnit=StorageUnit))?
        (' storage_capacity ' (storage =INT StorageUnit=StorageUnit))?
        (' autoscaling ' clusterAutscaler =ID)?
        (' min_replicas :' minReplicas=INT)?
        (' max_replicas :' maxReplicas=INT)?
    ')' ) ?;
DeploymentRelation:
    Contains | DeployedIn;
Contains:
    'ee' ':' environmentalElement=[EnvironmentalElement:ID] 'contains' 'ee' ':'
            environmentalElementContent=[EnvironmentalElement:ID];
DeployedIn:
    'se' ':' softwareElement=[SoftwareElement:ID] 'deployed_in' 'ee' ':'
            environmentalElement=[EnvironmentalElement:ID] ('('
    ('port' name=INT)?
  ')' ) ?;
```

```
// Case Study
deployment_view ::
    types ->
        environmental_elements:
        container ,
        cache_service ,
        storage_service ,
        loadbalancer_service ,
        cluster ,
        relational_db_service ,
        no_sql_db_service ,
        node
    elements {
        software_element (ss_authentication_ms) ss_authentication_ms (
            min_replicas : 2
            max_replicas : 10
            autoscaling true
        )
        software_element (ss_encoder_ms) ss_encoder_ms (
            min_replicas : 2
            max_replicas : 10
        )
        software_element (ss_upload_video_ms) ss_upload_video_ms (
            min_replicas : 2
            max_replicas : 10
        )
        ...
        software_element (ss_auth_lb) ss_auth_lb ()
        software_element (ss_search_lb) ss_search_lb ()
        software_element (ss_encoder_lb) ss_encoder_lb ()
        ....
        environmental_element (container) ss_authentication_con (
            cpu 1
            ram 5gB
        )
        environmental_element (container) ss_search_con (
            cpu 1
            ram 5gB
        )
        ...
        environmental_element (cluster) ss_cluster_master (
            autoscaling true
        )
        environmental_element (no_sql_db_service) ss_nosql_db_service_users ()
        environmental_element (no_sql_db_service) ss_nosql_db_service_metadata ()
        environmental_element (storage_service) ss_video_st_service (
            storage_capacity 4 gB
        )
    }
```

Figure 6-4: Case Study Deployment View - Sarch Grammar

```
// Case Study
relations {
        se: ss_authentication_ms deployed_in ee: ss_authentication_con (
            port 3000
        )
        se: ss_encoder_ms deployed_in ee: ss_encoder_con (
            port 4000
        )
        se: ss_upload_video_ms deployed_in ee: ss_upload_video_con (
            port 5000
        )
        se: ss_view_video_ms deployed_in ee: ss_view_video_con (
            port 6000
        )
        se: ss_search_ms deployed_in ee: ss_search_con (
            port 8080
        )
        se: ss_auth_lb deployed_in ee: ss_auth_lb_service ()
        se: ss_upload_video_lb deployed_in ee: ss_upload_video_lb_service ()
        se: ss_search_lb deployed_in ee: ss_search_lb_service ()
        se: ss_view_video_lb deployed_in ee: ss_view_video_lb_service ()
        se: ss_encoder_lb deployed_in ee: ss_encoder_lb_service ()
        se: ss_metadata_db deployed_in ee: ss_nosql_db_service_users ()
        se: ss_users_db deployed_in ee: ss_nosql_db_service_metadata ()
        se: ss_video_st deployed_in ee: ss_video_st_service ()
        se: ss_video_ca deployed_in ee: ss_video_ca_service ()
        ee: ss_cluster_master contains ee: ss_authentication_con
        ee: ss_cluster_master contains ee: ss_encoder_con
        ee: ss_cluster_master contains ee: ss_upload_video_con
        ee: ss_cluster_master contains ee: ss_view_video_con
        ee: ss_cluster_master contains ee: ss_search_con
        ee: ss_cluster_master contains ee: ss_auth_lb_service
        ee: ss_cluster_master contains ee: ss_upload_video_lb_service
        ee: ss_cluster_master contains ee: ss_search_lb_service
        ee: ss_cluster_master contains ee: ss_view_video_lb_service
        ee: ss_cluster_master contains ee: ss_encoder_lb_service
    }
```

Figure 6-5: Case Study Deployment View Relations - Sarch Grammar

This model is a crucial input for the execution of model-to-model transformations. Here is where the components or environment elements representing modules or components within the Infrastructure as Code (IaC) schema are defined. These, in turn, represent parts of the cloud infrastructure. The relationships of each environment element help determine the type of infrastructure element to be generated.

6.3 Model-to-model Transformations

In this step, the model-to-model transformations defined in Chapter 5 are executed, transitioning from an MSA model to a new model represented in a Terraform Infrastructure as Code (IaC) schema. This section will present some sections of the IaC schema generated from the execution of the model-to-model transformations. Figure **6-6** displays the definition of a node group in Terraform, specifying the minimum and maximum replica numbers, values defined in the model created in Sarch and the result of the model-to-model transformation for the environmental element **ss_auth_con** is presented, deploying the software element **ss_auth_ms**.

Figure **6-7** illustrates the result of the model-to-model transformation for the environmental element **ss_ag_service**, which is contained in the cluster **ss_cluster_master** and enables the deployment of **ss_ag**. Additionally, the relationships modeled in the Components and Connectors (C&C) view are represented, as this component serves as an API Gateway communicating with the system's microservices.

Input Model

```
component_and_connector_view ::
    elements {
        component loadbalancer ss_auth_lb
    }
;

deployment_view ::
    elements {
        software_element (ss_authentication_ms) ss_authentication_ms (
            min_replicas: 2
            max_replicas: 10
            autoscaling true
        )
        environmental_element (container) ss_authentication_con (
            cpu 1
            ram 5gB
        )
    }
     relations {
        se: ss_authentication_ms deployed_in ee: ss_authentication_con (
            port 3000
        )
        ee: ss_cluster_sarch contains ee: ss_authentication_con
    }
```

Output Model

```
//------------- K8s node_group module

resource "aws_eks_node_group" "view_auth_ng" {
        cluster_name     = var.cluster_name // cluster name
        node_group_name = "ss_node_auth
        node_role_arn    = aws_iam_role.node_group_role.arn
        subnet_ids       = var.vpc_subnet_ids //Get VPC subnets

        instance_types = ["t2.micro"]

         scaling_config {
           desired_size = 1
          max_size     = 2 --> max replica number
          min_size     = 1 --> min replica number
          }
     }
//------------- K8s Deployment Module
resource "kubernetes_deployment" "ss_auth_dp" {
metadata {
 name      = "ss-auth-dp"
 namespace = kubernetes_namespace.cluster_namespace.metadata.0.name
}
spec {
     replicas = 1  -----> min replica number
     selector {
         match_labels = {
          app = "ss-auth-ms"
         }
     }
     template {
         metadata {
              labels = {
                  app = "ss-auth-ms"
              }
         }
     spec {
        affinity {
          node_affinity {
            preferred_during_scheduling_ignored_during_execution
            {
              weight = 50
              preference {
              match_expressions {
                  key      = "name"
                  operator = "In"
                  values   = [var.node_pool_name_auth]
      }}}}}
    container {
       name = "container"
       image = "nginx: latest "
       port {
            container_port = 80
      }}}}}}
```

Figure 6-6: Transformation result - Deployment on Cluster

Input Model	Output Model

component_and_connector_view ::
  elements {
    component apigateway ss_ag
  }
 	
  relations {
    attachment http_connector: ss_ag, ss_auth_lb
    attachment http_connector: ss_ag, ss_search_lb
    attachment http_connector: ss_ag, ss_upload_video_lb
    attachment http_connector: ss_ag, ss_view_video_lb
  }
;
 	
deployment_view ::
  types ->
    apigateway_service
 	
  elements {
    software_element (ss_ag) ss_ag ()
    environmental_element (apigateway_service) ss_ag_service ()
  }
 	
  relations {
    se: ss_ag deployed_in ee: ss_ag_service ()
    ee: ss_cluster_sarch contains ee: ss_ag_service ()
  }
	//------------ K8s Ingress module
	resource "kubernetes_ingress" "ss_ag" {
	 metadata {
	  name = "ss-ag"
	  annotations = {
	   "kubernetes.io/ingress.class" = "nginx"
	  }
	 }
	 spec {
	  rule {
	   http {
	    path {
	     path = "/auth"
	     backend {
	      service_name = "ss-auth-lb"
	      service_port = 80
	     }
	    }
	   }
	  }
	  rule {
	   http {
	    path {
	     path = "/search"
	     backend {
	      service_name = "ss-search-lb"
	      service_port = 80
	     }
	    }
	   }
	  }
	  rule {
	   http {
	    path {
	     path = "/upload"
	     backend {
	      service_name = "ss-upload-lb"
	      service_port = 80
	     }
	    }
	   }
	  }
	  rule {
	   http {
	    path {
	     path = "/view"
	     backend {
	      service_name = "ss-view-video-lb"
	      service_port = 80
	     }
	    }
	   }
	  }
	 }
	}

Figure 6-7: Transformation result - Ingress on Cluster

Whenever at least one service related to a cluster is added, the configuration of a Kubernetes module will be performed. This module supports services that need to operate within the cluster. The cluster must be described in the deployment view as shown in Figure **6-8**.

Input Model

```
deployment_view ::
    types ->
        cluster
    elements {
        environmental_element (cluster ) ss_cluster_sarch ()
    }
    relations {
        ee: ss_cluster_sarch contains ee: ss_ag_service ()
    }
```

Output Model

```
// For AWS
module "eks" {
    source = "terraform-aws-modules/eks/aws"
    version = "~> 18.0"
    cluster_name = var.cluster_name
    cluster_endpoint_public_access = var. cluster_endpoint_public_access
    cluster_endpoint_private_access = var. cluster_endpoint_private_access
    vpc_id = var.vpc_id
    subnet_ids = var.subnet_ids
    node_security_group_additional_rules = {
        ingress_allow_access_from_control_plane = {
        }
    }
}
// For GCP
resource " google_container_cluster " " kubernetes_cluster " {
  name     = var.cluster_name
  location = var.region
}
resource "google_container_node_pool" "nodes" {
  name       = var.node_pool_name
  location   = var.region
  cluster    = var.cluster_name
  node_count = 1
  node_config {
    preemptible  = true
    machine_type = "e2-medium"
  }
}
```

Figure 6-8: Transformation result - Cluster

Configuring the Horizontal Pod Autoscaler property will result in an IaC schema fragment as shown in Figure **6-9**.

Input Model

```
deployment_view ::
    types ->
        environmental_elements:
        container
        elements {
            software_element (ss_authentication_ms) ss_authentication_ms (
                min_replicas : 2
                max_replicas: 10
                autoscaling true
            )
            environmental_element (container) ss_authentication_con (
                cpu 1
                ram 5gB
            )
        }
        relations {
            se: ss_authentication_ms deployed_in ee: ss_authentication_con (
                port 3000
            )
        }
```

Output Model

```
resource "kubernetes_horizontal_pod_autoscaler_v1" "example_podautoscaler" {
  metadata {
    name = "php-apache"
    labels = {
      app = "php-apache"
    }
  }
  spec {
    max_replicas = 10
    min_replicas = 1
    scale_target_ref {
      kind = "Deployment"
      name = "php-apache"
      api_version = "app/v1"
    }
    target_cpu_utilization_percentage = 20
  }
}
```

Figure 6-9: Transformation result - HPA

After generating the entire IaC schema, you have a set of files and folders that enable the deployment of the selected cloud provider's infrastructure. In the case study, the resulting file tree is presented in Figure **6-10**.

```
/
├── modules/
│   ├── ag/
│   │   ├── main.tf
│   │   ├── provider.tf
│   │   └── variable.tf
│   ├── ca/
│   │   ├── main.tf
│   │   ├── provider.tf
│   │   └── variable.tf
│   ├── db/
│   │   ├── main.tf
│   │   ├── provider.tf
│   │   └── variable.tf
│   ├── kubernetes/
│   │   ├── main.tf
│   │   ├── provider.tf
│   │   ├── variable.tf
│   │   ├── deployment.tf
│   │   └── output.tf
│   ├── lb/
│   │   └── ...
│   ├── st/
│   │   └── ...
│   └── vpc/
│       └── ...
├── main.tf
├── variable.tf
└── CHANGELOG.md
```

Figure 6-10: IaC Schema Folders Tree

This case study has validated the effectiveness of the model and the execution of each defined transformation rule. It has successfully covered a wide range of scenarios that may arise in a microservices-based architecture (MSA).

6.4 Threats to Validity

It is essential to consider that the primary objective of this work is to develop a model that allows defining elements of cloud computing infrastructure, which are related to the architecture of a microservices-based software system and ensure platform scalability and independence. To validate the results of the implemented model, a case study was conducted representing a software system with a microservices-based architecture, incorporating the main characteristics of such architecture. However, the validity of the results and their generalization are affected by various threats, both internal and external.

6.4.1 Internal Validity

The accuracy of the results may be compromised if the input software system uses unanticipated environmental elements, such as queuing systems or specialized cloud provider services, since the model focuses on transforming systems with microservices architectures into an infrastructure-as-code scheme. To mitigate this threat, it is crucial to develop new versions of transformation rules to address a wider range of microservices-based software system cases that require the use of specialized cloud provider services.

Focusing exclusively on resource scaling tactics without considering other scalability strategies, such as code optimization or equitable workload distribution, limits the model's ability to comprehensively address the software system's needs, posing a potential threat to the internal validity of the proposed model.

6.4.2 Construct Validity

Dependency on the Sarch Architecture Description Language (ADL) could pose a threat to the model, as significant or conceptual modifications to the Sarch ADL grammar would affect the results and require adjustments to the proposed transformation rules.

Using Terraform to model infrastructure as code implies that any significant changes in Terraform licenses, versions, or configurations would require adjustments to the proposed transformation rules, representing another threat to the construct validity of the model.

6.4.3 External Validity

The exclusive focus on microservices-based architectures limits the model's applicability to other architectural styles, posing a threat to external validity. Additionally, focusing solely on performance and scalability makes it difficult to generalize the model to software systems requiring additional quality considerations, reducing its relevance and applicability in a broader context.

7 Conclusions and Future Work

7.1 Conclusions

An innovative strategy has been devised for defining cloud computing infrastructure, specifically tailored for microservices architectures (MSA). This approach facilitates the definition of Infrastructure as Code (IaC) schemes that precisely align with the key scalability quality attribute, while also enabling the definition of IaC schemes for multiple cloud providers. It is worth noting that this solution is provider-agnostic, meaning it can be used with different cloud service providers. In this initial version, the available providers are AWS and GCP. Throughout this process, the architecture description language, called Sarch, has been utilized for modeling software system architectures. An extension of this language has been implemented, enabling the definition of scalability parameters in both software components and environmental elements.

Additionally, a case study has been outlined to comprehensively validate the proper functioning of the model and established transformations. This case study has been successfully applied to at least two major cloud service providers, highlighting the effectiveness and practical applicability of the proposed strategy. The results obtained have demonstrated the adaptability and effectiveness of the strategy in real-world environments, solidifying its viability as a valuable approach for defining cloud infrastructure focused on microservices architectures.

7.2 Future Work

As a perspective for future research, an aditional expansion of this model to other architectural styles is proposed, enriching and significantly extending the scope and applicability of this work. Exploring the adaptability of the proposed strategy to diverse architectural contexts will help understand its versatility and establish robust foundations for implementation in a variety of systems. The inclusion of additional quality attributes, including security, is seen as a fundamental step towards fortifying cloud infrastructures. Incorporating security measures and controls within the Infrastructure as Code (IaC) framework will add an extra layer of robustness to cloud environments, providing a more comprehensive tool ready to address the requirements and challenges of modern software systems.

Ultimately, the integration of machine learning capabilities is proposed to enhance decision-making in defining cloud computing infrastructure. This approach would dynamically adapt the infrastructure to changing system demands, optimizing efficiency and real-time responsiveness. Exploring this integration could open new frontiers in intelligent automation of cloud infrastructure management, further improving the effectiveness and adaptability of systems. Collectively, these projected extensions will contribute to the continuous evolution and refinement of cloud infrastructure definition strategies in the landscape of modern computing systems.

Bibliography

[1] Association, IEEE S. u. a.: *1471-2000-IEEE Recommended Practice for Architectural Description for Software-Intensive Systems*. 2000

[2] Bass, Len ; Clements, Paul ; Kazman, Rick: *Software architecture in practice*. Addison-Wesley Professional, 2022

[3] Berrio-Charry, Eduardo ; Vergara-Vargas, Jeisson ; Umaña-Acosta, Henry: A component-based evolution model for service-based software architectures. In: *2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)* IEEE, 2020, S. 111–115

[4] Bhattacharjee, Anirban ; Barve, Yogesh ; Gokhale, Aniruddha ; Kuroda, Takayuki: A model-driven approach to automate the deployment and management of cloud services. In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)* IEEE, 2018, S. 109–114

[5] Brambilla, Marco ; Cabot, Jordi ; Wimmer, Manuel: *Model-driven software engineering in practice*. Morgan & Claypool Publishers, 2017

[6] Burns, Brendan ; Beda, Joe ; Hightower, Kelsey ; Evenson, Lachlan: *Kubernetes: up and running*. " O'Reilly Media, Inc.", 2022

[7] Cervantes, Humberto ; Kazman, Rick: *Designing software architectures: a practical approach*. Addison-Wesley Professional, 2016

[8] Chen, Yiran ; Luo, Tao ; Liu, Shijin ; Zhang, Shuai ; He, Liang ; Wang, Jia ; Temam, Olivier: Dadiannao: A machine-learning supercomputer. (2014), S. 609–622

[9] Clements, Paul ; Garlan, David ; Little, Reed ; Nord, Robert ; Stafford, Judith: Documenting software architectures: views and beyond. In: *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 2003, S. 740–741

[10] De Santis, Sandro ; Florez, Luis ; Nguyen, Duy V. ; Rosa, Eduardo u. a.: *Evolve the Monolith to Microservices with Java and Node*. IBM Redbooks, 2016

[11] Donham, Jacob: A domain-specific language for microservices. In: *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*, 2018, S. 2–12

[12] Feiler, Peter H. ; Gluch, David P. ; Hudak, John J.: The architecture analysis & design language (AADL): An introduction / Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst. 2006. – Forschungsbericht

[13] Garlan, David ; Monroe, Robert ; Wile, David: Acme: Architectural description of component-based systems. (2000)

[14] Howard, Michael: Terraform–Automating Infrastructure as a Service. In: *arXiv preprint arXiv:2205.10676* (2022)

[15] Jeong, Seung-Woo ; Huh, Eui-Nam: A Faster Multi-Cloud Provisioning Framework for Microservice Users. In: *2024 IEEE International Conference on Consumer Electronics (ICCE)* IEEE, 2024, S. 1–4

[16] Jouppi, Norman P. ; Young, Cliff ; Patil, Nishant ; Patterson, David ; Agrawal, Gaurav ; Bajwa, Raminder ; ... ; Yoon, Dong H.: In-datacenter performance analysis of a tensor processing unit. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, S. 1–12

[17] Kamal, Muhammad A. ; Raza, Hafiz W. ; Alam, Muhammad M. ; Mohd, M: Highlight the features of AWS, GCP and Microsoft Azure that have an impact when choosing a cloud service provider. In: *Int. J. Recent Technol. Eng* 8 (2020), Nr. 5, S. 4124–4232

[18] Kitchenham, Barbara ; Brereton, O P. ; Budgen, David ; Turner, Mark ; Bailey, John ; Linkman, Stephen: Systematic literature reviews in software engineering–a systematic literature review. In: *Information and software technology* 51 (2009), Nr. 1, S. 7–15

[19] Leite, Jair ; Oquendo, Flávio ; Batista, Thais: SysADL: a SysML profile for software architecture description. In: *Software Architecture: 7th European Conference, ECSA 2013, Montpellier, France, July 1-5, 2013. Proceedings 7* Springer, 2013, S. 106–113

[20] Li, Shanshan ; Zhang, He ; Jia, Zijia ; Zhong, Chenxing ; Zhang, Cheng ; Shan, Zhihao ; Shen, Jinfeng ; Babar, Muhammad A.: Understanding and addressing quality attributes of microservices architecture: A Systematic literature review. In: *Information and software technology* 131 (2021), S. 106449

[21] Medvidovic, Nenad ; Taylor, Richard N.: Software architecture: foundations, theory, and practice. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, 2010, S. 471–472

[22] Medvidovic, Nenad ; Taylor, Richard N.: Software architecture: foundations, theory, and practice. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, 2010

[23] NADAREISHVILI, Irakli ; MITRA, Ronnie ; MCLARTY, Matt ; AMUNDSEN, Mike: *Microservice architecture: aligning principles, practices, and culture.* " O'Reilly Media, Inc.", 2016

[24] NEWMAN, Sam: *Building microservices.* " O'Reilly Media, Inc.", 2021

[25] PAPAPOSTOLU, Tasos: $\mu\sigma$ADL: An Architecture Description Language for MicroServices. In: *Human Interaction and Emerging Technologies: Proceedings of the 1st International Conference on Human Interaction and Emerging Technologies (IHIET 2019), August 22-24, 2019, Nice, France* Springer, 2020, S. 885–889

[26] QIAN, Ling ; LUO, Zhiguo ; DU, Yujian ; GUO, Leitao: Cloud computing: An overview. In: *Cloud Computing: First International Conference, CloudCom 2009, Beijing, China, December 1-4, 2009. Proceedings 1* Springer, 2009, S. 626–631

[27] RICHARDSON, C ; FLOYD, F: *Microservices From Design to Deployment, volume 2016. NGINX.* 2016

[28] RICHARDSON, Chris: *Microservices patterns: with examples in Java.* Simon and Schuster, 2018

[29] ROZANSKI, Nick ; WOODS, Eoin: *Software systems architecture: working with stakeholders using viewpoints and perspectives.* Addison-Wesley, 2012

[30] SANDOBALIN, Julio ; INSFRAN, Emilio ; ABRAHAO, Silvia: An infrastructure modelling tool for cloud provisioning. In: *2017 IEEE international conference on services computing (SCC)* IEEE, 2017, S. 354–361

[31] SANDOBALIN, Julio ; INSFRAN, Emilio ; ABRAHAO, Silvia: ARGON: a tool for modeling cloud resources. In: *Service-Oriented Computing–ICSOC 2017 Workshops: ASOCA, ISyCC, WESOACS, and Satellite Events, Málaga, Spain, November 13–16, 2017, Revised Selected Papers* Springer, 2018, S. 393–397

[32] SANDOBALIN, Julio ; INSFRÁN, Emilio ; ABRAHÃO, Silvia: Towards Model-Driven Infrastructure Provisioning for Multiple Clouds. In: *Advances in Information Systems Development: Designing Digitalization* Springer, 2019, S. 207–225

[33] TAIBI, Davide ; LENARDUZZI, Valentina ; PAHL, Claus ; JANES, Andrea: Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages. In: *Proceedings of the XP2017 Scientific Workshops*, 2017, S. 1–5

[34] VERGARA-VARGAS, Jeisson ; UMAÑA-ACOSTA, Henry: A model-driven deployment approach for scaling distributed software architectures on a cloud computing platform. In: *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)* IEEE, 2017, S. 99–103

[35] Vergara Vargas, Jeisson A.: A model-driven deployment approach for applying the performance and scalability perspective from a set of software architecture styles. In: *Departamento de Ingeniería de Sistemas e Industrial* (2017)

[36] Zeng, Rong ; Hou, Xiaofeng ; Zhang, Lu ; Li, Chao ; Zheng, Wenli ; Guo, Minyi: Performance optimization for cloud computing systems in the microservice era: state-of-the-art and research opportunities. In: *Frontiers of Computer Science* 16 (2022), Nr. 6, S. 166106