POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Extending DevOps Modeling Language to support Kubernetes

Tesi di Laurea Magistrale in
Computer Science and Engineering

Author: **Andrea Pirrotta**

Student ID: 220766
Advisor: Prof. Elisabetta Di Nitto
Co-advisors:
Academic Year: 2023-24

Abstract

In the modern software landscape, cloud computing has emerged as the go-to hosting solution for enterprise applications, providing customers with virtually unlimited access to computing, storage, and networking resources. These resources, managed by cloud providers and accessible through the Internet, are best leveraged through the DevOps methodology. DevOps integrates development and operations into a cohesive workflow composed of many distinct phases: development, provisioning, deployment, monitoring, and more. Infrastructure as Code (IaC) and DevOps tools work in tandem to ensure solution reliability, traceability, and replicability. The DevOps Modeling Language (DOML) is an all-in-one solution that enables the definition of both application components and their underlying infrastructure in a single model, mapping components to infrastructural elements. DOML is a model-based language built on the Eclipse Modeling Framework. DOML models are translated into IaC through the Infrastructural Code Generator (ICG). Microservices have become the architecture of choice for cloud applications. Using microservices, the application's features are broken down into small, containerized features. The use of small, isolated services is the main advantage of the architecture, speeding up development and testing and simplifying deployment and version management. However, this approach results in the use of a large number of containers, even for small applications, making the container's lifecycle management very complex. For this reason, Container Orchestrators, platforms that manage the state and lifecycle of multiple containers, have become fundamental for any cloud application. Kubernetes, the leading container orchestration platform, is widely adopted in the cloud due to its powerful abstractions, replication capabilities, and scaling features.

This thesis extends DOML through the DOML-E extension mechanism, introducing Kubernetes support in the form of AWS Elastic Kubernetes Service (EKS), and accordingly extends the Infrastructural Code Generator. The result is DOML-K8s, a version of DOML capable of defining Kubernetes clusters and deploying applications on them. Key contributions include the development of DOML-K8s, the validation of DOML-E, and the extension and validation of the ICG. The thesis rigorously tests DOML-K8s through real-world examples, demonstrating its ability to create and deploy complex and scalable

applications. Finally, it assesses the capabilities of DOML-K8s and discusses potential future developments for the language.

Keywords: DevOps Modelling Language (DOML), Model-driven engineering (MDE), Infrastructure as Code (IaC), Cloud, Metamodel, Code Generation, PIACERE Framework, Container Orchestration, Kubernetes

Abstract in lingua italiana

Nel panorama software moderno, il cloud computing è emerso come la soluzione di hosting preferita per le applicazioni aziendali, offrendo ai clienti un accesso virtualmente illimitato a risorse di calcolo, archiviazione e rete. Queste risorse, gestite dai provider cloud e accessibili tramite Internet, sono sfruttate al meglio tramite la metodologia DevOps. DevOps integra development e operation in un flusso di lavoro coeso composto da fasi distinte: development, provisioning, deployment, monitoring e altro. Gli strumenti Infrastructure as Code (IaC) e DevOps lavorano in tandem per garantire affidabilità, tracciabilità e replicabilità delle soluzioni. Il DevOps Modeling Language (DOML) è uno strumento di DevOps all-in-one che consente la definizione sia dei componenti dell'applicazione che della loro infrastruttura sottostante in un singolo modello, mappando i componenti agli elementi infrastrutturali. DOML è un linguaggio model-based costruito sull'Eclipse Modeling Framework. I modelli DOML vengono tradotti in IaC tramite un parser Template-Based noto come Infrastructural Code Generator (ICG).
I microservizi sono diventati l'architettura preferita per le applicazioni cloud. Utilizzando i microservizi, le funzionalità dell'applicazione vengono suddivise in piccoli servizi containerizzati. L'uso di servizi isolati rappresenta il vantaggio principale dell'architettura, velocizzando lo sviluppo e il testing, e semplifica il deployment e la gestione delle versioni. Tuttavia, questo approccio comporta l'uso di un gran numero di container, anche per piccole applicazioni, rendendo molto complessa la gestione del ciclo di vita dei contianer. Per questo motivo, i Container Orchestrator, piattaforme che gestiscono lo stato e il ciclo di vita di più containers, sono diventati fondamentali per qualsiasi applicazione cloud. Kubernetes, la principale piattaforma di orchestrazione dei container, è ampiamente adottata nel cloud grazie all'alto livello di astrazione e alle potenti capacità di replicazione e di scalabilità.
Questa tesi estende DOML tramite il meccanismo di estensione DOML-E, introducendo il supporto per Kubernetes sotto forma di AWS Elastic Kubernetes Service (EKS), e di conseguenza estende l'Infrastructural Code Generator. Il risultato è DOML-K8s, una versione di DOML in grado di definire cluster Kubernetes e deployare applicazioni su di essi. I contributi chiave includono lo sviluppo di DOML-K8s, la validazione di DOML-E e

l'estensione e la validazione dell'ICG. La tesi testa rigorosamente DOML-K8s attraverso esempi del mondo reale, dimostrando la sua capacità di creare applicazioni complesse e scalabili. Infine, valuta le capacità di DOML-K8s e discute i potenziali sviluppi futuri per il linguaggio.

Parole chiave: DevOps Modelling Language (DOML), Model-driven engineering (MDE), Infrastructure as Code (IaC), Cloud, Metamodello, Generazione di Codice, framework PI-ACERE, Orchestrazione di Containers, Kubernetes

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

Introduction **1**

1 Background **5**
1.1 Cloud Computing . . . 5
1.1.1 Historical Evolution . . . 5
1.1.2 Payment model and Types of Services . . . 7
1.1.3 Benefits and new Challenges . . . 8
1.2 DevOps practice . . . 9
1.3 Infrastructure as Code . . . 10
1.3.1 Resource provisioning . . . 11
1.3.2 Resource Configuration . . . 14
1.4 DOML - A modeling language for Infrastructure as Code . . . 17
1.4.1 Alternatives . . . 18
1.4.2 DOML Metamodel and PIACERE Framework . . . 20
1.4.3 Model evolution and extension . . . 22
1.5 Container Orchestration . . . 23
1.5.1 Containers . . . 23
1.5.2 Container Orchestration Requirements . . . 26
1.5.3 Container Orchestration Platforms . . . 27
1.6 Summary . . . 29

2 Problem Statement **31**
2.1 Problem Identification . . . 31

2.2 Analysing Kubernetes Architecture . . . 32
2.2.1 Overview . . . 33
2.2.2 Cluster, namespace and nodes . . . 33
2.2.3 Deployments, Pods and Services . . . 33
2.2.4 Kubernetes architecture . . . 34
2.2.5 Scaling deployments . . . 35
2.2.6 Custom Resource Definition . . . 35
2.2.7 Drawbacks of Kubernetes . . . 36
2.2.8 EKS . . . 37
2.2.9 EKS Alternatives . . . 37
2.3 DOML Language Structure . . . 38
2.4 Existing DOML orchestration elements . . . 39
2.5 Integration Objectives and Requirements . . . 41
2.6 Summary . . . 41

3 Implementing Kubernetes inside DOML 43
3.1 Extending the DOML and ICG: Overview . . . 43
3.2 Kubernetes components to be modeled . . . 45
3.2.1 DOML-K8s flow of development . . . 48
3.3 New DOML elements . . . 50
3.3.1 Infrastructure layer . . . 50
3.3.2 Concrete layer . . . 52
3.3.3 Common Layer . . . 52
3.3.4 Summary . . . 53
3.4 Extending the Infrastructure Code Generator . . . 54
3.4.1 ICG architecture . . . 55
3.4.2 Flow of execution . . . 56
3.4.3 EKS template . . . 57
3.4.4 Creating the Kubernetes ICG plugin . . . 58
3.4.5 Kubernetes Resources templates . . . 60
3.4.6 Putting it all together . . . 61
3.5 Summary . . . 61

4 Validation 63
4.1 Scalable Web Server . . . 65
4.1.1 Nginx Web Server . . . 66
4.1.2 Scalable Web Server in EKS . . . 66
4.1.3 DOML-K8s implementation . . . 67

4.1.4 Validation Results . . . 68
4.2 Web Scraping with Selenium Grid . . . 70
4.2.1 Selenium Grid . . . 70
4.2.2 Implementation in EKS . . . 71
4.2.3 DOML-K8s implementation . . . 73
4.2.4 Validation Results . . . 76
4.3 Conclusions . . . 78

5 Conclusion and Future developments **81**
5.1 Objectives Achieved . . . 81
5.2 Future works . . . 82
5.2.1 Ingress . . . 83
5.2.2 Fargate Clusters . . . 84
5.2.3 CoreDNS . . . 84

Bibliography **87**

A Appendix A: Scalable Web Server use case **91**

B Appendix B: Web Scraping use case **99**

List of Figures **109**

List of Tables **111**

Acknowledgements **113**

Introduction

Context

During the last decades, computation has moved from large datacenters, owned by a single organization hosting their application, to cloud datacenters. Cloud providers own very large datacenters with virtually unlimited computing, storage, and networking capabilities in the form of cloud resources. These resources are then offered to customers wanting to deploy their application on the provider's infrastructure, on a pay-per-use model. The key to the shift toward cloud computing has certainly been advancements in virtualization and containerization technology. Using virtualization, cloud providers can host multiple applications on a single server, thus maximizing utilization. Using containerization, applications become much more portable, testable, and replicable. As application developers became more and more accustomed to cloud platforms, a new architectural pattern, known as micro-service, started growing in popularity. With micro-service architectures an application is broken down into its basic functionalities, each functionality then becomes a service, a self-contained piece of code hosted in a container and reachable through API calls or RPC. The ease of development, testing, and deployment of a micro-service, is counterbalanced by the added complexity of managing a large number of containers. As a result, numerous orchestration platforms have been designed, with the objective of simplifying the management of a large number of interconnected containers. Orchestration platforms provide a means of controlling sets of containers, scaling them according to the application's needs, and reacting to failures or unexpected behaviors. Among the various cloud orchestration platforms, Kubernetes is often the preferred choice due to its ease of use and versatility, while also being very powerful. The main drawback of Kubernetes, the complex setup process, is often shouldered by the cloud providers themselves by offering a managed and easy to setup version of Kubernetes as a cloud resource.
With the shift to cloud computing, a new approach called DevOps has risen in popularity, becoming the de facto standard for cloud development. The DevOps practice consolidates in a single process the tasks of provisioning and configuring the infrastructure, deploying of the application, and monitoring and maintaining of services. Alongside the DevOps

methodology, Infrastructure as Code has also seen widespread adoption, as a means of making the development and deployment of applications faster, less error-prone, and simpler. In this setting, numerous approaches to IaC development have been proposed, with the objective of exploiting the capabilities of IaC, while requiring a gentler learning curve compared to traditional IaC languages such as Terraform and Ansible.
The DevOps Modeling Language, or DOML, is a model-based language that is part of the PIACERE framework. It has been designed as a domain-specific language to model a cloud application and its infrastructure, in a way that is agnostic to IaC languages. DOML is based on an EMF Ecore metamodel, while its syntax is Xtext-based. An Eclipse-based Integrated Development Environment supports the user in building the model and allows the use of all other PIACERE framework tools. The model is then translated, by means of the Infrastructural Code Generator, into suitable IaC languages. DOML can be extended in order to support domain-specific resources, or to introduce new concepts into the language, using the DOML-E extension mechanism.

Thesis Contribution

This thesis focuses on integrating cloud orchestration inside the DOML, by extending its element using DOML-E. In particular, the main objective is to provide even non-expert users of DOML with the ability to configure and utilize Kubernetes, using AWS Elastic Kubernetes Service. The thesis revolves around the extension of the DOML into DOML-K8s, introducing into the language all of the relevant resources needed to create an EKS instance and to deploy containerized applications on the EKS cluster.

To sum up, the thesis provides the following contributions:

- Extension of DOML with the EKS resource, allowing the user to define Kubernetes clusters in a simple way.
- Extension of DOML with the definition of Kubernetes deployments and services, which allow the user to deploy and connect containers on a Kubernetes cluster, as well as the definition of autoscaling facilities for Kubernetes resources.
- Extension of DOML with the ability to deploy existing external Kubernetes resources on an EKS cluster.
- Extension of the ICG with new templates and new parsing plugins, used to generate Terraform code for EKS, and yaml files for Kubernetes deployments.
- Validation of a Kubernetes infrastructure developed in DOML, showing both the

functionality of various applications, as well as its autoscaling capabilities.

Theses Structure

The thesis is structured as follows:

- Chapter 1 provides an extensive overview of the background for this thesis. Starting with the overview of the cloud landscape and its evolution, moving to a description of DevOps and IaC, the introduction of the DOML, and an explanation of the need for container orchestration.
- Chapter 2 describes Kubernetes and all of its components, as well as its integration into AWS called EKS. Then dives into the DOML language structure, and explores the current capabilities of the DOML regarding cloud orchestration, comparing them to Kubernetes. Finally, it proposes the integration objectives for Kubernetes into DOML-K8s.
- Chapter 3 presents the integration process of EKS in the DOML: the extension mechanisms, new DOML-K8s metamodel elements, and the modifications made to the ICG templates and plugins.
- Chapter 4 focuses on the validation process for the integration, proposing two use cases for Kubernetes. This chapter highlights the ease of use of Kubernetes through EKS in the DOML-K8s, as well as its powerful replication and autoscaling mechanism. The Chapter also proves the flexibility of DOML and DOML-K8s, through a simple and versatile mechanism of inclusion of external Kubernetes resources.
- Chapter 5 draws conclusions on the thesis work and presents some possible future developments.

1 | Background

This Chapter presents the evolution of cloud computing, the DevOps practice, and Infrastructure as Code. Then presents the the DevOps Modeling Language.
Finally, it introduces the concept of container orchestration, discussing the problem it solves and the challenges in using it correctly.

1.1. Cloud Computing

In the early 2000s, applications were hosted on servers owned and managed by the company developing and selling the applications. Hosting even smaller applications required a big upfront expenditure to design and build the infrastructure. Some organizations, such as Amazon and Salesforce, intercepted this need from smaller companies to access an infrastructure without incurring the acquisition and management costs associated with it. As a result, they started building cloud data centers, each holding a virtually infinite amount of computing, storage, and networking resources, these resources were then offered to customers on a pay-per-use model.

1.1.1. Historical Evolution

The term Cloud computing was first invented by American Computer scientist John McCarthy, who envisioned a future in which "computers were organized and provided as a public utility". In a day in which computing for large applications was carried out through supercomputers and mainframes, which were used and managed by a single organization, the idea of taking large computing resources and dividing them among multiple users and applications was seen as the next evolutionary step of the IT industry [13].

In the past, an organization wanting to build and deploy a large-scale application would have had to build its own data center. Hereby the organization would be the sole owner and maintainer of the infrastructure, having to manage, set up, and update all equipment. There was a significant barrier to entry represented by the high initial cost of designing and building the data center. The organization was responsible for deciding what type of server

to acquire, how to handle storage, how to organize the networking inside the data center as well as to and from the internet, cooling, and power delivery. The organization was buying all of these components in enough quantity to satisfy their expected technological need. This led to an initial oversizing of the infrastructure, thus part of the significant capital expenditure was wasted, whereas over time the infrastructure would become undersized, leading to a loss in potential income from the hosted application and the need for further investments.

Nowadays we can define cloud computing as a "model for deploying and accessing facilities via the internet" [16]. Cloud providers offer computing, networking, and storage resources in a flexible and accessible manner. A customer could acquire access to the resources over the internet, and at any point in time they would only pay for the resources actually used, they could then quickly and easily adjust their usage according to their application's needs. Salesforce was one of the first companies to believe in and work on cloud computing in the late 1990s, providing their Customer Relationship Management system to their users over the internet as a service [35]. Amazon was also among the first to invest in cloud computing, leading to the launch of Amazon Web Service in 2002, which then became the biggest cloud platform by market share and profit by the early 2010s [31].

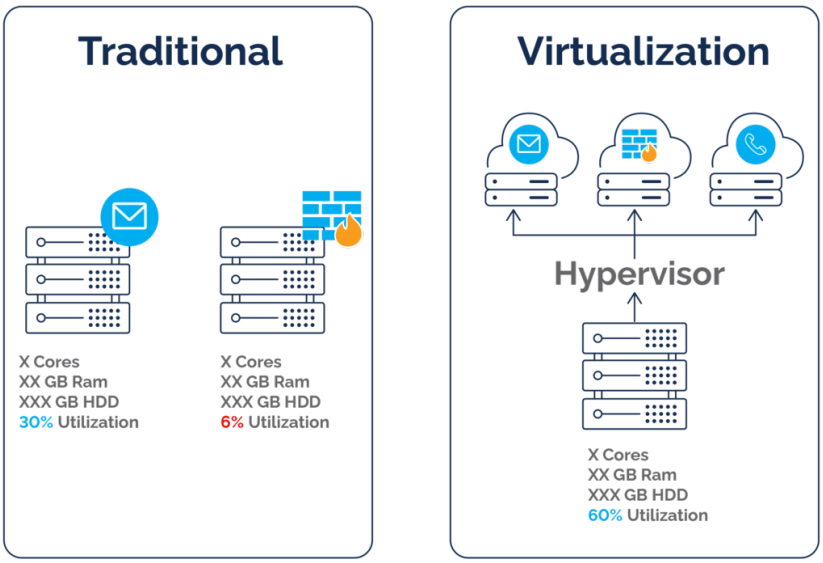

Figure 1.1: Traditional Datacenters and Cloud Datacenters, image from [38]

The biggest technological advancement that allowed the modern cloud to be born is the evolution of virtualization. Using virtualization datacenters that historically hosted a few very large applications, can be divided into smaller virtual machines and private networks that serve multiple, varying-size applications, as shown in Figure 1.1. Cloud providers par-

tition large servers into multiple smaller VMs and offer them to customers under different usage modes (PaaS, IaaS, SaaS) and paying models. The access to the physical resources is abstracted from the customer and governed by the Virtual Machine Hypervisor. In this way cloud providers can maximize utilisations of machines and leverage economies of scale to offer low prices, customers only pay for the resources they actually use and are supported by features for efficiently and automatically scaling their applications.

1.1.2. Payment model and Types of Services

One of the main reasons for the widespread adoption of the cloud and its current ubiquity on the market is the flexibility and convenience of the payment model used by cloud providers. Billing follows one of many models [4]:

- Pay as you go: users are billed depending on the actual use of the resource, eg. space occupied on disk or gigabytes of data transferred.
- Fixed upfront payment: customers buy the right to use a product for a given length of time, usually spanning multiple years, and the resources comprised in the deal are agreed with the vendor and fixed for the whole period. This solution is less flexible but yields a sizable discount.
- Dynamic pricing: revolves around the concept of spot instances. Providers rent the free and unused instances of their services at an aggressively discounted price. The glaring flaw with this approach is the uncertainty about when the resource will be free and thus available to be used.

Cloud resources can be grouped into three macro categories: computing resources, offering computational capability ranging from containers on small virtual machines to entire racks of servers; networking resources, used to securely interconnect elements in the cloud and outside the cloud; storage resources, offering various types and medium of storage options, ranging from databases to object storage to block and file storage. Resources are more often grouped by the type of service offered. That is how much control the customer has on the resource, and how much of the resource configuration is fixed and managed by the cloud provider [35]:

- Infrastructure as a Service (IAAS): refers to services offered by the cloud provider in which the customer does not control the underlying infrastructure, and is thus limited for example in the available networking components, but has control over OS, storage, and deployed applications.
- Platform as a Service (PAAS): the provider gives the user access to its platform

and freedom to deploy their application, which can then interact with the platform using proprietary libraries and frameworks. The customer does not manage the platform at OS and networking level, thus there are constraints over the possible configuration of the environment and which applications can be deployed.

- Software as a Service (SAAS): customers pay to access the software over the internet. Users do not buy the software itself but rather pay to access it, in a way determined by the software provider. The service offers little to no customization options, and the customer never accesses the underlying infrastructure.

1.1.3. Benefits and new Challenges

The main benefit obtained by moving from big on-premise systems to cloud systems is that the infrastructure is no longer an asset of the organization, that requires significant upfront capital investment and a cost over time for management, but rather is a utility that can be rented and used with a flexible cost schema and simplified configuration and management.
While in the past developing and deploying a business application required not only a complex development phase but also a thorough study of the design of the infrastructure it would be deployed on, with a subsequent large capital investment. Now companies can focus mainly on developing the application, the task of creating the infrastructure is simplified by the existence of multiple services offered by cloud providers that can be used and interconnected at will.

The cloud is thus an enabling technology to more easily develop, deploy, and scale large applications. The scaling aspect has become more prominent recently, developing an application with effective and flexible scaling means maximizing the availability of the system, and thus offering a better end-user experience, while keeping costs to a minimum. Moving to the cloud also provides other benefits such as:

- Cloud applications can be accessed from anywhere, as long as an internet connection is present.
- The computing and storage capacity is virtually infinite. Computing and storage instances can scale freely inside the cloud, with an accompanying increase in cost.
- The physical infrastructure is managed and updated by the cloud provider, the customer does not concern himself with such problems and only thinks about how to use the cloud services.

Of course, every technology comes with a trade-off. There are multiple counterarguments

to using the cloud as the single target to deploy an application.
The first obvious challenge is that of securing the cloud perimeter of the application. Since the application is deployed on servers one does not have physical access to nor control on the lowest level, cloud providers had to build a security level on top of their infrastructure. This security overlay provides the customer facilities to assign identities to users and give permissions to access and use resources. Identity and Access Management systems should be used to properly secure the resources and allow access only to trusted parties. On top of that one must also set up multiple security levels, ranging from firewalls to encryption, and attack detection and prevention systems. Further consideration must be made when it is needed to connect cloud infrastructure with an on-premise one, thus creating a hybrid cloud.
Another problem is the strong dependency an organization has with the cloud provider. Since all or most of the organization services are deployed in the cloud, if one day the provider were to change their pricing or the functionality of their products, the organization would have no say on it and would have to work hard in order to adapt their application.

This new scenario brought about new possibilities and facilities for companies to develop and deploy applications. However, the old development process was not suitable to fully utilize this new technology, thus a significant effort was needed from both academia and industry to design a new approach to application development and delivery.

1.2. DevOps practice

To deal with this new environment, in which companies had to develop their application and, alongside, provision resources from the cloud providers, a new development methodology had to be ideated and standardized. Thus a significant effort was put into evolving the current standards development practice into what is now known as DevOps.
DevOps is the process of integrating development, delivery, and operations into a single facility [11], the DevOps cycle can be seen in Figure 1.2. Inside an organization, instead of having separate teams, one developing the application and one managing the infrastructure, we now build cross-functional teams that work to continuously develop and deliver new features. This leads to a significant shift toward collaboration, but also an increase in quality and speed of delivery. Small or incremental updates to features are delivered continuously. Working on small deliveries allows us to better test the quality of the solution and its overall integration with the rest of the system. This process can then be supported by tools that verify the correctness of the process and even automate parts of

it.

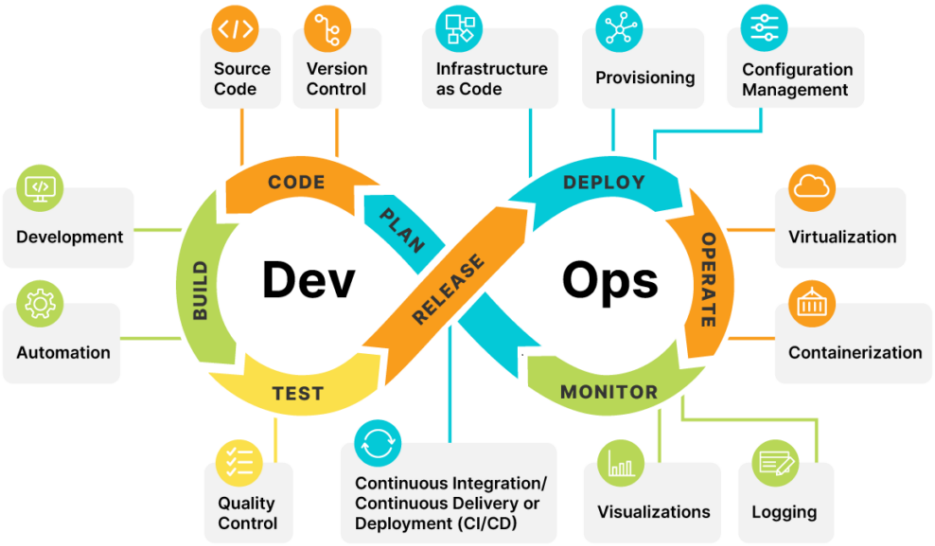

Figure 1.2: DevOps process, image from [7]

Introducing automation tools inside DevOps creates what is known as Continuous Integration / Continuous Development. CI/CD is the practice of using automatic tools to build, test, deploy, and monitor software. Using CI/CD the codebase is stored in a centralized repository, we then build pipelines that take the code, build it into a suitable artifact (executable, docker image, functions), and deploy it onto a cloud platform. Using such tools to automate deployment increases the speed at which new versions of our applications are released, and ensures consistent deployment process [32].
This Thesis covers mainly DevOps tasks related to the operations part. That is the process of provisioning cloud resources (better covered in Section 1.3.1) and deploying software on the infrastructure (explained in Section 1.3.2).

1.3. Infrastructure as Code

As CI/CD and DevOps grew more popular, managing and deploying code became easier and easier. Amid this technological advancement, the thought of handling the infrastructure definition in the same way application code is managed became a possibility.
Infrastructure as Code (IaC from now on) is a practice that allows us to treat the definition and configuration of cloud infrastructure as code [25]. By defining languages with the capability to describe the provisioning of cloud resources and to automate their configuration steps, we can employ all of the software development strategies and processes

that have been studied and tested over decades, to better manage, develop, and grow an infrastructure.
In particular, having the infrastructure in code form, allows us to perform testing on it, use CI/CD tools, and use versioning to track changes. The development process is now more robust, accountable, and suitable for collaboration. On top of that, we can use specific tools to analyze our configuration, and optimize for cost or perform security checks. All of these practices lead to a more reliable, higher quality, and more accessible infrastructure.

1.3.1. Resource provisioning

Resource provisioning refers to the process of requesting resources from a cloud provider [34]. Each provider offers two main ways to provision resources: through their web portal or using their API.

Using the web portal is the easiest and most accessible way to provision resources. Having a user interface makes it easy to configure the resource, and the providers also generally include some guidance on how to do so right on the configuration page. On the other hand, this approach is not feasible when working on a big project. The main downside here is the lack of traceability, the limited collaboration, and the inability to replicate configurations.
Since every user with access to the cloud provider's console, and the necessary permissions, can create and modify cloud resources, it becomes very difficult to track exactly what has been created or modified, when, and by who. The matter becomes an even bigger problem in a process such as DevOps, which is by nature incremental. Creating or modifying resources becomes dangerous when we have no clear view of our infrastructure. One may argue that this problem can be solved by keeping track of changes using other means (logging to a file or using project management tools), and a view of the state of the system can be obtained by exploring the project using the cloud provider's console. Nonetheless, these solutions become irrelevant when taking into consideration large projects or large teams. A closely related problem is the lack of collaboration. A lot of care must be put into managing a team deploying the infrastructure, making sure that different people are not provisioning duplicate resources, and that the resource is correctly configured. The final, and perhaps more pressing matter, is the inability to directly replicate resources. Since each resource is provisioned separately, and there is no unified definition of the infrastructure, replicating a cloud environment becomes a very long, taxing, and error-prone process. The problem greatly limits the effectiveness of this approach when working with the common setup of having separate development, test,

and production environments. Making sure each environment is exactly like the previous is not trivial, and requires very careful checks of each resource. On top of that, when replicating an environment, we incur the significant possibility of human error.

Using the provider's API, we are able to overcome some of the limitations previously introduced. Each resource is created by calling the API endpoint and providing the configuration details as call parameters. The first upside is that it is much easier to track API calls and keep a history of what has been created or modified. We can also programmatically execute such API calls, we can thus automate the provisioning process and even parametrize it. The downside is that we still have limited collaboration. On top of that provisioning resources through API calls is anything but simple.

As a result of these limitations, over the years many provisioning languages have been created, with the aim to streamline the process. Most of them provide an abstraction over the cloud provider's API and allow the user to simply define the infrastructure declaratively. An underlying engine will then translate this declaration into API calls and execute them.
A good provisioning technology must be able to satisfy the following requirements:

- Providing a central source of truth for the configuration of our infrastructure.
- Allowing collaboration and versioning, using tools like git
- Making the process of provisioning the infrastructure easier, faster, and less error-prone.
- Automating infrastructure provisioning and allowing for the use of tools to check the correctness and security of the infrastructure.
- Easily replicating the infrastructure across different environments.

. The following Section provides a brief description and comparison between the main provisioning technologies.

TOSCA (Topology and Orchestration Specification for Cloud Applications) [1] is a language used to describe the topology of a cloud application. The TOSCA specification allows for the definition of a YAML file that models application components, relationships, and workflows. The language itself only serves as a foundation that can be extended in order to describe the architecture of a cloud application. As such it is the basis for most of the technologies described in Section 1.4.1.

[1] TOSCA website: `https://www.oasis-open.org/committees/tosca/faq.php`

Provisioning technologies

This Section provides a comparison between the three most used provisioning technologies: Cloudify, AWS CloudFormation, and Terraform [6].

Cloudify [2] is a cloud orchestration framework that allows users to both model their infrastructure and automate their entire lifecycle. This tool is not limited to resource provisioning but is also able to monitor deployed applications, detect problems and failures, remediate problems, and handle maintenance. All of these features place Cloudify as a full orchestration tool but on the other hand, some of the more advanced features are available only with the premium version.

AWS CloudFormation [3] provides a common language to define and provision all infrastructure resources in an AWS project, all from within a single file that acts as the single source of truth for the infrastructure. A CloudFormation template defines a set of AWS resources to provision, each with its properties and dependencies, a set of resources is called a stack. Applying a template that references resources that already exist does not create a duplicate of the resource, but rather works only on the differences between the two stacks.

Terraform [4] is the most used provisioning language. Using Terraform users can declare the desired structure of their application infrastructure using the Terraform dialect. The Terraform engine will then check the current state of the deployed infrastructure, and provision, modify, or delete resources if needed, to reconcile the current state with the declared state.

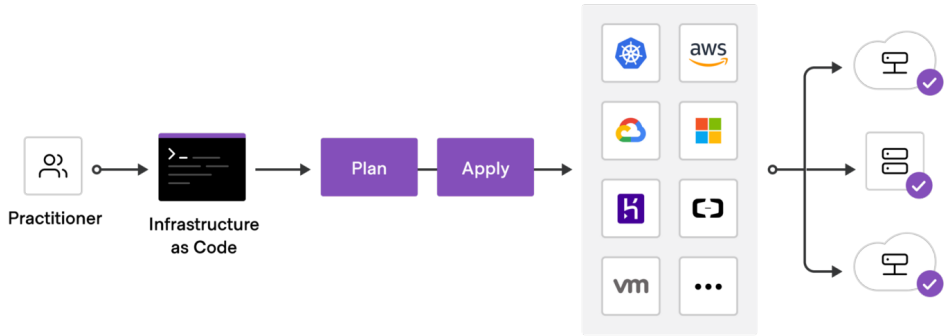

Figure 1.3: Terraform workflow, image from the Terraform website [37]

The engine is particularly good at identifying the differences between the current state

[2]Cloudify website: `https://docs.cloudify.co`
[3]CloudFormation website: `https://aws.amazon.com/it/cloudformation/`
[4]Terraform website: `https://www.terraform.io`

and the desired state of the infrastructure and creating a plan of action to perform (create a resource, delete a resource, modify it) in order to reach the desired state. This workflow can be appreciated in Figure 1.3. The user never actually tells Terraform what actions to take, but rather how he wants the infrastructure to be shaped, the Terraform engine then figures out the needed actions in terms of API calls and performs them.

The main advantage Terraform holds over its competitors is its ease of use and generality in supported providers. CloudFormation is the closest to Terraform in terms of workflow and ease of use, both define the infrastructure declaratively, but the huge drawback of CloudFormation is that it can only be used with AWS. The advantage of CloudFormation over Terraform is the superior error handling, if a resource fails to be provisioned the entire provision process is rolled back, thus not creating dangling resources. Cloudify offers more features and is the best all-in-one DevOps tool, but is let down by its pricing, and it is generally preferred to break down different DevOps tasks, like provisioning, configuration, and deployment, into separate specialized languages. Only Terraform will be considered for resource provisioning for the rest of the Thesis.

1.3.2. Resource Configuration

Resource configuration refers to configuring the state of Virtual Machines inside an infrastructure. The configuration is not intended as physical, determining the number of cores or memory size, but rather as the software running inside the machine. In the context of a cloud infrastructure VMs play a central role, thanks to their flexibility in both resource scaling and configuration possibilities. Thus the requirement for a language that allowed to quickly deploy and configure the software side of IAAS (see Section 1.1.2) resources became immediately relevant. A suitable language would need to satisfy the many requirements requested from the application's stakeholders and then passed onto DevOps teams [33]:

- Automated Application Deployment: the main focus of resource configuration, offering the possibility to deploy an application on the target machine in a quick and secure manner. The previous approach revolved around using bash scripts to check whether the required software was installed and running on the machine, as well as checking for firewall rules and open ports. Modern languages are able to query the target system, gather facts about its state, and deploy the software needed or configure the system parameters.
- System Dependencies: often time an application is not deployed on a single machine, but rather on multiple machines, each being a replica or holding a different

component, which would have to communicate with one another. This creates dependencies between the deployments, a suitable technology must be able to deploy the components in the correct order.

- Monitoring: there is also the need to deploy monitoring agents, that constantly stream the state of the system and flag unusual behavior.
- Active Provisioning: in the ever-changing and evolving cloud environment, the technology we are looking for must be able to dynamically adapt to changes in the environment in which our application is running. Like in the previous case, this is also done by gathering facts about the state of the system.

On top of that, using a language to configure IAAS machines provides us with most of the benefits previously highlighted in Section 1.3.1, such as traceability, documentation, collaboration, and versioning.
A comparison between different configuration technologies can be found in the following Section.

Configuration technologies

This Section provides an overview and comparison among some of the main resource configuration technologies: Ansible, Puppet, and Chef [42] [24].

Ansible [5] is an open-source configuration management tool that has proved very popular in the cloud development environment. This is thanks to its ability to see the infrastructure as a collection of interrelated servers, as well as being agentless, meaning there is no need to install an agent on the target machines.

[5] Ansible website: `https://www.ansible.com`

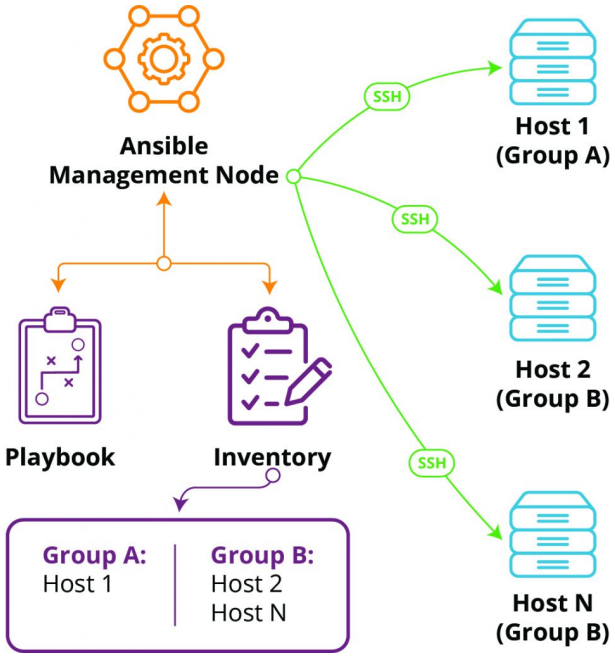

Figure 1.4: Ansible configuration process, from [39]

The dynamic view of the infrastructure is contained in the inventory file, which is the single point of truth for the identities (in terms of addresses) of each machine in the system. Ansible then provides a variety of modules to interact with these machines either imperatively or declaratively. Modules are small programs that are sent to the target machines over SSH, perform one very small and specific unit of work (like downloading software, setting environment variables, opening ports, adding a firewall rule, etc.), and are then deleted. Modules are grouped into a play, a play is a collection of tasks, where a task is just a name assigned to a module set to run with some given arguments. This allows us to create complex configurations starting from small and simple tasks. Each play is then executed on some hosts, hosts are named references to an address or a set of addresses in the inventory file. Plays are then grouped into playbooks. Each task in a play is idempotent, we are thus assured that running the same playbook multiple times will always lead to the same desired state of the system. This configuration process is shown in Figure 1.4.

Puppet [6] is an agent-based configuration management tool, but also offers an agent-less alternative called Puppet Bolt. From now on we will talk about Puppet Bolt, while referring to it as Puppet for convenience's sake. A Puppet configuration revolves around two files: one listing the target machines, and one defining the configuration steps. The inventory file contains the list of IPs or names of the target machines, much like with

[6]Puppet website: `https://puppet.com`

Ansible inventory files. The manifest contains the list of commands to execute on a subset of the target machines specified declaratively, commands can be grouped into modules just like in Ansible.

Chef [7] is an agent-based configuration tool that works by writing Ruby scripts that define a state that must be true on the target machines. These scripts are called recipes, and communicate with target machines using a command tool called Knife, recipes are collected into cookbooks. A cookbook defines a state that must be consistent across the machines. Chef fetches the state of the targets and takes action in order to reconcile it with the desired state.

Ansible and Puppet offer very similar features. Ansible has the advantage of having a large repository of modules and plays that can be quickly integrated into a playbook. Puppet has the edge in simplicity due to its declarative approach, even though Ansible can also be used declaratively. Chef falls behind the two due to its needing an agent on the target machines, and the additional layer of adoption of writing the state in ruby making it more complicated to use compared to the other two. From now on we will only consider Ansible for resource configuration.

1.4. DOML - A modeling language for Infrastructure as Code

The DevOps Modeling Language is a model-based language developed to support the DevOps practice, by allowing the programmer to define in three separate moments: the application components that make up the system, and their connections; the abstract infrastructure on which the application components must be deployed; the concretization of the infrastructure, using a specific cloud provider's services. The main advantage of using DOML is its ability to abstract and hide the specificity of IaC solutions such as Ansible and Terraform (see Section 1.3). This is achieved by separately defining the application from a software perspective, then the abstract infrastructure the software is to be deployed on, and only finally the infrastructure layer mapping to a specific software provider [3]. This level of abstraction also serves the purpose of decreasing the learning curve for DevOps technologies. This is done at first by condensing in a single language functionalities that before were divided upon multiple languages, such as Terraform and Ansible for provisioning and configuration. Abstracting away the implementation of the infrastructural components when first designing the infrastructure makes the user ex-

[7] Chef website: `https://www.chef.io`

perience even simpler. The final result is allowing users with low expertise to perform infrastructure design and deployment tasks.
The language implementation is composed of two parts: the DOML metamodel, described using Ecore from the Eclipse Modeling Framework (EMF) [8], and its syntax, based on Xtext [9].

1.4.1. Alternatives

The DOML is not the only Model-based IaC language, various similar technologies have been developed over the years. An overview of some of the most popular alternatives is presented below.

DICER

DICE (Data Infrastructure Capacity for EOSC) is a model-based DevOps platform with a strong focus on Big Data applications. DICE incorporates both development tools, such as the DICE IDE, and runtime tools, such as analysis tools or CD and testing tools. DICE allows the user to describe the components and architecture of a cloud application in a way that is platform-independent, these components are then mapped to a TOSCA-compliant deployment.

SODALITE

The SODALITE (SOftware Defined AppLication Infrastructures managemenT and Execution) framework is a DevOps technology that focuses on quick and easy deployment of IaC. SODALITE assists non-expert DevOps users in the definition of an infrastructure deployment. The goal of SODALITE is to generate an application deployment that is portable and platform-independent (using the TOSCA standard) while reducing coding efforts and supporting the user with various development tools.
SODALITE supports the modeling of the application by a multidisciplinary team composed of various roles. The *Application Ops Experts* work on modeling the application topology by generating an *Abstract Application Deployment Model*. An AADM is a connected graph of application components that describes relationships and interactions among them, as well as requirements on the infrastructure resources needed. The *Quality Experts* tackle the optimization of a deployment by creating an *Optimisation Model*, such model is then bound to a component in the AADM. An OM describes section-specific

[8] EMF website: `https://eclipse.dev/modeling/emf/`
[9] Xtext website: `https://eclipse.dev/Xtext/`

options that are used by the SODALITE optimizer. The *Resource Experts* model the infrastructure resources using *Resource Models*. The RM describes the resource types, their relationships, and policies, as a specialization of TOSCA elements, and can provide *Ansible Models* (which are then translated to Ansible playbooks) that are to be used when creating the specific resource.
The SODALITE platform is divided into three layers: the *Modeling layer* includes the so-called domain ontologies, the knowledge base containing the AADM, OM, and RM that are used to model the application architecture; the *Infrastructure as Code layer*, which offers APIs to optimize, validate and verify the RM and AADM; the *Runtime layer* which is in charge of the deployment of the application, as well as its monitoring and dynamic reconfiguration [8]. The three layers are put together by the IDE, which offers different tools to model the application, verify and optimize the generated models, and deploy the described application.

EDMM

The Essential Deployment Metamodel is a model-based deployment tool that allows the creation of a technology-agnostic deployment model. The EDMM follows the route of declarative deployment models, in which users describe the structure of the application to be deployed including all components, their configuration, and relationships. In particular, EDMM allows the definition of [41]:

- Components, a physical or logical unit of an application, each component has a type.
- Relationships between components, a physical, functional, or logical dependency between two components. Relationships are typed and can be used to specify things such as a connection between an application component and a queuing system, or the fact that an application component must be installed on a server.
- Operations, procedures to be performed on a component, for example, configuration steps.
- Properties, describing the current or the desired state of a component or relation. Properties can be used to model facts such as exposed ports or required software modules.
- Artifacts, the software implementation of a component or an operation. Artifacts range from shell scripts that install a given component to the binary file of an application.

- Deployment model, file that describes declaratively the desired state of an application using the previously listed EDMM components.

The EDMM Modeling and Transformation System supports the user in the creation of the deployment of an application, and subsequent transformation into the target technology used to actually deploy the application. The EDMM Modeling Tools presents the user with a web-based graphical interface to model the application topology, the components are based on TOSCA. The EDMM Transformation Framework provides a CLI that transforms the EDMM model into a file from the target deployment technology, such as Terraform or Kubernetes [40].

Radius

Radius [10] is an open-source project aiming at the creation of a streamlined platform that standardizes the deployment and management of cloud-native applications. In particular, the 3 key problems RADIUS addresses are: integration of Kubernetes into large, complex, and diverse cloud applications, this point is particularly relevant as it is the foundation for this Thesis; collaboration between developers and platform engineers, most large enterprises are creating specialized tools to standardize automation, tracking and observability of development but not all endeavors are successful, Radius hopes to become such platform; the need for an industry-standard for application development.
Radius itself provides an HTTP-based API and support tools that can be used to design, deploy, and manage cloud applications, the API can be hosted on a Kubernetes cluster or standalone. In Radius all of the elements composing the applications are called resources, the API allows the creation, read, modification, deletion, and list of all resources, as well as resource-specific operations such as rebooting a VM or restoring its state from a backup. The description of the desired resource is contained in the Bicep file, the file is sent to the API server which parses it and performs the necessary changes to the state of the application.

1.4.2. DOML Metamodel and PIACERE Framework

The language follows the practice of Model Driven Development [12], each system definition is a DOML model, which is an instance of the DOML metamodel. The models are then turned into deployable IaCs by the Infrastructural Code Generator. The ICG can produce IaC in a variety of target languages. This workflow is made possible by the PI-

[10] `https://radapp.io/`

ACERE framework [11], a suite of components supporting the user in designing, analyzing, and translating a DOML model into the deployable IaC.

Defining a metamodel using EMF allows for the creation of an eclipse-like IDE, which offers syntax highlighting, syntax suggestion, and auto-completion, as well as allowing the user to invoke other tools useful in supporting a DOML developer. A partial view of the PIACERE framework relevant to this Thesis can be seen in Figure 1.5. The components shown are part of the Design Time part of the framework, containing tools aimed at helping the developer in describing the architecture through the DOML syntax, and translating the code into IaC.

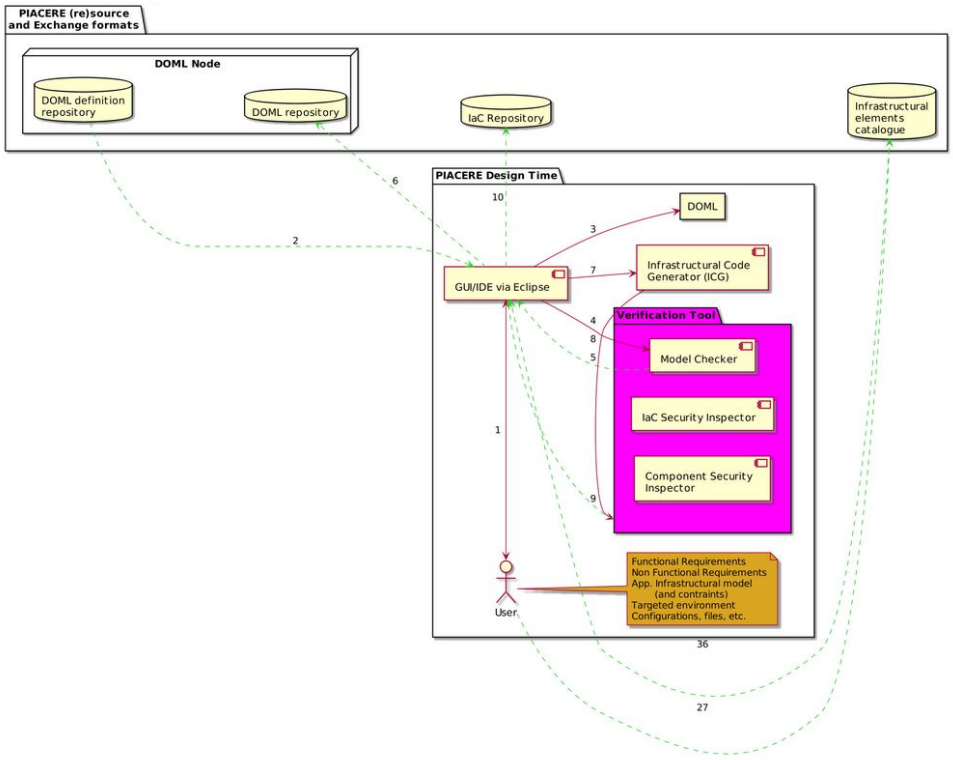

Figure 1.5: PIACERE Framework Design Time components, from [29]

A component of the PIACERE suite called DOML Model Checker (DMC) can be run at any point from the IDE to verify the internal consistency of the model. The DMC is able to assert not only the syntactical correctness of the model but also to look for possible errors or bugs in the described infrastructure. The DMC checks for inconsistencies such as VMs not connected to any network, duplicated network interfaces, AIL components not mapped to any concrete element, or software components sharing interfaces deployed on nodes that cannot communicate with one another. It is also possible for the user to

[11] PIACERE website: `https://piacere-project.eu`

describe additional requirements the DMC should check for, making the analysis phase customizable to the specific use case.
Another integral component of the PIACERE framework is the Infrastructural Code Generator. The ICG receives as input a DOML model in XML format and generates the IaC in the specified target languages. Using Template-Based Code Generation the ICG substitutes values taken from the model into templates, meaning the same DOML elements can be translated into different target languages by changing the template used. This proves particularly useful in the cloud world since the same components from the CIL can be translated into IaC for different cloud providers by simply changing the template used. The ICG is also highly extensible, offering the possibility to add new templates for new components, but also plugins to support a new language altogether.

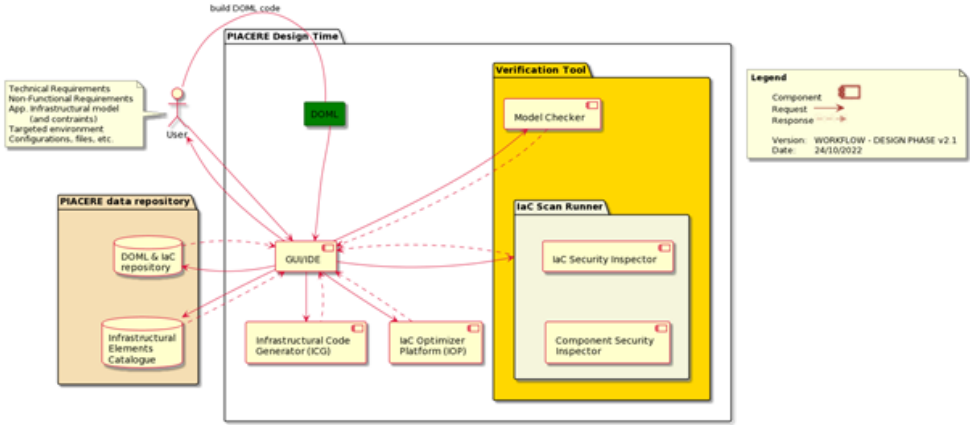

Figure 1.6: PIACERE Design Process, from [28]

The design process of an application uses all of the PIACERE framework tools. The tools can be invoked from the IDE, which serves as the entry point for the whole suite and supports the DOML developer in the creation of code. An overview of the design process is provided in Figure 1.6

1.4.3. Model evolution and extension

The DOML is by itself a highly expressive language, able to define complex infrastructures in great detail. Nonetheless, the cloud landscape is ever-changing and evolving, for this reason, the DOML also provides a powerful extension mechanism in the DOML-E [2].
DOML-E is the tool that supports the extension and evolution of the DOML, changes to the metamodel are reflected in the syntax and, together with the extensibility of the ICG, allow the DOML to keep up to date with the latest cloud advancements.

In particular, DOML-E allows the user to incorporate new concepts into the language by

creating new classes or extending existing ones. Firstly, DOML-E allows for the extension of existing elements by introducing new properties, thus refining already existing concepts for some specific scenarios. Then the DOML Source class can be used to integrate external IaC files directly into the DOML model, thus creating a layer of interoperability with external elements without altering the metamodel or syntax. Finally, a DOML class itself can be extended, allowing the user to address resources that are partially represented but need some custom properties to be properly described. If there was the need to represent resources that do not fit into concepts already existing in the DOML, the GenericResource class can be extended to add a completely new concept into the language. Once the desired changes have been made to the metamodel, the syntax is updated in order to respect the new version of the language.

1.5. Container Orchestration

The shift toward cloud computing influenced many fields of software development and application architecture is no different, as the field saw the adoption of micro-service architectures. Applications are broken down into small functionalities called services, the application flow becomes a series of API calls from one service to another. Each service is implemented through a container deployed on the cloud infrastructure, services communicate with one another using REST APIs or RPC. The upsides in ease of development, testing, update, and deployment, are countered by the increase in complexity related to the management of a large network of containers.
In this landscape, many technologies emerged, intending to efficiently manage this multitude of containers. Such platforms are called orchestrators, an orchestrator manages the lifecycle of containers, from deployment to runtime communication to scaling and monitoring. Among various container orchestration platforms, Kubernetes quickly rose to popularity. Although creating and managing a cluster is far from simple, the features provided by Kubernetes are unrivaled in terms of ease of use and versatility. Kubernetes allows users to declaratively create containers, handle networking, replicate containers for redundancy, handle load balancing, and even perform autoscaling depending on the application's needs.

1.5.1. Containers

Containers are among the most important cloud building blocks. A container is a lightweight application package, an image that can be hosted in a virtualized environment, just like traditional VM images, but consumes fewer resources and has a lower spin-up time. A

container holds packaged and self-contained software that implements part of an application. This characteristic allows containerized applications to be easy to develop, test, and deploy. The backbone of containerization technology is the container engine, which allows to host multiple containers in a common virtualized environment while ensuring high portability. Among the many container applications, two competitors stand out: Docker [12] and Singularity [13].

Docker

Docker is a software that allows users to run their containerized applications in the docker environment. A container is a bundle of user code and kernel libraries that lives on top of the docker engine.

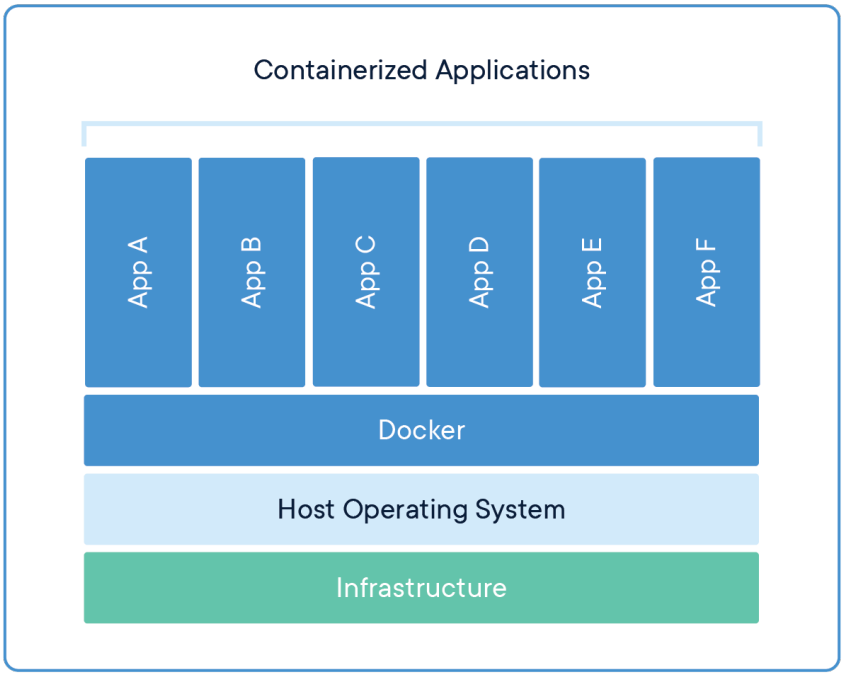

Figure 1.7: Docker stack, from the Docker website [9]

The engine itself sits on top of the physical hardware or, in the case of data centers, on top of the VM hypervisor. The container is then hosted on top of the engine (see Figure 1.7), which is responsible for translating system calls to something understandable by the underlying OS. The container becomes thus totally portable and can be hosted on any machine that supports running the docker engine. The portability property of the docker container ensures that the behavior of the system will be the same on a local

[12] Docker website: `https://www.docker.com`
[13] Singularity website: `https://docs.sylabs.io/guides/3.5/user-guide/introduction.html`

development machine, as well as on the deployed cloud environment [27]. The container is made up of the user application code, and the specific libraries needed to run such code, the rest of the kernel lives inside the engine and is shared with other containers, this allows the image to be very light and have minimal start-up time [18].

Singularity

Singularity is a container solution created to answer the necessities of scientific applications. The main goal of Singularity is to make computing mobile, creating fully portable environments that can be seamlessly integrated with scientific computational resources. With Singularity, the container image encapsulates the entire software stack, alongside any data files and libraries needed. This image is highly portable, the reproducibility is highlighted by the ability to snapshot the state of the image at any given time and lock it down for future use. Another important feature that makes Singularity stand out in the field of scientific computing, is the ability to generate a unique hash from the image. The image and its hash can then be distributed alongside a scientific paper, allowing for independent validation of the study's results.
Singularity was built from the ground up to work in High Performance Computing (HPC) environments, which are fundamentally different from consumer or cloud hardware. HPC focuses on purpose-built, highly parallel computation devices, supported by very fast networking with low latency [17]. Running container technologies like Docker on existing traditional HPC resources is not trivial, since they generally require kernel features that are not yet stable or even available on HPC hardware. Singularity instead is designed to work on technologies supported by the older Linux distributions used in HPC and, as a result, Singularity can be run on an HPC cluster just like any other command on the system [21].

Comparison

The first big difference between Docker and Singularity has already been introduced, that Docker cannot be run on most existing HPC supercomputers, simply because the fundamental features needed at the OS level to run the Docker engine are not present in the Linux distributions used for HPC. Another limitation of Docker which promotes the use of Singularity in HPC environments is the lack of security. Docker by itself is not an insecure software, but a problematic design choice of the Docker architecture is that of running the Docker daemon, the main Docker process that orchestrates the containers on a machine, with root privileges. While this is fine when working with a cloud VM, this is definitely not possible in HPC clusters [14]. These two motives are enough to declare

Singularity as the better choice for High Performance Computing, but when taking a step back, it is hard to find features that make Singularity better than Docker for consumer or enterprise cloud applications. Docker already has a long history in cloud environments, is better suited for modern Linux distributions (for example Alpine is a modern lightweight Linux distro often used as a base for the containers), and has a huge community of users. This translates to a vast amount of pre-built images published over the years, which can be easily pulled from the Docker registry for public use.
Throughout the rest of the Thesis, we will only consider docker for container virtualization, due to its being the most versatile and widespread for non-HPC cloud applications.

1.5.2. Container Orchestration Requirements

Modern applications are based on a micro-service architecture. The functionalities of the system are broken down into their most basic features, each feature becomes a service that is implemented in a self-contained environment and meant to be called through web calls (often REST over HTTP). These services are then bundled as a container image, where each image implements one or more services. The containers are then deployed on the cloud using the methodology and tools described in Sections 1.2 and 1.3. This methodology makes the services highly reusable since they carry out a very specific and self-contained task. By carrying out extensive unit tests (test on the correct functionality of a single feature) and integration tests (test on the correct interaction between components), we are assured that the final deployed system will behave as expected.
The paradigm of service-oriented architecture allowed to create and deliver applications faster than ever but also provided some drawbacks. Complex applications see an explosion in the number of containers deployed, to the point of making manual management almost impossible. Monitoring tasks are also aggravated by the large number of entities creating logs and alerts, designing a response plan becomes harder since the system is tightly integrated and a fault of a small feature can take down the entire system. Container orchestration takes care of correctly setting up the initial configuration of a multi-container application, as well as tracking its evolution over its lifecycle. The main requirement a Container Orchestration Platform must satisfy is the ability to manage a fault-tolerant, highly available, and potentially geographically dispersed cluster of containers.

In order to achieve high availability and fault tolerance, an orchestrator implements the 3 principles of system design reliability [19]:

- Elimination of single point of failure: done by introducing redundancy at the component level. More than one replica of each container is deployed, such that if one

fails, another one is always active and the system can continue to work.

- Reliable crossover: the crossover point, the one responsible for redirecting traffic to active components, must be reliable and replicated as well.
- Detection of failures: the orchestrator must be able to detect when a failure occurs inside the system, and react accordingly. An example is restarting a failed container and redirecting traffic to a working one.

A key pattern used to respect the three principles is that of the load balancer. The load balancer is a component placed in between the user (or the caller service) and the target service, its job is to optimize the use of resources and maximize throughput by routing the requests to the correct replica of a service.

Another important aspect of an orchestrator is its ability to provide an efficient networking platform at scale. For security purposes the network in which the containers and the orchestrator run must be separated from the rest of the system. It is up to the orchestrator to offer a functional and scalable means for the containers to communicate with one another, as well as with outside networks when needed. This network must generally be very flexible since containers are often shut down and restarted.

Finally, a container orchestrator must offer some form of scaling functionality, in order to reactively start up more containers according to the application load.

1.5.3. Container Orchestration Platforms

This Section provides an overview and comparison between two of the most used Container Orchestration technologies: Docker Swarm and Kubernetes.

Docker Swarm

Docker Swarm [14] is the container orchestration tool built info Docker, which supports the management of a cluster of Docker Containers [26]. As an extension of Docker, it allows to create a network of containers distributed across different nodes. One node is called the Manager node, it is a replicated node responsible for knowing the status of all other nodes and giving out tasks to the worker nodes when needed. Worker nodes host one or more containers and perform tasks sent out by the manager node, in doing so the manager node also acts as a load balancer. In a swarm the single unit of management is the service, a service is a group of containers with the same image, and services can scale according to the demands of the application. The swarm manager node ensures high

[14] Docker Swarm website: `https://docs.docker.com/engine/swarm/`

availability by restarting a node that goes offline. Once a service has been set up in a swarm, it can be accessed by every node thanks to the swarm network.
Swarm was designed from the beginning as a simple and user-friendly orchestration platform, easy to use for those who already have experience with the Docker ecosystem and its CLI. Docker Swarm is considered ideal for small to medium-scale deployment scenarios, where a quick and simple setup is prioritized. Creating a swarm is simple, but more advanced features like autoscaling, division of workload, and access management must be handled using external tools.

Kubernetes

Kubernetes [15] is the most used container orchestration platform. It's open source and offers a platform to easily deploy, scale, and manage containers. A Kubernetes cluster is a collection of servers that act as a single machine from the user's point of view. Resources in the cluster can be virtually divided into sub-clusters using namespaces, almost all resources are scoped to a namespace and cannot be seen from other namespaces. In the cluster, there are two main types of nodes: the master node and the worker nodes. The worker nodes host applications in the form of replicated containers, while the master node manages the containers making sure that their status remains consistent. This architecture ensures high availability, load-balancing principles are also applied when scheduling tasks for workers and inside workers. The number of replicas of a container can scale automatically according to various metrics such as application load or resource utilization. Containers communicate through services, which can connect containers with one another or with the outside of the cluster using the Kubernetes network.

Comparison

Kubernetes is more complex to set up and manage compared to Docker Swarm, but the result is its ability to offer advanced features without the use of third-party software. Horizontal autoscaling is a mechanism that allows Kubernetes resources to scale by means of creating new replicas of a container image, this scaling is dependent on parameters like resource utilization. Kubernetes also offers more control over the resource, especially regarding access authorization.

The comparison is summarized in the following table:

[15] Kubernetes website: `https://kubernetes.io`

	Kubernetes	Docker Swarm
Complexity	Complex setup and management	Simple setup, seamless docker integration
Access and networking	Granular access policies on resources, advanced networking	Networking and access managed at the container level
Autoscaling	Supported at resource level	Manual

Table 1.1: Kubernetes and Docker Swarm comparison

Kubernetes requires a more complex setup and has a steeper learning curve compared to docker swarm, on the other hand, it is more powerful, offering more advanced features like autoscaling, built-in monitoring, and resource access management [5]. In order to overcome the complex setup of Kubernetes and empower users with all of its functionalities, most cloud providers have implemented a custom-managed version of Kubernetes as a dedicated service in their environment. Examples are Google Kubernetes Engine and AWS Elastic Kubernetes Service. A more in-depth view of the Kubernetes architecture and EKS is provided in Section 2.2.

1.6. Summary

As already discussed in length during this first Chapter, the rapid evolution of cloud computing brought about a series of advancements in the field of application development. Using cloud computing the cost of managing an application was significantly reduced, and the platforms also provided high vertical and horizontal scalability. These considerable upsides were also countered by new challenges. The new DevOps methodology was created in order to deal with the large number of heterogeneous but interconnected tasks that were part of the application development and deployment process, and Infrastructure as Code became an integral part of the process. Micro-service architectures rose to popularity, simplifying development, testing, and deployment, becoming ideal for a CI/CD-enabled development process. To efficiently manage the multitude of containers needed by micro-service applications, container orchestrators such as Kubernetes were designed.

Kubernetes offers a multitude of benefits: infrastructure abstraction, ease of deployment, simple and efficient scaling, built-in monitoring, and failure detection and reaction. Kubernetes has become pivotal to most if not all medium-to-large scale cloud applications, and yet correctly using the technology is still challenging. Firstly, creating and deploying a Kubernetes cluster is complex, and requires lots of experience. Then there is also the matter of mapping the application to the architecture. Currently, the application

components and the infrastructure are separated, there is no single IaC language that allows to define in a single file application component definition, infrastructure definition, and component-to-infrastructure mapping. Complex CI/CD pipelines, involving different technologies, are commonly used to automate deployment of software components on Kubernetes clusters. These pipelines are complex to build, require deep knowledge of a range of DevOps and IaC technologies, and are prone to breaking when one of its composing tools undergoes updates or changes.

For these reasons, this Thesis deals with the matter of integrating Kubernetes into the DOML. The final objective is to provide the DOML user with the facilities needed to deploy and manage applications composed of a large number of replicated and interconnected containers through Kubernetes, thus simplifying and centralizing in a single tool the definition of the application components, the deployment of the cluster, and the mapping of software components to the Kubernetes infrastructure. The following Chapter contains a more in-depth overview of Kubernetes and its components and features and formally defines the scope and objective of the integration of Kubernetes inside the DOML.

2 | Problem Statement

The problem this Thesis tackles is that of incorporating a Container Orchestration Platform (see Section 1.5) into the DevOps Modeling Language. This Chapter addresses the two topics that are pivotal to this Thesis: Kubernetes and the DOML. The first part of the Chapter identifies the problem that this Thesis tries to solve, and provides an in-depth overview of Kubernetes. The second part describes the current capabilities of the DOML in the field of Container Orchestration, exposes its limitations, and proposes the integration of a Kubernetes-based orchestrator in the form of Amazon EKS (see Section 1.5.3).

2.1. Problem Identification

As already introduced in Section 1.6, Kubernetes has become fundamental for any medium-to-large scale application deployed on the cloud. The technology's capabilities significantly decrease the effort needed for deploying and managing a large number of containers. Kubernetes is fundamentally an abstraction of its underlying infrastructure, on which the user can declaratively deploy resources. Containers are deployed on the cluster, each container can be replicated for redundancy, reliability, and load balancing. Containers can then communicate through the Kubernetes network and scale according to a variety of metrics. Kubernetes continuously monitors the resources' state, can detect faults, and restart failed containers. This allows the user to avoid managing the state of a deployed application and enables effective and reactive scaling. On top of that, the abstraction means that the user simply deploys resources on the cluster and not on the single node, Kubernetes will take care of deploying the resources on one of the available nodes. Likewise, the load-balancing rules are not managed by the user, Kubernetes applies them automatically when a container is attached to one. All of these reasons are enough to prove the need to use Kubernetes in a cloud application. The challenge now becomes how to create a Kubernetes cluster and how to deploy resources on it.

The first problem is perhaps the trickiest, creating a Kubernetes cluster is a very long and complex process that requires deep knowledge of the subject and experience on Kubernetes-related projects. The structure of a Kubernetes cluster will be explored more

accurately in the next Section, but we can already introduce that it requires at least two nodes: a master node and a worker node. As such creating a cluster requires provisioning one or more virtual machines (using more than one is ideal to assure reliability), configuring all of the necessary Kubernetes components on the corresponding machine, then we must handle networking, opening the correct ports, starting communication, and configuring access to the master node. These steps are not trivial, the configuration is very long and should be customized to the particular use case, then there is also the matter of correctly sizing the virtual machines on which the cluster is deployed, and these VMs must then be monitored, updated, and secured.
This is already a tall obstacle to climb, especially for non-expert users, such as those who are the main audience of the DOML. On top of this technical problem, we have a much more logical one. The current landscape of DevOps and IaC technologies does not offer a single tool that is able to put together the definition of the application's components, the cluster's infrastructure, and the mapping of the components to the infrastructure. Right now, the most common way to deploy an application component on a cluster is through complex CI/CD pipelines, that use a multitude of tools to achieve this goal. These pipelines are hard to create, harder to debug, and prone to breaking if one of the composing tools were to be updated or modified.

As a result, this Thesis tries to create a single tool that puts together the application's software architecture, the Kubernetes cluster definition, and finally the mapping of the software components onto the cluster. This is achieved by introducing Kubernetes into the DOML and leveraging the language's feature of connecting the application and the infrastructure layer. Putting the three previously described processes in a single tool decreases the barrier of entry, bringing down the number of DevOps tools needed to create a modern cloud application to 1, and simplifying and improving the user experience. The final Section 2.5 of this Chapter formalizes this objective and proposes the requirements that will be used to validate the integration.

2.2. Analysing Kubernetes Architecture

To better understand how to integrate Kubernetes inside the DOML, an in-depth study of the Kubernetes components is necessary. This study was carried out based on the works of: Arun Gupta and Alan Hohn [15], and that of Marko Luksa [23].

2.2.1. Overview

Kubernetes is an open-source container orchestration platform that allows to declare the desired state of an application using two main concepts: deployments and services. The engine starts the application in the desired state and constantly monitors the status of its containers. If at any point in time, the actual state differs from the desired state, then Kubernetes performs one of the following actions: auto-restart the container, re-schedule a container on a different node, or replicate a resource. Anything in Kubernetes is a resource described by a YAML file, the file contains both the description of the resource, effectively its desired state, as well as its current state.

2.2.2. Cluster, namespace and nodes

A Kubernetes Cluster defines the perimeter comprising all of the resources that can be accessed using Kubernetes. Resources are grouped logically into namespaces, the name of a resource is unique inside a namespace, and this effectively creates logical clusters from a single physical one. A cluster consists of one or multiple nodes, these nodes are transparent to the user, who sees the cluster as one monolithic virtualized machine containing a variety of resources.

2.2.3. Deployments, Pods and Services

A Pod is a group of one or more containers. A pod is the smallest unit of resource that can be addressed in a Kubernetes cluster. All containers of a pod are deployed to the same host and in the same namespace and share the same IP address. This effectively creates a local network inside each pod, whereby containers of the same pod communicate through localhost. For this reason, containers in a pod can also share volumes, where a volume is a form of persistent storage attached to one or multiple containers. Kubernetes constantly monitors the status of each pod and has the power to restart, terminate, scale up, or scale down the pod.

A Deployment is used to declare a replica set of pods, meaning each pod is replicated n-times where n is a user-defined number that can be changed dynamically at runtime. It is up to Kubernetes to query the deployment state and make sure the correct number of replicas of a pod is always up and running. Kubernetes schedules each pod on one of the available nodes, we have no control over this assignment.

A Service allows a pod to communicate with the Kubernetes network. Each pod is assigned an IP when created, but pods are meant to be ephemeral and are often shut down and

restarted. Services are attached to a pod by means of a deployment, in doing so we assign a static permanent IP to a pod which is shared among all of its containers. The IP address can be private, only accessed from other resources in the cluster, or public, exposed to the network in which the cluster is deployed. Services can even perform load-balancing duties.

2.2.4. Kubernetes architecture

Kubernetes uses a Client/Server architecture. The User or Client interacts with the cluster through the Master node, in which the Kubernetes controller is hosted, the entry point of the Master is the kube-apiserver component. The apiserver then interacts with the other Master components: the kube-controller manager acts as the main controller for the whole cluster and determines what actions must be performed on the resources; the etcd is a distributed key-value store that holds the state of each resource in the cluster; the kube-scheduler is responsible for deciding which node a pod should be deployed on. The master node is often replicated in order to achieve high availability, master nodes live in a logical space called control plane.
The controller then interacts, always by means of the apiserver, with the pods on the Worker Nodes by communicating with kubelet and kube-proxy. Kubelet is a service that runs on every node, it translates commands from the apiserver into commands to the docker engine in order to manage containers inside the pods. Kube-proxy manages networking rules inside a node, ensuring correct routing to each pod.

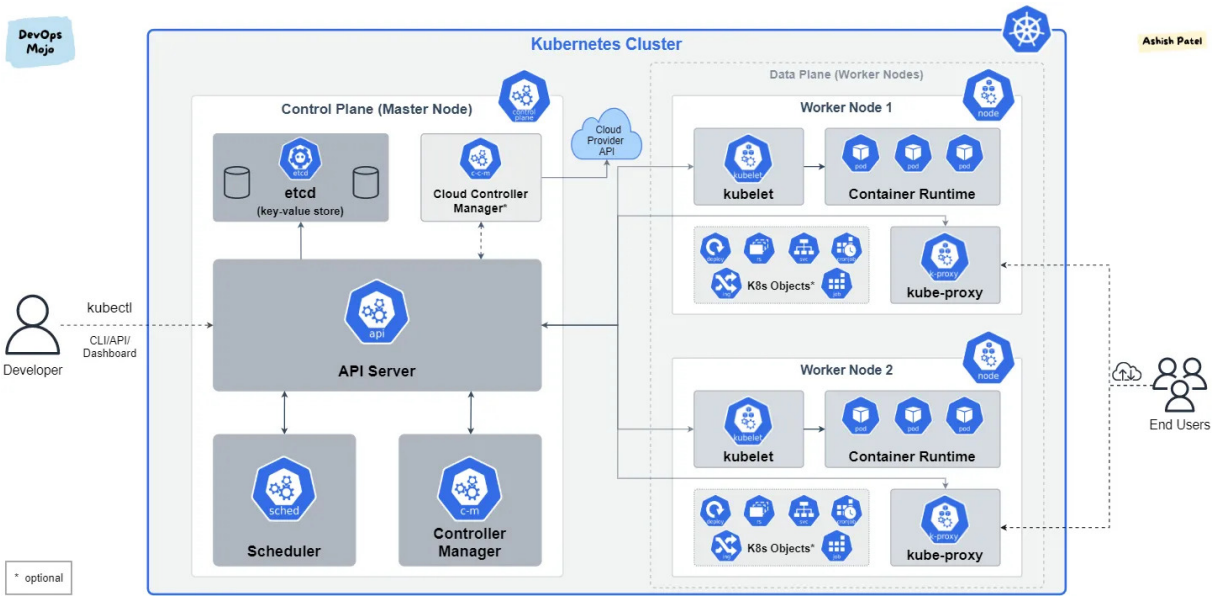

Figure 2.1: Kubernetes architecture, from the article: Kubernetes — Architecture Overview [20]

This architecture (fully represented in Figure 2.1), while surely complex, proves to be a very powerful abstraction over the cluster, by means of which the user is not burdened with the management of the single nodes in the cluster or even the single pods, but is simply tasked with describing the state of the application and relaying it to the cluster, interacting with the master node of the cluster through well-defined APIs. Kubernetes then takes care of maintaining that state consistent.

2.2.5. Scaling deployments

Deployments can be scaled up or down from the API, by setting the number of replicas of a pod. A more reactive approach can be achieved by setting up Horizontal Pod Autoscalers. An HPA is a special kind of resource attached to a pod, the HPA will monitor metrics from the pod such as resource utilization or other custom metrics, and then react by scaling the number of replicas according to the application's needs. The upper and lower limits of the replication are specified in its YAML. This mechanism empowers each pod with a simple yet very effective autoscaling mechanism to cope with the needs of the application.

2.2.6. Custom Resource Definition

Custom Resource Definition or CRD is the mechanism built into Kubernetes that allows the creation of resources of types external to the base Kubernetes installation environment,

these resources are called Custom Resources. These third-party CR extend the Kubernetes API, introducing new functionalities into our cluster. A CRD is defined through a YAML file of type CustomResourceDefinition, this by itself extends an existing resource and assigns to it a new label. The real power of CRD comes from pairing them with controllers. Controllers are Kubernetes resources that constantly monitor other resources, by defining CRDs we can listen to events published for a particular CR and perform some specialized logic.
CRDs allow us to integrate into Kubernetes virtually any concept, we can add the concept of a Database, an Application, and a Monitoring tool. Controllers will then monitor these CR and perform some actions accordingly, for example replicating data across the Database nodes, checking the internal state of the application, or exporting custom metrics from some resources. Every CRD is built on top of the basic Kubernetes resources but packaged in a way that is very simple to install and use in the cluster.

2.2.7. Drawbacks of Kubernetes

While Kubernetes itself totally abstracts the underlying architecture and nodes, when setting up a Kubernetes cluster the tasks of provisioning all of the VMs that make up the cluster and configuring them to run the correct Kubernetes components are unavoidable, just like the task of handling networking the nodes in a correct way. In order to avoid this complex setup requirement and further abstract the infrastructure, cloud providers often implement a managed version of Kubernetes and offer it as a service. The idea behind that is simplifying the setup of the cluster, thus creating another level of abstraction on top of Kubernetes that removes the matter of managing the servers on which the cluster runs, and integrating networking and security inside the cluster with the rest of the cloud environment.
Although part of the burden of creating the cluster and managing the nodes can be simplified using such managed services, another important problem is still standing. Once the cluster has been created, the next step is that of deploying the required containers on the cluster. While IaC languages such as Terraform can be used to provision and configure Kubernetes, automating the deployment of application elements usually requires configuring some form of CI/CD pipeline with technologies such as GitHub actions or Jenkins. There is no single language that provides in a single point the definition of the cluster, the definition of the application components, and the deployment of components on the cluster.

2.2.8. EKS

This Thesis then proposes the integration of AWS Elastic Kubernetes Service inside the DOML. EKS is an abstraction of a Kubernetes cluster that offers easy deployment, management, and scaling of containerized applications. In particular, EKS offers a fully managed control plane with automatic scalability, to guarantee the dependability and availability of the service. Worker nodes can be either Amazon Fargate instances or EC2 instances. On top of that, EKS is tightly integrated with the rest of the AWS resources. Container images can be pulled from the Amazon Elastic Container Registry, persistent storage can be attached to a container using Amazon Elastic Block Store, and Amazon's Virtual Private Cloud can be used for networking [22]. As far as this Thesis goes, only a cluster with EC2 instances will be implemented, since the EC2 service is already integrated in the DOML. Once the integration is concluded, the DOML will be able to handle in a single file the definition of the application components, the declaration of the Kubernetes Cluster through EKS, and the mapping and deployment of the components to the cluster.

2.2.9. EKS Alternatives

EKS was not the only possible route for introducing Kubernetes into the DOML. Setting up Kubernetes is technically possible in the current version of the DOML since it can be achieved by provisioning and configuring virtual machines. The process was already discussed in Secion 2.1, it is far from simple, quick, and straightforward, and cannot be carried out by an inexperienced user. Correctly configuring a Kubernetes cluster requires deep knowledge of the architecture, a very good understanding of the workload and scope of the application, as well as extensive experience in creating clusters. On top of that the cluster is likely to undergo modification on the architectural level as the application matures and evolves.

The choice of using EKS instead of providing a simpler way to provision and configure VMs for the cluster is the result of an analysis of the nature of the task, as well as the strengths of the DOML. Although we could have chosen to mimic the manual creation of a cluster, parametrizing some of its steps and mapping this resource to a DOML class, this choice would have proved too complex, and disconnected from the main goal of DOML. The matter of configuring the cluster in a way that is most suitable to the application is incredibly complex, we could not possibly identify a restricted set of parameters that can fully configure a manually created cluster. As a result, this route would have created DOML classes that are too complex and clash with the initial goal of the DOML of being

a simple language that requires a low barrier of entry from the user. Even overlooking this problem, the created cluster would have to be managed and scaled, adding a further level of complexity. EKS on the other hand is fully managed, simple to instantiate, and provides all of the same features as a custom installation, minus some very workload-specific optimizations.

2.3. DOML Language Structure

As briefly introduced in Section 1.4, each DOML model is composed of three layers, each containing different classes.

Application Layer (AL): containing classes to define the software components of an application. In this layer, it is possible to define either a generic piece of software (i.e. a generic piece of code or a more complete application) or a SAAS as parts of the application. Each component can expose or consume interfaces, associations used to describe connections, dependencies, and interactions between components.

Abstract Infrastructure Layer (AIL): contains classes describing generic and abstract components of an infrastructure, such as computing nodes, or networking and security elements. For example, VMs are defined in this layer, each with its own interfaces, OS, and images. VMs can be grouped into AutoScalingGroups for scaling purposes. Containers are also defined here and can be aggregated into ContainerGroups. AutoScalingGroups and ContainerGroups are presented in a more complete way in Section 2.4. In this level, we also handle networking and security, by defining networks, subnets, and network interfaces to the outside or used within our network, as well as security rules to manage access to our components.

Concrete Infrastructure Layer (CIL): provides classes that map the abstract infrastructure components from the AIL into the concrete components from a service provider. Most AIL elements are mapped one-to-one to CIL elements. Inside the CIL we can define more than one concrete infrastructure, in which we map AIL elements to CIL elements, but only one can be active at a time. Inside each concrete infrastructure declaration, it is possible to define more than one provider, but not every AIL element has a corresponding mapping element in all providers. For instance, OpenStack supports fewer services than AWS at the present time.

The additional **Common Layer** contains miscellaneous utility classes, such as those representing a generic DOML Element from which each other class inherits, or the Configuration class, which is responsible for mapping AL elements to AIL elements.

2.4. Existing DOML orchestration elements

The DOML, in its current form, already integrates some orchestration technologies [10].

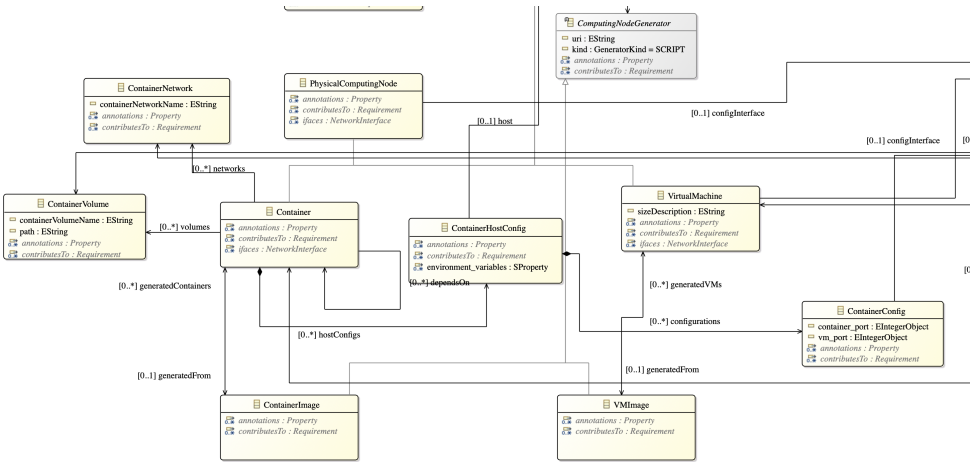

Figure 2.2: ContainerNetwork, ContainerVolume, and related classes, from DOML specification [10]

The *ContainerNetwork*, *ContainerVolume* and *ContainerGroup* classes of the AIL (see Image 2.2) allow the creation of a collection of homogeneous containers sharing the same network and volumes. This solution satisfies one of the requirements of a container orchestration tool (see Section 1.5.2), that of building a unified network to interconnect a group of containers. The load balancing aspect could be accounted for by connecting extending the *ContainerGroup* class to support the use of a load balancer using the *LoadBalancerKind* enum. But the capabilities of this approach end here, there is no autoscaling, container replication, or detection and management of failures.

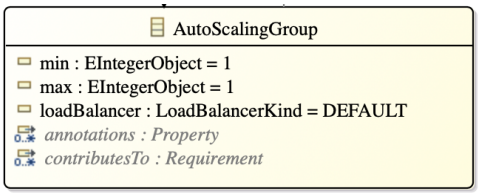

Figure 2.3: AutoScalingGroup classes, from DOML specification [10]

Orchestration of Virtual Machines can be achieved by using the *AutoScalingGroup* class (see Image 2.3), which allows to autoscale the number of computing nodes for a VM and load balance between the nodes. This approach is limited with respect to a technology like Kubernetes. While it is possible to define scaling policies, scaling VMs is much trickier and

takes longer than scaling containers, which are designed to be lightweight and portable. On top of that communication between VMs of the same AutoScalingGroup is all but simple.

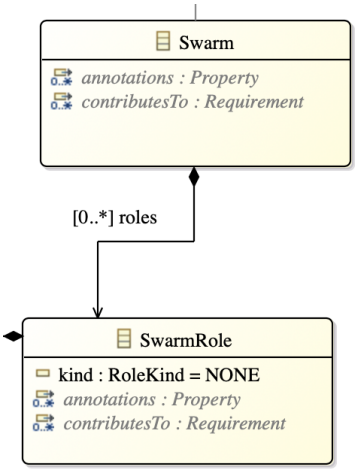

Figure 2.4: Docker Swarm classes, from DOML specification [10]

The final form of container orchestration is that of using the *Swarm* class (see Image 2.4), which is meant to represent a Docker Swarm (see Section 1.5.3). At present the class is not implemented, meaning it is not recognized by the ICG.

It is abundantly clear that the provided DOML orchestration tools are not enough to support modern micro-service and container-driven architectures. In its current form, DOML cannot support the orchestration of heterogeneous containers, it cannot create an environment where different containers are deployed and can communicate. On top of that, DOML does not offer any means of scaling or replication of containers.

	Kubernetes	**DOML**
Container deployment	Simple and declarative	Supported by the Container class
Networking	Support for advanced networking features	Support for networking between heterogeneous components possible using ContainerNetwork, but usage is complex
Replication	Simple and declarative	No replication support
Autoscaling	Supported at resource level	No autoscaling support
Failure detecion	Built-in restart and notification for failed containers	No built-in solution

Table 2.1: DOML and Kubernetes comparison

A summary of the comparison of DOML orchestration features with respect to Kubernetes

can be found in Table 2.1. In order to provide the missing features, this Thesis sets the goal of implementing Kubernetes, in the form of the AWS managed service EKS, inside the DOML using DOML-E and the ICG extension mechanism.

2.5. Integration Objectives and Requirements

One of the primary goals of DOML is to make the design and deployment of a web application simple and fast, all while providing a simple learning curve for the language. For this reason, the scope of integration of this Thesis revolves around extending the DOML to provide support for AWS Elastic Kubernetes Service. The newly expanded language has been called DOML-K8s, an extended version of DOML with support for Kubernetes. The DOML-K8s fulfills the following requirements:

- Validation of the DOML extension mechanism DOML-E. Extending the DOML with new elements that can be used to create a cluster in EKS with minimum knowledge requirements.
- Simple creation of a Kubernetes Cluster by means of EKS. Definition of EKS inside DOML-K8s, and support for IaC generation through the ICG extension mechanism.
- Quick setup of an application comprising multiple containers. Allowing users to declare which containers have to be deployed on the cluster and how they connect to each other.
- Seamless integration of the cluster within a larger application. Containers within an EKS cluster must be able to communicate with other resources belonging to the same application.
- Autoscaling of containers based on application workload. The integration must be able to support the autoscaling of Kubernetes resources when needed.

2.6. Summary

As presented in the previous Chapter, the need for container orchestration is growing, as an increasing number of applications are being developed with a cloud-first philosophy, and the reference architecture is shifting toward micro-services. The Thesis aims at integrating Kubernetes, in the form of AWS EKS, into the DOML. The final objective is to provide the user of the DOML with a powerful orchestration platform, which can support all sorts of consumer and enterprise applications based on containers.
Kubernetes' architecture revolves around two main elements: the control plane, containing

the master node which manages the entire cluster; and the worker nodes, where containers are deployed. The user defines deployments, declarative yaml files containing information regarding the container that has to be deployed: the image, the number of replicas needed, the type of networking used from the container (open ports, assigned IP addresses, connections external or internal to). The master node then handles the deployment of the containers on the worker nodes and ensures that the state of the cluster is always the same as the one defined by the user. Putting together the cluster's master-slave architecture is not simple, for this reason, most cloud providers offer a managed version of Kubernetes which releases the user of all of the configuration effort for the cluster.

In its current version, the DOML does not support a fully capable orchestration platform. For this reason, this Thesis set as its main objective the inclusion of EKS, Amazon's managed implementation of Kubernetes, inside the DOML, creating DOML-K8s. Such inclusion must be simple to use and require minimal knowledge from the user about how Kubernetes itself works. The user must be able to easily define the cluster and the application components that need to be deployed on such cluster.

The next chapter presents the work that has been done in order to implement EKS inside the DOML, both in terms of extension of the DOML and modifications to the ICG.

3 | Implementing Kubernetes inside DOML

This Chapter will describe the modifications made to the DOML metamodel and to the ICG, which allowed for the integration of EKS inside the DOML and the creation of DOML-K8s. The following Sections present the integration approach, describe the steps taken to perform the integration, and then compile a list of EKS components to be implemented in the DOML metamodel. Finally, the new classes are created as extensions of existing DOML classes, and the ICG is extended, in order to support the generation of the needed IaC.

3.1. Extending the DOML and ICG: Overview

Both the DOML and the ICG provide users with a versatile extension mechanism, that can be used to support new cloud resources and concepts. The DOML-E extension mechanism allows the creation of new elements by extending existing DOML metamodel classes. The extension process can be performed from the PIACERE IDE, which serves as the entry point for all of the framework tools.

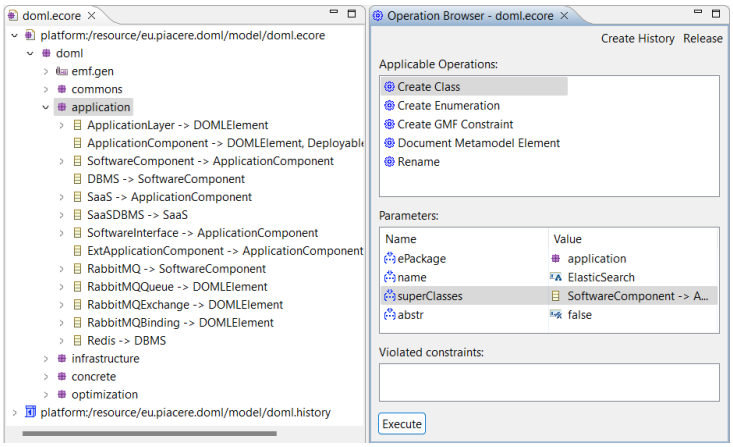

Figure 3.1: DOML-E extension menu, from [2]

The extension menu, pictured in Figure 3.1, allows two basic forms of extension. Properties extension can be used to refine existing classes, adding new properties with the objective of introducing new features to concepts already described in the DOML. For example, we can think of extending a Storage class with the encryptionType property, to define the encryption schema for the storage element. Classes can be extended as well, extending existing DOML classes allows for the specialization of concepts that are deemed to be too generic, or can be used to introduce new forms of existing elements. For example, we could extend the SaaS class to introduce a new specific SaaS service or extend the Container class to support another form of containerization.
The ICG was already introduced in Section 1.4.2. At a high level, it can be defined as a parser that reads DOML files in the DOMLX XML format and generates the corresponding Infrastructure as Code. The generation mechanism is template-based, meaning the ICG will read a DOML resource, and use its properties to populate a template in order to generate the IaC output file. Each IaC target language has a plugin, an ICG module that is able to parse the resources for that target language, and a set of templates for its resources. There are two possible ways to introduce new concepts into the ICG: adding a template for an existing target language, introducing the capability of generating code for a new resource in an existing language, and adding a plugin for the parsing and translation of a new target language. The ICG architecture and extension mechanism will be better explained in Section 3.4.

The insertion of new concepts into the DOML language is a multi-step process. The first fundamental step is the analysis and design of the new classes that must be added to the metamodel. This requires both determining names and attributes of the classes, as well as choosing the parent class that will be extended to become the new class, using the DOML-E extension mechanism (see Section 1.4.3). Once this first phase is completed, implementing the new elements is as simple as using the PIACERE IDE to extend the classes and update the language syntax, as described by Sergio Canzonieri in his Thesis [2]. Once the metamodel extension is completed, the final step is the extension of the ICG.

For the purpose of this Thesis, we need to use both the template and plugin ICG extension mechanisms. In order to introduce the EKS resource into the DOML, it is necessary to add the EKS template which will be translated into a terraform file to provision the resource on AWS. Then, to deploy resources on the created EKS cluster, we must also support the YAML target language to create Kubernetes deployments (see Section 2.2). This requires the creation of the YAML plugin and the addition of the required templates. Once these steps are completed, we are able to create Kubernetes applications using the DOML and generate the IaC code to deploy the application.

3.2. Kubernetes components to be modeled

The first step is determining what concepts need to be modeled in the DOML. For this purpose, we have divided the elements into two groups: *where* and *what*. Where refers to the modeling of the Kubernetes infrastructure, while what refers to classes modeling the application components deployed on the infrastructure.

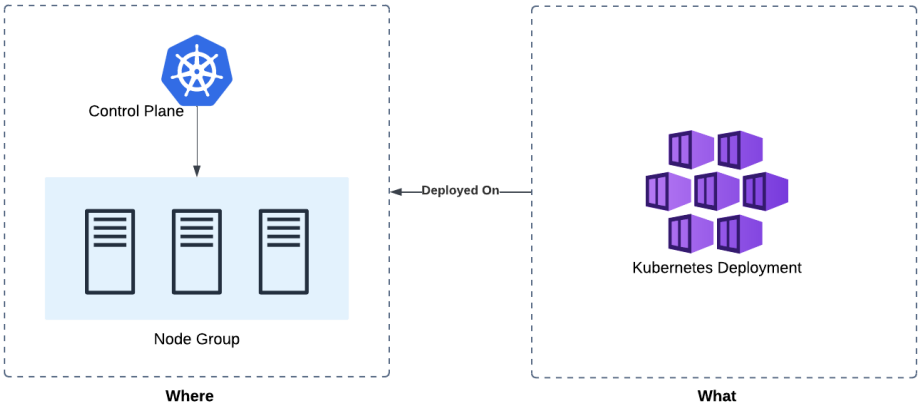

Figure 3.2: Logical division in what and where

For a clear distinction between *what* and *where*, suppose we take as reference an identity management application following a micro-service architecture. The application offers various services such as login, sign-up, and identity verification, with each service being implemented by a separate container deployed in a Kubernetes cluster. The *where* refers to the infrastructure comprising the Kubernetes cluster, the cluster nodes, the network where the cluster is deployed, and the access and security rules for the cluster. The *what* are all of the resources deployed on the cluster, meaning the containers inside the pods, the pod replicas, the deployment, the services used for networking, and the HPAs. This abstraction provides a strong division between infrastructure and application, which is logically useful to define the elements that the DOML should model, and what layer they should belong to. An overview of this logical division can be seen in Figure 3.2.

Where

For the *where* part we need to create an EKS cluster, the definition of a cluster using Terraform requires various parameters. First of all, we must assign a unique name to the cluster, then attach to it a vpc (the ID of the AWS virtual network in which the cluster is

deployed) and some security groups (firewall rules that manage access to the cluster) for networking and security purposes. It is also possible to make access to the cluster public and specify a range of addresses that can access the cluster. All of these parameters must be attributes of the EKS cluster class in the DOML. Then we must also define the worker nodes connected to the cluster, to do so we define one or more node groups, groups of EC2 instances that are used by the cluster. For this purpose, we preferred to model the node groups as a separate class, with a one-to-many relation between the cluster class and the node group class. The node group class must have a name, an EC2 instance type, and then must specify the minimum, maximum, and desired size of the group, which is used by the cluster to scale the node group resources when needed.
To summarize we need to create two classes, one for the cluster called KubernetesCluster and one for the node group called KubernetesNodeGroup, in the infrastructure layer of the DOML. We also need to create a class for the cluster in the concrete layer, which maps to the infrastructure layer cluster class. Since there are no classes that share attributes with the ones we need to create, the only option is to extend respectively the ExtInfrastructureElement and ExtConcreteElement classes, which were created for the purpose of being the superclass for elements that are not represented in the DOML.

What

For the *what* part, we need to model the containers to be deployed on the cluster, this basically means modeling deployments, pods, services, and horizontal pod autoscalers. In order to model a pod, we must define a class with two fundamental attributes: the URL of the Docker image used to create the container hosted on the pod, and the exposed ports for the container. Kubernetes allows the definition of multiple containers within a single pod, but for the sake of simplicity, we decided to implement only single-container pods. The exposed ports require further consideration because a port exposed on a pod is still inaccessible in Kubernetes without first attaching a service to that pod. For this reason, we decided to create a separate class to model exposed ports in a container, this class must allow the user to map an internal port of the container to an external port of the pod, and to define the type of service attached to that port, either a ClusterIP or a LoadBalancer (see Section 2.2.3). This results in the creation of two classes, both in the infrastructure layer, the deploymentImage, and the podExposedPort class. Both are extended from the ExtInfrastructureElement class since the existing elements to model container images have extra attributes that are not needed. In particular, the existing ContainerConfig class used to model exposed ports on a container maps the port on the container to a port on the VM hosting the container, while the ContainerClass currently

used to model containers has attributes such as the networks the container is deployed in and the volumes attached to the container, which are not used for the purpose of EKS. One final class to be added to the AIL is the KubernetesAutoScaler class, which is used to create an HPA attached to a deployment, here we must define the scaling metric, as well as the minimum and maximum number of pods desired. Once again this class must inherit from the ExtInfrastructureElement.

Linking Where and What

Finally, we need a way to connect the infrastructure declared in the *Where* part with the application-related elements declared in the *What* part. This is done through a deployment in which we declare the pod image, the number of replicas of each pod, and attach services and HPAs to the pods. We are thus able to map components of our application to an EKS cluster through the deployments. The best place to do so is in the Deployment part of the DOML. The existing Deployment class of the DOML, part of the Commons layer, is responsible for mapping a software component to an infrastructure node. So the most sensible choice is that of creating a KubernetesDeployment class, which extends the Deployment class, mapping an ApplicationComponent to a KubernetesCluster. The class must also have an attribute that references the deploymentImage that implements that ApplicationComponent, and attributes for the number of replicas of the container and the namespace for the deployment.

While these classes are certainly enough to define a Kubernetes application, there is an important aspect still missing. One hidden strength of Kubernetes is its community. Being an open source technology, used in most of the medium-to-large scale cloud applications, a lot of premade resource definitions in the form of YAML Kubernetes files can be easily found online. Kubernetes itself allows extensive usage of existing resource definition by allowing the user to deploy a resource through a URL (usually pointing to a git repository), instead of a local file. This aspect becomes particularly relevant when we take into consideration CRDs, which were previously discussed in Secion 2.2.6. If the user found himself having to use a custom resource, he could use the DOML-E extension mechanism to add the DOML-K8s metamodel classes for that CRD and extend the ICG. But this cannot be considered a simple or quick solution and is certainly one that would undermine the versatility of the DOML by requiring the user to extend its functionality for every new Kubernetes custom resource needed. The solution is creating an additional class in the Commons Layer called ExtenalKubernetesDeployment, yet again inheriting from the Deployment class, which maps an Application Component to a Kubernetes Cluster through an external resource, by specifying its URL. This class can be used to integrate

existing custom resources which would be too cumbersome to add into the DOML-K8s, or for resources that are used only once and do not add meaningful features to DOML-K8s.

3.2.1. DOML-K8s flow of development

By introducing all of these new concepts, we are fundamentally changing the way in which a user develops a Kubernetes DOML application.

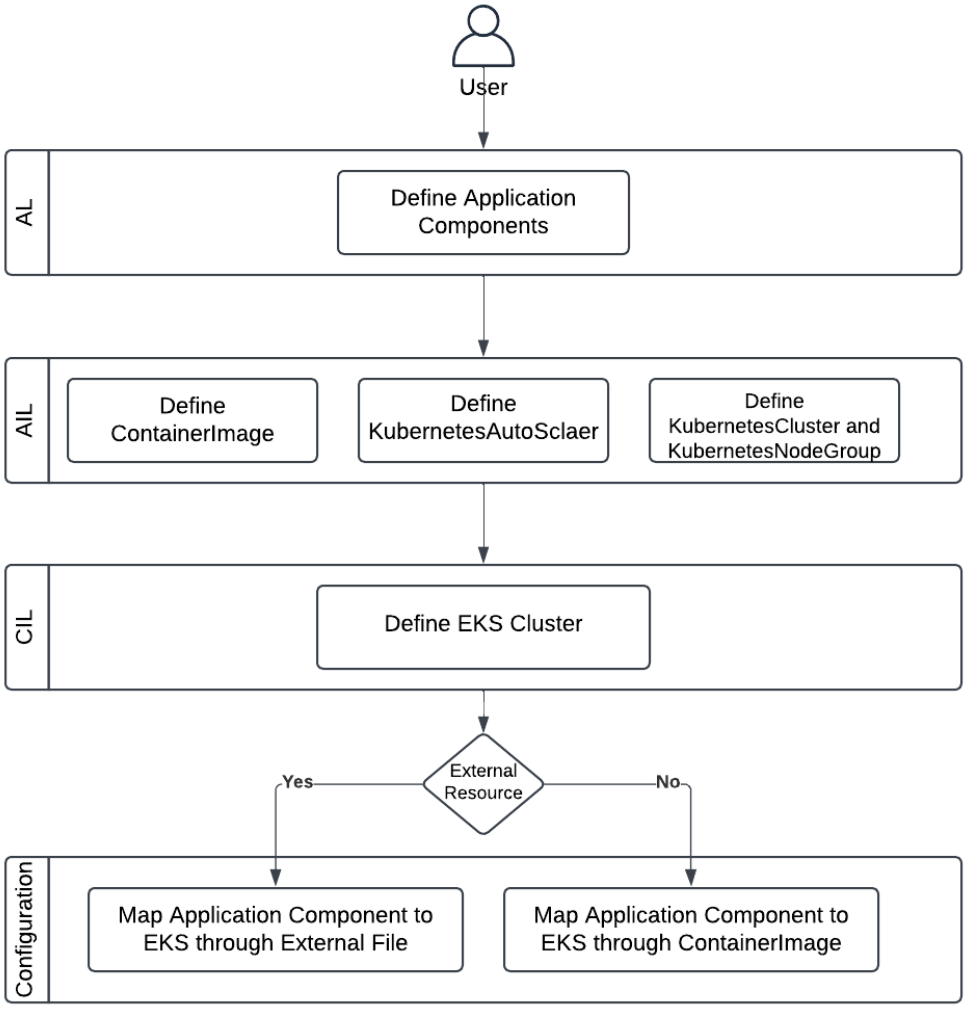

Figure 3.3: Kubernetes DOML application design flow

The above Figure 3.3 shows the flow a user follows when creating a Kubernetes application using DOML-K8s. For the sake of simplicity, we omit the definition of the parts of the application that do not directly concern Kubernetes.

- The user starts from the Application Layer, defining the Application Components.

The components of the application are declared, alongside their dependencies and interfaces. In this step, the user defines the services that make up the application.

- Then moving to the Abstract Infrastructure Layer, the user defines the Kubernetes cluster. The user declares one or more Kubernetes cluster and their nodes. Here the user also defines the networks and security groups that are attached to the cluster.
- Again in the AIL, the user also declares the images of the container that will be deployed on the cluster.
- The final resource that can be defined in the AIL is the KubernetesAutoScaler, here the user defines the scaling metric and the bounds of the number of replicas.
- In the Concrete Infrastructure Layer, the user defines the EKS implementation of the cluster that maps to the previously defined AIL cluster.
- Finally in the configuration section, the user maps an application component to the Kubernetes cluster through the container image or an external Kubernetes file. Here the user can also attach an autoscaler to the component.

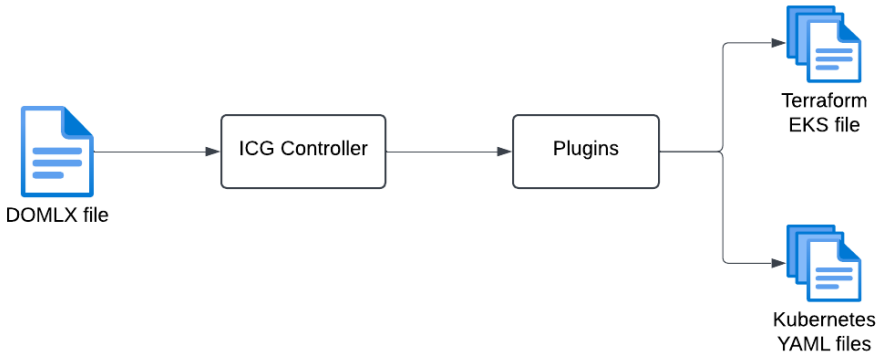

Figure 3.4: Kubernetes DOML application code generation

Once the DOML file has been written, the flow pictured in Figure 3.4 is followed to generate the IaC files. The user calls the ICG, which takes as input the DOMLX representation of the DOML file. The ICG generates both the EKS terraform file, used to provision the cluster, and the yaml deployment files to define the resources to be deployed on the cluster.

3.3. New DOML elements

This Section described the new classes that were introduced inside the DOML metamodel. The Application Layer was not modified, as such is omitted below.

3.3.1. Infrastructure layer

The Abstract Infrastructure Layer holds classes for the definition of an abstract infrastructure decoupled from the choice of cloud provider (for a more exhaustive description see Section 2.3). The following classes were added to the AIL:

KubernetesCluster

The KubernetesCluster class represents an EKS cluster, including the vpc, subnet, and security group for the cluster and the nodes, the boolean to set the public access to the cluster and the accepted addresses (used if the cluster is set to public), and the node groups attached to the cluster.

Superclass
ExtInfrastructureElement

Attributes
publicAccess: boolean[0..1] sets the cluster to be publicly accessible from the internet, defaults to "true".
acceptedAddresses: string[0..1] range of addresses that can access the cluster if the access is public, defaults to [0.0.0.0/0].

Associations
nodeGroup: KubernetesNodeGroup [1..*] the node groups that are used as worker nodes for the cluster.
clusterSecurityGroup: SecurityGroup[0..*] list of security groups for the cluster, optional.
nodeSecurityGroup: SecurityGroup[0..*] list of security groups for the nodes, optional.
deploymentNetwork: Network[1..1] vpc network where the cluster is deployed.
deploymentSubnets: Subnet[0..*] subnets where the cluster is deployed.

KubernetesNodeGroup

The KubernetesNodeGroup class represents a set of EC2 nodes used by an EKS cluster as worker nodes, including the minimum, maximum, and desired size for the node group, and the type of EC2 instance.

Superclass
ExtInfrastructureElement

Attributes
minSize: integer[1..1] minimum number of instances in the group.
maxSize: integer[1..1] maximum number of instances in the group.
desiredSize: integer[1..1] initial number of instances in the group.
type: string[1..1] type of EC2 instance.

DeploymentImage

The DeploymentImage class represents a Docker image used to generate a container.

Superclass
ExtInfrastructureElement

Attributes
url: string[1..1] url for the image.

Associations
exposedPort: PodExposedPort [0..*] ports exposed by the container.

PodExposedPort

The PodExposedPort class represents a port exposed by a container.

Superclass
ExtInfrastructureElement

Attributes
containerPort: integer[1..1] port on the container mapped to the outside.
exposedPort: integer[1..1] external port mapped to the internal containerPort.
public: boolean[1..1] sets the exposed port to be visible only from inside the cluster, or also from the outside.

KubernetesAutoScaler

The KubernetesAutoScaler class represents a Horizontal Pod Autoscaler in Kubernetes. Each autoscaler has associated the scaling metric, the threshold, and the maximum and minimum number of replicas.

Superclass
ExtInfrastructureElement

Attributes
metric: string[1..1] scaling metric for the pod.
metricType: string[1..1] type of scaling metric, eg. average or maximum.
threshold: integer[1..1] threshold for the scaling.
minSize: integer[1..1] minimum size for the HPA.
maxSize: integer[1..1] maximum size for the HPA.

3.3.2. Concrete layer

The Concrete Layer contains classes that map AIL elements to services from a cloud provider (for a more exhaustive description see Section 2.3). All of the classes implemented here are mappings to the classes implemented in the AIL:

KubernetesCluster

The KubernetesCluster class represents the concretization of an EKS cluster.

Superclass
ExtConcreteElement

Associations
maps: infrastructure.KubernetesCluster [1..1] mapping to the cluster in the infrastructure layer.

3.3.3. Common Layer

The Common Layer comprises utility classes, as well as classes that do not neatly fit in the three main layers(for a more exhaustive description see Section 2.3). The *Configuration* class is already part of the Common Layer and is used to bind *Application Component* classes from the AL to VMs in the AIL. For this reason, the ideal way to map the application components to containers in the cluster is to use an extension of the Configuration class. Two classes have been introduced in this layer:

KubernetesDeployment

The KubernetesDeployment class maps an Application Component to a Kubernetes Cluster through a DeploymentImage. Also includes the number of replicas for the container, the namespace for the deployment, and the autoscaler associated with the resource.

Superclass

Deployment

Attributes
replicas: integer[0..1] number of replicas of the container, defaults to 1.
namespace: string[0..1] namespace in which the deployment is deployed on the cluster.

Associations
component: application.ApplicationComponent [1..1] component logically mapped to the cluster.
cluster: infrastructure.KubernetesCluster[1..1] cluster on which the component is logically deployed.
image: infrastructure.DeploymentImage[1..1] image implementing the application component.
autoscaler: infrastructure.KubernetesAutoScaler[0..1] autoscaler to be attached to the resource.

ExternalKubernetesDeployment

The ExternalKubernetesDeployment class maps an Application Component to a Kubernetes Cluster through an external deployment file.

Superclass
Deployment

Attributes
url: string[1..1] URL where the external deployment file is hosted.

Associations
component: application.ApplicationComponent [1..1] component logically mapped to the cluster.
cluster: infrastructure.KubernetesCluster[1..1] cluster on which the component is logically deployed.

3.3.4. Summary

The extension of the DOML into the DOML-K8s resulted in the creation of eight new classes. Five classes were added to the Abstract Infrastructure Layer, 1 was introduced into the Concrete Layer, and the remaining two to the Common Layer.

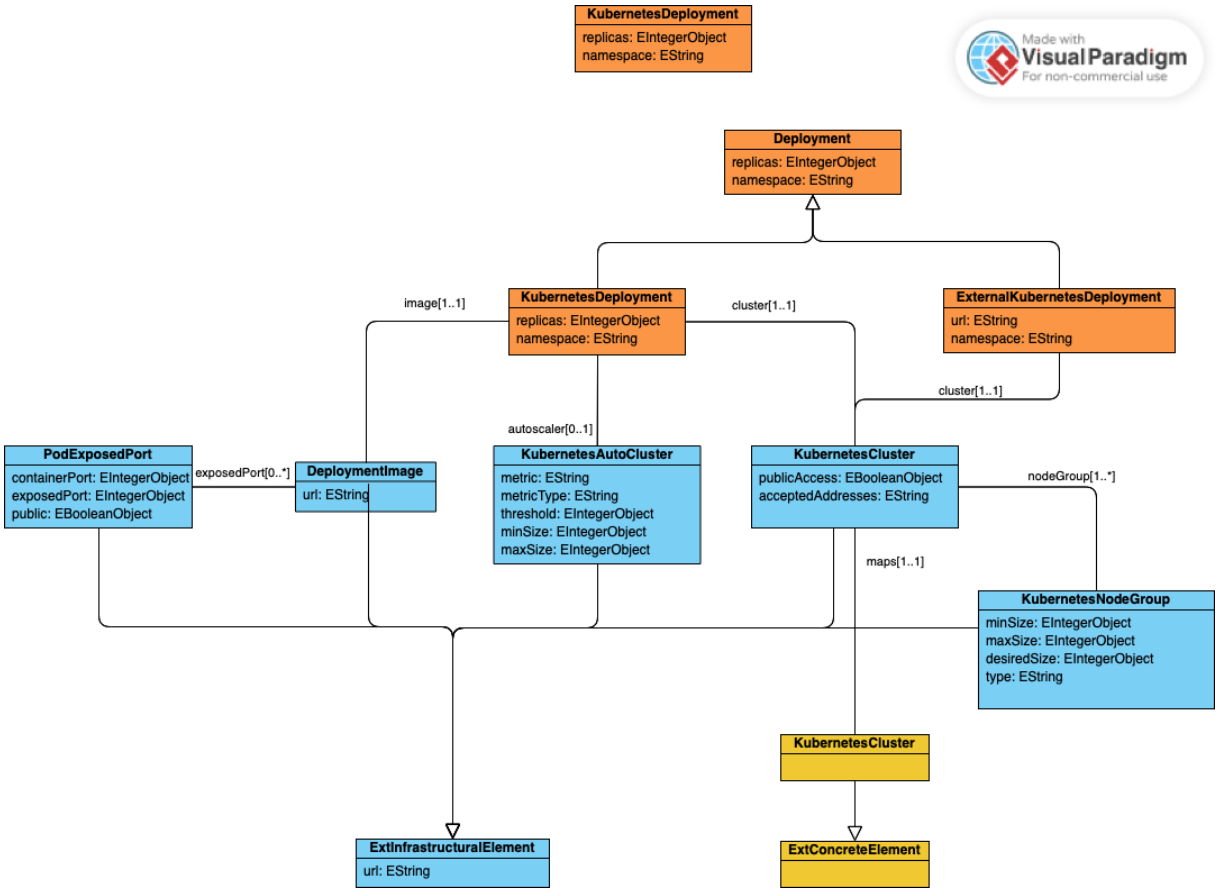

Figure 3.5: New DOML classes

The resulting classes are pictured in Figure 3.5, where AIL classes can be seen in blue, CL classes in yellow and Common Layer classes in orange. Associations with already existing elements are omitted for simplicity.

3.4. Extending the Infrastructure Code Generator

This Section provides an architectural description of the Infrastructural Code Generator and its technologies, before explaining the flow of execution that generates the target Infrastructure as Code from a DOML file. Then presents the modifications made to the ICG codebase to integrate the generation of IaC for EKS and Kubernetes deployments. The initial codebase for the ICG can be found in the Tecnalia repository [36] while the modified and extended code can be found on GitHub [30]. Throughout this Section, unless explicitly stated, we will reference the original Tecnalia repository.

3.4.1. ICG architecture

The Infrastructural Code Generator is a tool available through command line or REST API, which takes as input the DOML model in an XML format, called DOMLX, and generates the corresponding Infrastructure as Code files.

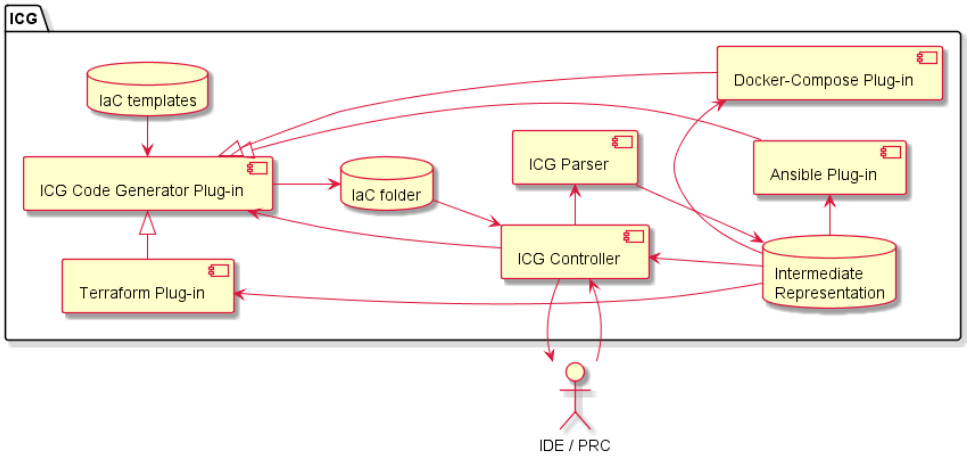

Figure 3.6: ICG architecture, from PIACERE deliverable [1]

The ICG is written in Python, Figure 3.6 shows its internal architecture.

The **ICG Controller** is the entry point of the application, it is invoked when calling the tool from command line or REST API and is responsible for managing the entire flow of the application.

The **ICG DOML Parser** is the component responsible for the parsing of the DOML model and subsequent creation of the Intermediate Representation. The DOML IR is a JSON data structure that describes the DOML model in a way suitable for the generation of IaC code. In particular, at the root of the IR, we have an array of JSON objects, each being called a step. Each step contains all of the data, from the DOML model elements, that are needed to generate the target IaC, as well as a label used by the ICG plugins to determine the target language of the step. For instance, the Terraform step contains information about all of the cloud resources that must be provisioned, while the Ansible steps contain information about how the software components must be configured.

The **ICG plugins** take care of code generation, the plugins are Python modules called by the Controller whenever a step of the IR must be translated into IaC. Each plugin is responsible for the generation of code for a single target language, for example, the Terraform plugin generates Terraform code for resource provisioning, while the Ansible plugin generates Ansible code for resource configuration. The code generation process

follows the Template-Based Code Generation strategy, by which each plugin selects a template from a library and fills the placeholders in the template with the data from the relevant IR step. Each template contains both a fixed part, and a variable part using placeholders. The templating library used by the ICG is Jinja2 [1], a flexible yet very powerful library that combines simple templates with advanced features such as variable definition and conditional structures. The TBCG approach has been chosen due to its simple but powerful nature, but also thanks to its extensibility. We can easily add new DOML elements by adding a new template, and support entire new target languages by adding the plugin and templates.

3.4.2. Flow of execution

In order to correctly extend the ICG we must first study its flow of execution. The following pseudocode represents a high-level abstraction of the steps needed to convert the XML representation of a DOML file into the target IaC, the components in action are those described in the previous Section.

Algorithm 3.1 ICG parsing algorithm

```
1: ICG is called from command line or REST API
2: The ICG Controller is invoked
3: The Controller calls the ICG Parser
4: The Parser generates the Intermediate Representation from the DOMLX
5: for each target step in the IR do
6:     The Controller invokes the target language plugin
7:     for each resource in the step do
8:         The plugin selects the IaC template
9:         The plugin populates the template
10:        The plugin writes the IaC to the output file
11:    end for
12: end for
13: The Controller compresses output files
```

The algorithm can be divided into 3 phases: the parsing phase (from lines 1 to 4) in which the ICG loads the model and metamodel, and creates the Intermediate Representation of the model based on the metamodel; the language choice phase (from lines 5 to 12), in which the ICG chooses the correct language plugin for each step of the Intermediate

[1] Jinja2 website: `https://jinja.palletsprojects.com/en/stable/`

Representation; the IaC generation phase (from lines 7 to 11), in which the ICG uses the plugins to load the template for each resource, populates the template, and writes the resulting IaC to the output file.
To extend the ICG we have to work on all three phases. Starting from the easiest job first, we need to add a Terraform template to create the Terraform code to provision EKS, which will be used in phase 3. We also need to add a new plugin for Kubernetes, capable of creating deployments in the form of YAML files. The new plugin must be accompanied by the template for a Kubernetes deployment, which for simplicity includes in a single file the deployment, services, and HPA. The choice of using a single template is explained in Section 3.4.5. Since we are adding a new plugin, we also need to modify the ICG Parser to generate a new step in the IR that can be translated into YAML Kubernetes files.

3.4.3. EKS template

The first and simplest step is that of creating the EKS template in Terraform. The base of the template comes from the Terraform repository [2]. The following properties are taken from the DOML model and used as parameters to populate the template:

- `cluster_name`: identifier of the cluster, used to refer to it and displayed in the AWS console.
- `cluster_endpoint_public_access`: boolean value that sets the cluster as either publicly reachable or not.
- `cluster_endpoint_public_access_cidrs`: cidr of public addresses that can reach the cluster.
- `cluster_security_group_id`: optional, list of IDs of the security groups for the cluster.
- `vpc_id`: id of the vpc in which the cluster is deployed.
- `subnet_ids`: id of the subnets in which the cluster is deployed.
- `node_security_group_id`: optional, list of IDs of the security groups for the worker nodes.
- `node_group_name`: name of each node group in the cluster.
- `min_size`: minimum size of the cluster in terms of number of instances.
- `max_size`: maximum size of the cluster in terms of number of instances.

[2] `https://registry.terraform.io/modules/terraform-aws-modules/eks/aws/latest`

- `desired_size`: initial size of the cluster in terms of number of instances.
- `instance_types` : type of the EC2 instance.

The template code can be found on GitHub: **eks.tpl**. For the ICG to recognize this template, we simply have to place it inside the folder `templates/terraform/aws`. We must also update the `template-location.properties` file, which is used as the central source of truth for all the ICG templates, associating each resource name with the path of the corresponding template.

3.4.4. Creating the Kubernetes ICG plugin

When creating the Kubernetes plugin, we took inspiration from the existing Terraform plugin. After all, the Kubernetes plugin can be seen as a simplified version of its Terraform counterpart. In fact, the plugin simply needs to create the YAML file associated with each KubernetesDeployment class, which basically translates to a deployment on an EKS cluster of a Docker image implementing an Application Component (see Section 3.3), and to add the command to deploy the YAML file to the `deployments.txt` output file.
This results in the simple and straightforward Python module **kubernetesPlugin.py**. The simplicity of the plugin is also due to a design choice of using a single template for all of the Kubernetes resources described by the KubernetesDeployment element, this choice is better explained in Section 3.4.5

As previously explained in Sections 3.4.1 and 3.4.2, the plugin is called by the Controller whenever the corresponding step from the IR is processed. So for this new plugin to be called, we have also had to modify the ICG Parser. The modifications resulted in the creation of a new function, that is called when the Parser recognizes a KubernetesDeployment or ExtenralKubernetesDeployment element in the DOML model. The function generates an IR step for the "Kubernetes" language, with the data needed to generate the YAML files. The function is reported below:

```
def parse_k8s_deployment(deployment, infra_object_step):
    # prepare ds
    application_object_step = {"programming_language": "kubernetes", "data":
        {}}
    # extract component name
    application_resource = deployment.eGet("component")
    deployment_component_name = application_resource.eGet("name")
    logging.info(f'Parsing kubernetes deployment for component
        {deployment_component_name}')
```

```
    object_representation = {}

    # assign name
    object_representation['name'] = deployment_component_name
    # extract and assign image and ports
    image = deployment.eGet("image")
    ports = image.eGet("exposedPort")
    port_list = []
    for port in list(ports):
         port_list.append(DomlParserUtilities.save_attributes(port,{}))
    object_representation['image'] =
        DomlParserUtilities.save_attributes(image, {})
    object_representation['image']['ports'] = port_list

    # extract and assign replicas and namespace
    object_representation['replicas'] = deployment.eGet('replicas')
    object_representation['namespace'] = deployment.eGet('namespace')

    # extract and assign autoscaler
    try:
        object_representation['autoscaler'] =
            DomlParserUtilities.save_attributes(deployment.eGet("autoscaler"),
            {})
    except:
        pass

    # fill ds
    application_object_step["data"] = {"KubernetesDeployment" :
        object_representation}
    application_object_step["data"]["cluster_name"] =
        deployment.eGet("cluster").eGet('name')
    application_object_step["step_name"] = deployment_component_name
    application_object_step["step_type"] = "component"

    return application_object_step

def parse_k8s_ext_deployment(deployment, infra_object_step):
    application_object_step = {"programming_language": "kubernetes", "data":
        {}}
```

```
    application_resource = deployment.eGet("component")
    deployment_component_name = application_resource.eGet("name")
    logging.info(f'Parsing extenral deployment for component
        {deployment_component_name}')
    object_representation = {}
    object_representation['url'] = deployment.eGet("url").split(',')

    application_object_step["data"] = {"ExternalKubernetesDeployment" :
        object_representation}
    application_object_step["data"]["cluster_name"] =
        deployment.eGet("cluster").eGet('name')
    application_object_step["step_name"] = deployment_component_name
    application_object_step["step_type"] = "component"

    return application_object_step
```

. The choice of calling the step "Kubernetes" and not "YAML" may seem counterintuitive, since the target language is YAML. But this choice was made to leave flexibility for future developments of the ICG, in case the creation of a general-purpose YAML plugin was ever needed.

The final task was adding the name of the plugin to the `external-plugins.properties` file, which acts as the source of truth for external plugins in the ICG, much like the `template-location.properties` file for the templates.

3.4.5. Kubernetes Resources templates

The final step was that of creating the Jinja2 template for the resources that can be deployed on Kubernetes. The two possible routes were: creating a single template for the three resources supported, namely Pods and Deployments, Services, and Horizontal Pod Autoscalers; and creating separate templates for each type of resource. We opted for the first solution because it was the simplest to manage from the point of view of the Kubernetes plugin. On the other hand, this required the use of more conditional logic in the template to discern between cases in which not all three Kubernetes resources were present in the deployment. The resulting template can be found on GitHub: **deployment.tpl**.
The conditional structures of this template make sure that the Service part of the template is present in the generated YAML file only if there are services attached to the pod.

This is also true for the HPA section.

The template was added to the newly created folder `templates/yml/kubernetes`. Just like for the EKS template, we had to update the `template-location.properties` file.

Bug Fix

Once the ICG was extended to support the new DOML-K8s elements, we noticed that some preexisting bugs in the ICG were preventing us from correctly using these new elements. Most of the work was directed toward modifying the network elements template for aws. The existing template did not correctly parse the subnet information into the final Terraform code, perhaps due to version drift between the current subnet AIL element in the metamodel and a previous version. The template also did not support the addition of an internet gateway and the routing tables, meaning there was no way to define a public network within AWS using DOML. Since this issue would prevent us from testing DOML-K8s applications, we worked on modifying the existing template, resulting in an updated version of the template that supports the definition of routing tables, associations between a table and a subntet, and definition of an internet gateway. The updated version of the network template can also be found on GitHub: **network.tpl**.

3.4.6. Putting it all together

To summarize the work done regarding the extension of the ICG. We started by creating the Terraform EKS plugin in Section 3.4.3, then adding it to the ICG template repository in the folder `templates/terraform/aws` and updating the `template-location.properties` file. We then moved to Section 3.4.4 in which we created the Kubernetes plugin to generate the YAML files for the deployment, and also modified the Parser to add the IR step related to the new plugin. For the ICG Controller to recognize the new plugin, we added it to the `external-plugins.properties` file. Finally, we created the template for the YAML deployment file in Section 3.4.5, this required the creation of the new folder `templates/yml/kubernetes` where the template is located, and the update of the `template-location.properties` file.

3.5. Summary

Chapter 2 described the requirements and objectives of the integration of Kubernetes in the DOML, analyzed the architecture and features of Kubernetes, and presented the language structure of the DOML. This Chapter builds on top of the previous one and

presents the integration of EKS in the DOML in a very practical-oriented fashion.
The integration of new concepts into the language is a long and complex process. The main phases of the integration can be summarized as follows: analysis and design of new DOML classes, metamodel extension, and ICG extension.

After an analysis of the EKS service, we determined the need to create various new DOML classes, some modeling the EKS infrastructure, and some mapping the application components to the infrastructure via Docker images implementing that component.
As a result, eight new classes were added to the DOML. While six of them were not represented in any way by the current metamodel and had to be created as an extension of the External Element classes, the KubernetesDeployment class and ExternalKubernetesDeployment class are extensions of the Deployment class from the Common Layer.

The extension of the ICG was the longest task in terms of time and preparation. The support for the EKS resource was added by means of a new template for the AWS provider. The integration of the parsing and generation of Kubernetes deployment was much lengthier. First requiring the creation of a whole new plugin, and then modifications to the ICG Parser and the Intermediate Representation. Finally, a new template for the deployment was also created.

The result of this integration is the ability to define Kubernetes applications using DOML-K8s and to deploy the application IaC generated by the ICG on AWS. The following Chapter will then explore DOML-K8s' functionalities and validate them against the requirements proposed in Section 2.5.

4 | Validation

In this Chapter we use the DOML-K8s resources described in Chapter 3 and validate them. The main objective of the following Sections is to build complete cloud applications using Kubernetes through DOML-K8s. The applications will then be used to validate the requirements proposed in Section 2.5 and summarized in Table 4.1.

Requirement	**Description**
R1	DOML can be extended using DOML-E
R2	DOML-K8s must allow the creation and usage of EKS
R3	DOML-K8s must support the creation of a heterogeneous multi-container Kubernetes application
R4	DOML-K8s must support the integration of a Kubernetes cluster within a larger application
R5	DOML-K8s must support the autoscaling of Kubernetes resources based on the workload

Table 4.1: DOML-K8s Requirements

The validation was carried out through real world use cases, cloud application using Kubernetes that reflect how these technologies are used in the industry.
For the first use case, we decided on something simple yet almost ubiquitous in modern cloud applications: a web server implemented on Kubernetes. Web servers that host web pages are the entry point for most consumer and business applications, offering access to the application through a browser. Since the traffic for the website must come through the web server, we must ensure high availability of the infrastructure. This can be achieved

using Kubernetes leveraging its strong replication and failure detection systems. We can deploy multiple replicas of the web server in order to handle load and Kubernetes would manage routing and load balancing between the replicas. Kubernetes also checks the status of each replica and restarts failed replicas. A step further can be made by adding autoscaling of replicas based on load, so that Kubernetes create new replicas of the web server when it is being overloaded with request, thus increasing the availability. With this use case we aim to validate requirements R1, R2 and R5, by showing how we can define and deploy a Kubernetes application using DOML-K8s, and how this application supports autoscaling.
The second use case is complementary to the first one, as it is used to validate requirements R3 and R4 by deploying a complex multi-container application on the same cluster as the web server, and have the two applications interact with one another. In particular, we decided on creating a distributed web scraping application using Selenium Grid. With Selenium, we are able to programmatically read websites content and process it as needed, all of this while ensuring high availability and parallelism by distributing tasks across various nodes. The Grid can be implemented in Kubernetes, thus scaling the number of nodes in the application according to the workload and bringing down overall execution time. By implementing the Grid in Kubernetes and having it scrape information from the web server of use case one, we validate the interaction between Kubernetes application and the overall inclusion of Kuberntes into a larger complex application. The choice of implementing the Grid in Kuberentes also serves to validate the creation of complex applications comprising multiple components. In fact, the grid itself is made up of two separate containers, one for the hub and one for the nodes (this architecture is explained more thoroughly in Section 4.2.1), as well as multiple external and internal deployments that are used to support austocaling of the nodes based on load.

Based on the two validation use cases above, we can provide a summary of how we intend to validate each requirement in the following table:

Requirement	Valiadtion Method
R1	Show the DOML-K8s model
R2	Show the ICG code generation result from DOML-K8s models, and the application created using the IaC
R3	Show the definition and deployment of an application with multiple containers from different images
R4	Show that different Kubernetes applications can easily interact with one another
R5	Show that a DOML-K8s application supports autoscaling of pods based on load

Table 4.2: Requirements Validation

4.1. Scalable Web Server

The first use case for Kubernetes has been designed to validate the requirements R1, R2, and R5. For that purpose we have chosen to implement an Nginx Web Server, serving a simple dynamic website that presents the user with a number that is randomly generated every time the page is loaded. The Nginx server and backend are built into a single Docker image and deployed on the cluster, this image implements our Scalable Web Server. The choice of using a single image instead of two separate ones, one for the Nginx server and one for the backend, was made for the sake of simplicity. The main goal of this use case is to validate requirements R1, R2, and R5, the implementation of a multi-container Kubernetes application will be tackled in a further use case. The EKS cluster makes the website reachable from the internet and supports autoscaling of the web server based on load. For the sake of brevity, for the rest of the Thesis, we will refer to the Scalable Web Server as SWS. Unless explicitly stated, all of the code we refer to below can be found on GitHub [1].

[1]Repository: ¸

4.1.1. Nginx Web Server

As previously stated, both the Nginx server and website backend are implemented through a single Docker image, which is uploaded in the Docker registry and pulled from EKS when the container has to be instantiated.

In order to build the SWS image we first designed the Nginx config file that is used to serve the website. The config file must redirect queries for the "/" resource to "localhost:5001", the address where the Python backend is running.
The backend is a simple Flask application, listening on port 5001, that returns an HTML page with a randomly generated number.
To run both the Nginx server and the backend in the same container we used the supervisor package, a tool designed to automate tasks. Supervisor takes care of starting the Nginx server as well as the Python application.
The backend and Nginx server are finally put together into a Docker image using a Dockerfile. The Dockerfile will copy the previously described files, move the Nginx configuration to the correct folder, expose port 80 so that the website is reachable, and finally run supervisord to start the Nginx web server and the backend. The generated Docker image was then uploaded to Docker Hub so that it could be retrieved from inside EKS.

4.1.2. Scalable Web Server in EKS

Now that the image implementing the web server has been completed, we move on to defining the Kubernetes resources that must be deployed on the cluster.

The first obvious step is creating a deployment for the SWS image that spawns one or more replicas of a pod from the SWS image. This creates a set of replicated containers, generated from the SWS image, that are ready to serve the website. In order to expose the website to the internet, we must then attach a LoadBalancer service to the deployment. These two resources alone are sufficient to build a dynamic website, proving once more the effectiveness of Kubernetes.
What is missing now is the autoscaling part. This can be done by attaching an HPA to the deployment, which scales based on CPU utilization. We are then able to stress test the website and cause Kubernetes to instantiate more pods. We should note that to allow the HPA to read the utilization metrics from the pods, we must also deploy on the cluster the external components "metric-server". The metric-server is a Kubernetes component that reads the resource utilization metrics from every pod on the cluster and makes them available to the HPAs. An HPA can then read metrics regarding its target pod and perform scaling actions as needed. In this specific case, there is no need to deploy the

metric-server since AWS considers it an important building block for an EKS application, so much so to add it to the list of official EKS add-ons, a list of third-party Kubernetes resources deemed to be essential for EKS applications. The add-on is by default enabled in DOML-K8s, as can be seen from the EKS template in Section 3.4.3.

4.1.3. DOML-K8s implementation

Now that the design process has been completed, we can move on to writing the DOML-K8s model that describes the SWS system.

DOML-K8s model

The different layers of the model contain the following components:

- The AL layer contains the SWS Software Component that describes the entire SWS. It is notable that if the Nginx server and backend were implemented through two separate images, we should have created two separate application components.
- The AIL contains definitions for: the network, the subnets, the security group, the cluster and its nodes, the SWS image, and the autoscaler.
- The CIL contains the mappings to the infrastructural elements from the AIL: the network and the cluster.
- The configuration section contains the deployment definition for the SWS, which maps the SWS Software Component to the cluster through the image and attaches the autoscaler.

A diagram of the DOML-K8s model is shown in the Figure below.

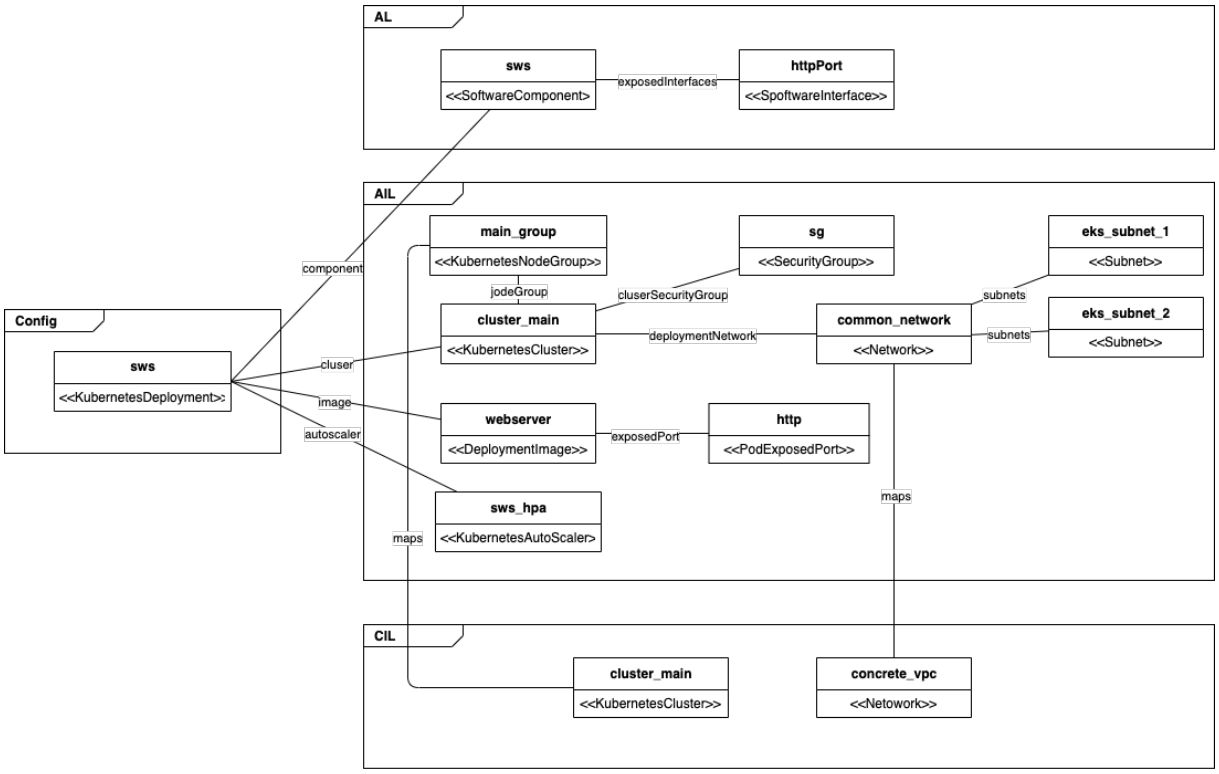

Figure 4.1: SWS model diagram

The model is then parsed by the ICG and generates the IaC files. The ICG generates both a Terraform file and a Kubernetes YAML file. The files can be found in Appendix A

The generated Terraform file contains the definition of the AWS networking resources, Security Group, and EKS cluster. We can run `Terraform apply` to create the infrastructure in our AWS project.

The Kubernetes YAML file defines the deployment, the service, and finally the autoscaler for the SWS. We can then follow the commands from the `deployments.txt` file to deploy all of the resources to EKS.

4.1.4. Validation Results

Once all of the resources have been created, we can query the EKS cluster through the `kubectl` utility to get all of the resources that have been created.

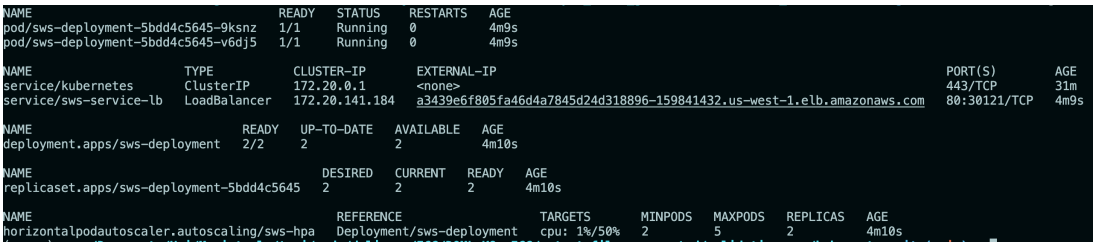

Figure 4.2: SWS resources

As we can see in Figure 4.2, the IaC generated from the ICG caused the creation of two pods for the SWS and an attached LoadBalancer service, as well as an HPA. We can now reach the webpage at the service IP.

Figure 4.3: SWS webpage

By connecting to the IP through HTTP we can view the SWS webpage, as shown in Figure 4.3. This IP is not static and will change every time the resource is restarted. This is fine for testing purposes, but in a production environment, we should define a static IP for the website, attach it to the service, and also create a DNS entry that associates the static IP to a public domain.

The final validation step is the evaluation of the autoscaling properties. To do so we can use a tool to generate HTTP traffic from our local machine, and direct that traffic to the website IP.

```
hey -z 1m -c 50 <url>
```

This command will issue HTTP requests to the endpoint for 1 minute using 50 concurrent workers. We can then inspect the state of the HPA and the number of pods for the SWS deployment.

Figure 4.4: HPA utilization

```
NAME                                READY   STATUS              RESTARTS   AGE
sws-deployment-5bdd4c5645-9ksnz     1/1     Running             0          7m3s
sws-deployment-5bdd4c5645-9lvbx     1/1     Running             0          16s
sws-deployment-5bdd4c5645-nwhdx     1/1     Running             0          16s
sws-deployment-5bdd4c5645-psfn8     0/1     ContainerCreating   0          1s
sws-deployment-5bdd4c5645-v6dj5     1/1     Running             0          7m3s
```

Figure 4.5: SWS scaled resources

As we can see from Figures 4.4 and 4.5, the HPA detects a utilization that exceeds the threshold, as a result, the pods have scaled up to 5 replicas, the limit set in the HPA yaml definition.

This use case has thus correctly validated the initially proposed requirements. R1 and R2 have been validated by showing how the DOML-K8s model can correctly generate a Kubernetes application. R5 has been validated by showing the effectiveness of the autoscaling process by means of the HPA.

4.2. Web Scraping with Selenium Grid

The second use case has been designed to validate all of the requirements described at the beginning of the Chapter. It consists of a distributed web scraping application that can be used to read large amounts of data from public websites. In particular, we decided to scrape the random number generated by the SWS created in Secion 4.1. The intent is to build a large project, that uses Kubernetes to deploy different applications, interconnected with one another.

4.2.1. Selenium Grid

Selenium is an open source web scraping tool, that can be used to read data from public websites. Selenium Grid is a distributed implementation of Selenium that can be easily implemented using Kubernetes. A Grid contains one Selenium Hub and one or more Selenium nodes.

The **Selenium-hub** is the master node and entry point of the Selenium application. A scraping job is started by connecting to the hub through HTTP (using the Selenium library or raw HTTP calls) and submitting a job. A job consists of a function that requests data from a webpage by providing its URL and then reads the content of the HTML. Upon completion, the hub returns from the call with the return value of the job function. The hub also offers other features, such as a web UI and a set of APIs that can be used to obtain information about the state of the Grid and the current jobs.

The **Selenium node** is a worker node of a Selenium Grid, the node receives jobs from the hub and executes them. When executing the job function the node will use one of many browsers (e.g. Chrome, Firefox, or Edge) to get the webpage, process its content, and return the job return value.

Due to the distributed nature of the Grid, Kubernetes is particularly useful thanks to its ability to easily create replicas of the nodes, thus expanding the capabilities of the Grid and processing multiple jobs in parallel. For our particular use case, we want to deploy an initially small Grid and submit a large number of jobs to scrape the randomly generated number from the SWS. Then, we want to scale the number of nodes according to the number of queued jobs in the hub.

4.2.2. Implementation in EKS

To implement the Grid in Kubernetes, we first need to create deployments for both the Selenium Hub and Selenium Nodes. We also need to attach a LoadBalancer service to the hub to expose its connections to the outside, as well as to the nodes so that they can join the Grid. We are then able to run a Python script that connects to the Hub on its public endpoint and submits a large number of jobs. All of the code we refer to below can be found on GitHub: **Web Scraper repo**.
The Python program scraper.py uses multithreading to spawn a multitude of Selenium jobs. Each thread uses the selenium library to connect to the hub URL and submits a job that loads the webpage from the SWS and extracts the generated number. The number returned from each thread is then added to a list, the program then prints the average of the generated numbers. One notable step that is made much easier by Kubernetes is the ability to dynamically resolve the address of the SWS "http://sws-service-lb.default.svc.cluster.local:80" from inside the cluster, so the program does not need to be changed every time the SWS is restarted and its address changes. While in a real deployed application, we would assign a static IP to the SWS LoadBalancer and a human-readable address associated, during the development phase this feature is incredibly useful and once again proves the effectiveness of using Kubernetes. This is also true for the Hub IP, which in production would be static and protected by very restrictive access policies.

Creating and using the grid is simple enough, what is significantly harder is scaling the number of nodes according to the queue size. We could easily implement a scaling logic akin to the SWS, where we monitor the CPU utilization of the nodes and scale them up when it is too high. However, this solution is not ideal in the case of multiple simple jobs, like the use case under consideration. Due to the large number of jobs we have low node

utilization and a large number of queued jobs. The better solution for decreasing overall task completion time is to scale up the nodes when we have too many jobs in the queue. This solution is complex and requires the use of external metrics, which in Kubernetes means using various Kubernetes resources as well as some CRs (Custom Resources were previously discussed in Section 2.2.6):

- A metrics server CR, similar to the one described in Section 4.1.2, that reads various scaling metrics from other resources in the cluster, and makes it available to the HPAs.
- A pod that reads the queue size metric from the Selenium Hub and makes it public to the cluster through an endpoint.
- A CR HPA that can read the queue metric from the metric server and scale the nodes accordingly.

After careful consideration, we decided to run with the following Kubernetes resource stack: Prometheus for metric collection, a custom Python metric exporter, and the Keda CRs to scale on the custom metric.

We used the Prometheus server to expose the metrics. Prometheus is a third-party Kubernetes tool that collects metrics from various endpoints and makes them available to the cluster through API, and to the user through a web UI. The endpoints must export the metrics in a Prometheus-compatible format. To deploy the Prometheus server, we used an external yaml file that deploys various Prometheus CRs, as well as 1 replica of the Prometheus server, and exposes a LoadBalancer for it.

We then built a custom metric exporter that exports the selenium_queue_size metric. The Docker image for the exporter simply runs a Python application that exposes the "/metrics" endpoint. When called, the application queries the Selenium Hub for its queue length and returns the obtained value. The image can then be deployed through a simple deployment and attached to a ClusterIP service so that it can be queried by Prometheus.

In order to tell Prometheus to query the metric exporter at the "/metrics" endpoint, we must once again use an external yaml file. The file config.yaml creates a configuration for Prometheus, telling it to read metrics from the metric exporter.
We also need to create a ServiceMonitor resource that is used by Prometheus to fetch the metric. The ServiceMonitor is a CR introduced by the Prometheus CRD.

Finally, we must create the Keda ScaledObject resource, a CR version of the Kubernetes HPA. This requires first deploying the Keda manifest to use its CRDs. The manifest can

be found on the official Keda documentation [2]. Applying this manifest creates all of the CR necessary to use Keda ScaledObjects. So the final step is to deploy the ScaledObject, creating an HPA that can scale up the Grid nodes when the queue exceeds a length of 5.

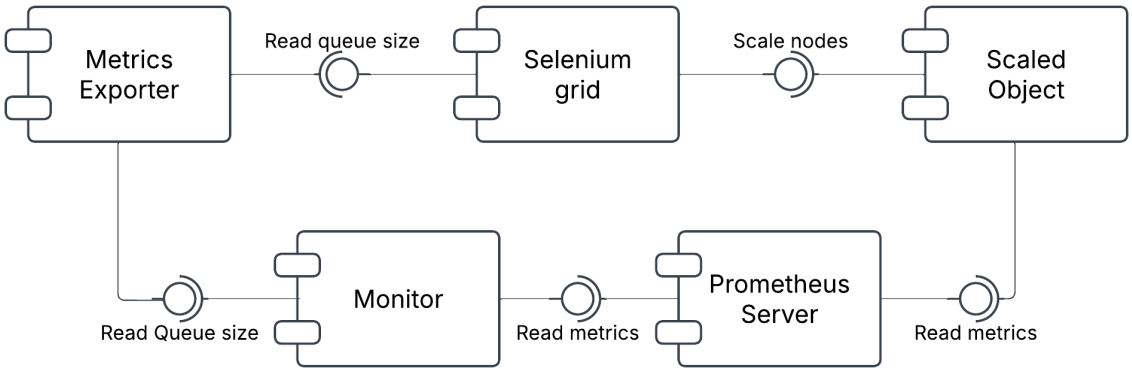

Figure 4.6: Selenium Grid Scaling Flow

The full interaction between the Grid, Prometheus server, metric exporter, and Keda scaler is pictured in Figure 4.6. As we can see the Grid offers a Read queue size interface, and the exporter reads the queue_size metric from it. The Monitor then uses the Read queue size provided by the Metrics Exporter to read the metric. The Prometheus server reads, through the Monitor's Read metrics interface, the queue size and makes it available to the other components through the provided Read metrics interface. The Keda scaler finally reads the metric from the Prometheus Server and scales the Grid nodes accordingly, using the Scale nodes interface provided by the grid..

4.2.3. DOML-K8s implementation

Once the application design has been completed, we can write the DOML-K8s model that describes the Selenium Grid system.

DOML-K8s model

Since this use case validates requirements R4 and R3, which revolve around the integration of multiple applications using Kubernetes, the Grid should be deployed on the same cluster used in the previous use case for the SWS. The only change we decided to perform cluster-wise is the addition of a second node group to better handle the load and further validate

[2]For this Thesis, we used Keda version 2.13.0, which can be obtained from `https://github.com/kedacore/keda/releases/download/v2.13.0/keda-2.13.0.yaml`

R1. With respect to the SWS model described in Section 4.1.3 the following components were added:

- The AL layer contains the selenium Software Components that describe the web scraper. We have a component for the hub, the nodes, the metric exporter, and finally the scaler. The hub offers an HTTP interface that is consumed by all the other components.
- The AIL contains new definitions for the images used by the hub and nodes, as well as the image for the queue metric-exporter. Here we also add a second node group to the cluster, in order to better handle the new load.
- The CIL adds no classes compared to the previous use case since we are reusing the same VPC and EKS cluster.
- The configuration section contains multiple new deployment definitions: the Selenium hub and Selenium nodes deployment, the metric-exporter deployment, and the external deployments for Prometheus, the ServiceMonitor, the ConfigMap, the Keda CRD, and the ScaledObject.

A diagram of the DOML-K8s model is shown in Figure 4.7.

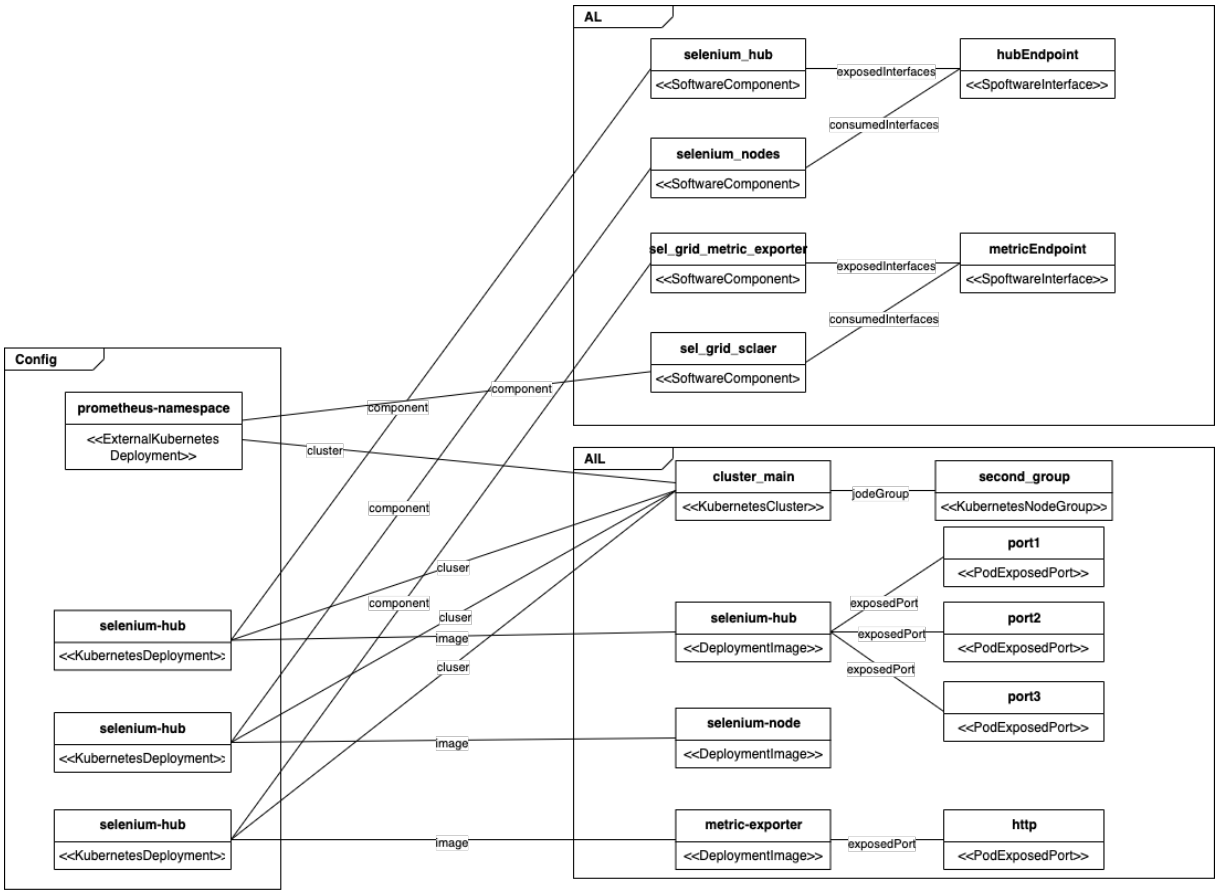

Figure 4.7: Selenium Grid model diagram

For the sake of presentation, we avoided showing the elements already shown in the previous use case, they can be seen in Figure 4.1. We also show a single ExternalKubernetesDeployment, since all of the others share the same relations with the same classes. This model is then parsed by the ICG and generates the IaC files.

The Terraform file is unchanged from the previous use case, apart from the addition of a second node group. Three separate Kubernetes YAML files are created by the ICG. Two define the Selenium Hub deployment with its service and the Selenium node deployment. The files can be found in Appendix B. Notably, the selenium node image is not the standard image, but rather a custom image that adds to the original one the environment variables useful for the node to reach the hub in the cluster.
The final YAML file, is the one used to deploy the metric-exporter and its ClusterIP service.
We can then follow the commands from the `deployments.txt` file to deploy all of the resources to EKS.

4.2.4. Validation Results

Once all of the resources have been created, we can query the EKS cluster through the `kubectl` utility to get all of the resources that have been created.

```
NAME                                                    READY   STATUS    RESTARTS   AGE
pod/selenium-grid-metric-exporter-deployment-56dd477d79-shgrn   1/1     Running   0          23m
pod/selenium-hub-deployment-f75fc788f-c98pp             1/1     Running   0          38m
pod/selenium-nodes-deployment-59b4c58775-4qt4f          1/1     Running   0          26m
pod/selenium-nodes-deployment-59b4c58775-xmwmr          1/1     Running   0          26m

NAME                                          TYPE           CLUSTER-IP       EXTERNAL-IP                                                              PORT(S)
                    AGE
service/selenium-grid-metric-exporter-service-ip   ClusterIP      172.20.130.187   <none>                                                                   8080/TCP
                    23m
service/selenium-hub-service-lb               LoadBalancer   172.20.19.236    a84dd17475aa94893b283ebb8af46231-90347633.us-west-1.elb.amazonaws.com    4442:32102/TCP,4443:30
886/TCP,4444:31466/TCP   38m

NAME                                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/selenium-grid-metric-exporter-deployment   1/1     1            1           23m
deployment.apps/selenium-hub-deployment             1/1     1            1           38m
deployment.apps/selenium-nodes-deployment           2/2     2            2           26m

NAME                                                   DESIRED   CURRENT   READY   AGE
replicaset.apps/selenium-grid-metric-exporter-deployment-56dd477d79   1         1         1       23m
replicaset.apps/selenium-hub-deployment-f75fc788f      1         1         1       38m
replicaset.apps/selenium-nodes-deployment-59b4c58775   2         2         2       26m

NAME                                                REFERENCE                                 TARGETS     MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/keda-hpa-selenium-hpa   Deployment/selenium-nodes-deployment   0/5 (avg)   1         100       2          2m57s
```

Figure 4.8: Selenium resources

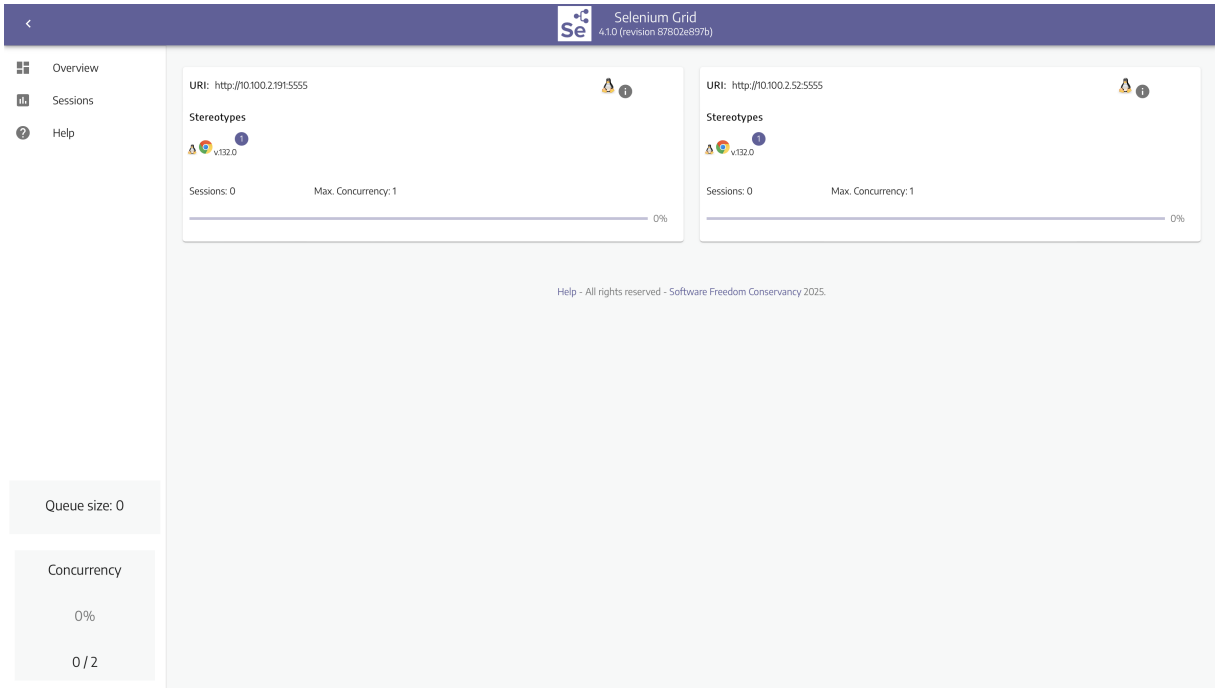

Figure 4.9: Selenium Grid UI

The IaC has been used to create the pods for the Selenium Hub and Nodes, the Hub service, the Keda HPA, and the metric-exporter pod and service. All of them belong to the Selenium namespace and can be seen in Figure 4.8. We can verify that the grid is working by connecting to the hub UI, which can be seen in Figure 4.9

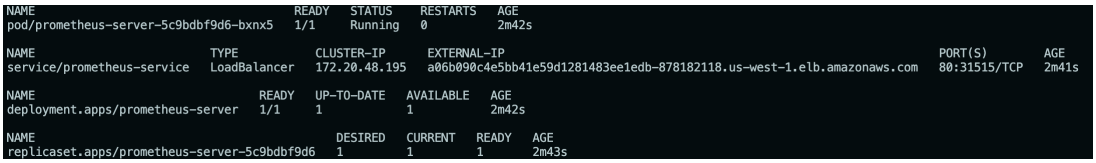

Figure 4.10: Prometheus resources

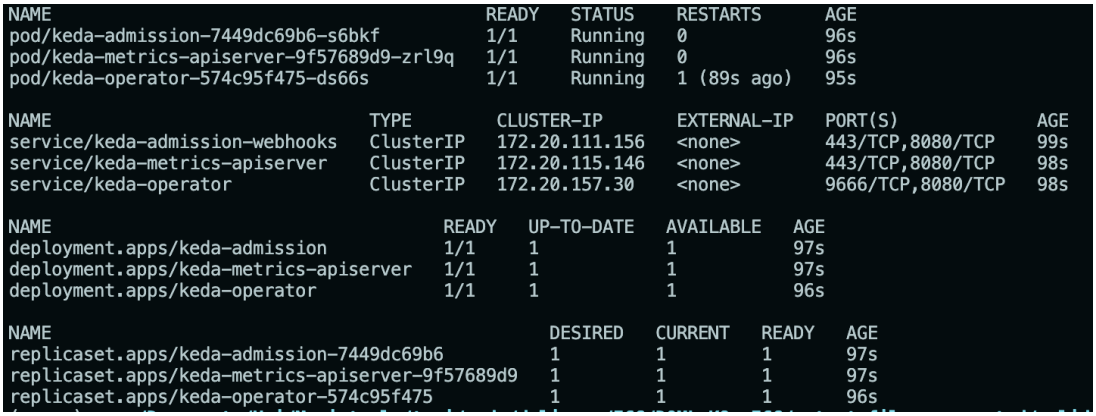

Figure 4.11: Keda resources

As a result of the use of the external deployment, we have also generated new resources in the Keda and Prometheus namespaces. The resources are shown in Figures 4.10 and 4.11.

We are now able to plug the Selenium Hub IP into the scaper program from the beginning of Section 4.2.2 to run multiple jobs. The code runs correctly and returns the average of the number generated by the SWS.

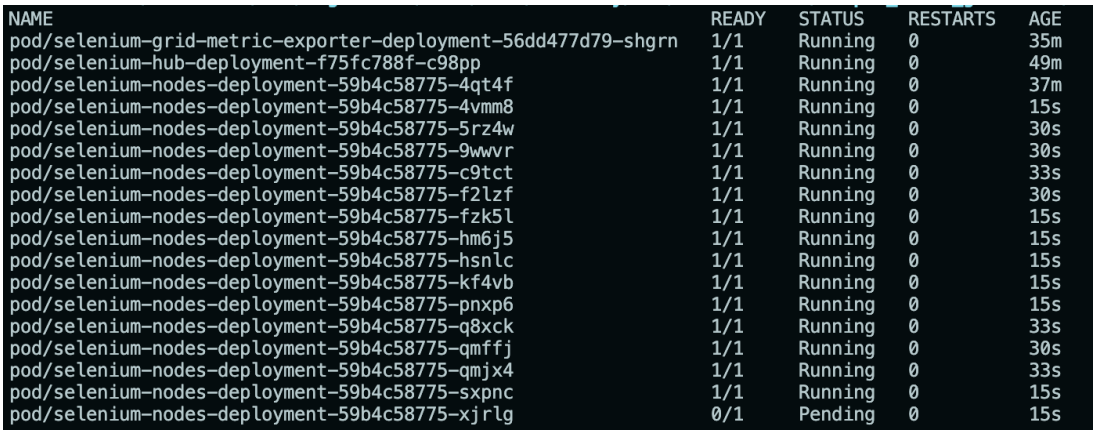

Figure 4.12: Selenium scaled resources

We can see the scaling in action by using `kubectl` to get the selenium-nodes pod. As

seen in Figure 4.12, the number of pods has now scaled up according to the number of jobs queued up.

This use case has correctly validated the initially proposed requirements. R1 and R2 have been validated by showing how the DOML-K8s model can correctly generate the Kubernetes application. R5 has also been validated, by showing the autoscaling in action in Figure 4.12. R3 and R4 have been validated by showing the interaction between multiple containers in a single application, and between the Selenium application and SWS application.

One might point out that the extensive use of external resources goes against the validation of DOML-K8s, instead exposing deficiencies in the language. However, the use of the external Kubernetes files actually proves the flexibility and ease of use of DOML-K8s. Instead of polluting the language with very specific resource definitions useful only once for very specialized applications, the language offers the possibility of using external and already existing resources very easily. This works in favor of the user, especially an inexperienced one, as it allows the reuse of existing third-party IaC. In case the resources were deemed so important and useful to need proper representation in the language, they could easily be added to the metamodel and ICG. For example, the Keda ScaledObject could be introduced as an extension of the KubernetesAutoScaler class. This use case further solidifies and validates the flexibility of the DOML project.

4.3. Conclusions

Through the use of two different but connected use cases, we have shown and validated the new features of DOML-K8s according to the requirements described at the beginning of the Chapter.

The first use case, a simple web server, proves the ease of use of DOML-K8s and Kubernetes in general. Using the DOML-K8s a user can define his application in a few lines of code, and deploy it in a matter of minutes. The application created is fully functional and also supports autoscaling, a feature whose importance cannot be overstated and has been discussed at length during this Thesis.

The second use case is a much more complex one: a fully scalable distributed web scraping platform using Selenium Grid. Once again the application has been defined in DOML-K8s very easily, and deployed in a matter of minutes. The big advantage of the language is the ability to incorporate external resources into DOML models, thus increasing flexibility, promoting code reusability reducing development time, and perhaps more importantly

supporting inexperienced users. This use case also serves to demonstrate how Kubernetes applications created using DOML-K8s can easily communicate with one another and be integrated into larger applications.

5 | Conclusion and Future developments

This Chapter draws conclusions on the work presented in this Thesis. Discusses the objectives achieved, and introduces possible future developments.

5.1. Objectives Achieved

This Thesis has concluded with the fulfillment of all of the objectives proposed in Section 2.5 and the validation of the requirements of the integration, as discussed in Section 4.3.

The first fundamental objective was the extension of DOML into DOML-K8s using the DOML-E extension mechanism. This result validates the flexibility and extensibility of the DOML by virtue of including totally new concepts in the language, and is emphasized by the nature of the new metamodel classes. While some classes were totally new to the language and as such generated as extensions of the ExternalElement classes, others like the KubernetesDeployment were already partially described in the metamodel and are thus extensions of previously existing classes. This proves the extensibility of DOML both in terms of integrating new concepts, and refining or extending existing ones. This objective was validated through requirement R1, by showing that we are able to generate a model for DOML-K8s that includes the new Kubernetes classes.
Another important achievement is the validation of the extension of the Infrastructural Code Generator. The ICG is an integral part of the PIACERE Framework, as it works in conjunction with the DOML IDE to generate the target IaC from a DOML model. The extension has once again been validated under two aspects: the extension of existing languages through the inclusion of a new template, and the introduction of a new plugin to support a new target language. The validation for this second objective comes from the validation of requirement R2, as we were able to prove that a DOML-K8s model is correctly used by the ICG to generate Infrastructure as Code that, once deployed, creates the desired application.

The new capabilities of DOML-K8s have then been tested and validated. We have shown the ability of DOML-K8s and the ICG to describe complex, multi-container, Kubernetes applications, and to generate the IaC that is used to create the EKS cluster and deploy the Kubernetes components. We are able to define and deploy applications composed of both internal and external Kubernetes components. The deployed applications work correctly and can communicate with one another. This has been validated through requirements R3 and R4 with the Web Scraping use case (see Section 4.2), in which we created an application comprising a moltitide of different containers, both internal and external to the DOML-K8s, and had it communicate with the web server.
Finally, the Kubernetes applications described in DOML-K8s can also support autoscaling natively, using the built-in Kubernetes Horizontal Pod Autoscaler, or using external scalers. This has been validated through requirement R5 in both of the use cases presented.

To summarize, the following are the main objectives achieved by this Thesis:

- The DOML has been successfully extended into the DOML-K8 using DOML-E.
- The PIACERE ICG has been suitably extended, according to the new classes in the DOML-K8s metamodel.
- The new capabilities to create an EKS cluster and deploy both internal and external Kubernetes components have been tested and validated with the two use cases presented in Chapter 4.
- The DOML-K8s has been used to create complex heterogeneous applications. The interaction between EKS applications has also been validated.
- The autoscaling capabilities of Kubernetes have also been implemented and tested, and work as desired.

The new DOML-K8s can now be used by experienced and inexperienced users alike, to describe and deploy complex cloud applications involving Kubernetes.

5.2. Future works

This Section presents future developments for DOML-K8s, such as new Kubernetes concepts that could be introduced in the language, or modifications to the current EKS definition.

5.2.1. Ingress

We have already talked about Kubernetes Services in Secion 2.2.3, and used them to access the containers deployed on our pods all throughout Chapter 4. We can expose a container to the internet through a LoadBalancer service, which assigns an IP address to a deployment and automatically manages load balancing between the pod replicas. This approach works but has a couple of downsides. First of all, using LoadBalancers an application made up of multiple different containers would have to expose each container separately, thus requiring a large number of IPs and increasing the complexity of managing routing to the application. With services like EKS each LoadBalancer service is associated with the LoadBalancer component of the cloud provider, having lots of LB services increases the overall cost of the infrastructure. Lastly, the routing policy of Kubernetes' LB services is opaque to the user, who has no power over it.

Ingress is a Kubernetes component that can be used to more efficiently manage routing to an application. An ingress works almost like a reverse proxy, it offers a single IP address that serves as the entry point of multiple Kubernetes components. The user can then define rules for routing requests from the ingress to the various containers. This addresses the limitations of LB services, trading a more complex networking definition for more precise routing and decreased costs. Ingress is also useful for managing rolling updates, for example, we can deploy a new version of a container and increasingly route more and more traffic to the new version while keeping the old one working. Ingress also comes with increased security, because all container services now become private services, exposed only through the ingress which has strict access rules.
The ingress rules route HTTP requests to the ingress to an internal service, the routing is based on the path in the HTTP request.

To support ingress into the DOML-K8s we must first choose what ingress we want to deploy. All of the ingress rules are defined through a YAML file, but in order for them to be validated and applied inside the cluster we must first deploy an ingress controller. The ingress controller is a pod in our cluster that is responsible for validating ingress rules and managing routing accordingly. There are many possible choices for an ingress controller, Kubernetes offers an official Nginx ingress controller, but there are also many third-party options.
A future integration of ingress into DOML-K8s needs to support a way to deploy an ingress controller in the cluster, and then to define ingress rules for the services.

5.2.2. Fargate Clusters

For this Thesis we have decided to deploy EKS clusters with managed node groups, meaning AWS manages the Kubernetes control plane, creates the worker nodes as EC2 VM instances, and connects them to the control plane. There is also the possibility of using Fargate instances instead of EC2, meaning using serverless instances instead of Virtual Machines.
The advantages of going serverless are mostly three:

- Simplifying node management: even though with managed node groups EKS creates and handles the EC2 instances, we are still burdened with the monitoring of the VMs which generates an operational overhead.
- Reducing cost for variable workloads: one of the main advantages of serverless over traditional VMs is that we only pay for the actual compute time, meaning in case of a workload on the cluster that is not constant we do not waste resources keeping our VMs running with low utilization.
- Improved security: serverless environments are completely isolated, in particular with EKS Fargate each pod runs in an isolated computing environment, thus increasing security.

Of course, serverless EKS also presents some downsides, namely: limited flexibility, missing support for some advanced Kubernetes features, higher cost for sustained workloads, and cold start latency.

For all of these reasons, as well as for the sake of simplicity, we have decided to only support managed node groups EKS clusters in this first version of DOML-K8s. Thanks to the high extensibility of DOML, future versions could easily support EKS Fargate clusters by introducing a Fargate profile class in the AIL, and modifying the eks terraform template.

5.2.3. CoreDNS

CoreDNS is a third-party Kubernetes component exposing the Kube-DNS ClusterIP service, it is responsible for name resolutions inside the cluster. We have implicitly used CoreDNS in Section 4.2, for instance, it was used to resolve the address of the sws load balancer service from inside the selenium node pod. By default, each pod is configured to reach CoreDNS at the Kube-DNS service address for name resolution of certain domains. The domains are: `default.svc.cluster.local`, `svc.cluster.local`, and `cluster.local`; we are thus able to use these domains inside our pods to reach other ser-

vices in the cluster. CoreDNS of course contains a record for each service in the cluster. Being able to resolve only internal addresses is certainly limiting for most if not all real-world applications, for this reason, CoreDNS can also resolve public domains by forwarding the name resolution request to external DNS servers. The configuration of the external DNS is done through a config map for CoreDNS. By default, CoreDNS uses the public DNS infrastructure to resolve external names, most commonly by forwarding to Google DNS.

Future versions of DOML-K8s could natively support the addition of DNS records inside CoreDNS. This could be achieved by introducing a new DNS Config Map AIL element that can be mapped to a cluster. This addition would result in more flexibility in the configuration of DNS and increased security, for example redirecting to a specific DNS server instead of a public one.

Bibliography

[1] D. Benedetto, L. Blasi, and L. Niculut. D3.6 infrastructural code generation, aug 2023.

[2] S. Canzonieri. Exploring extensibility of devsecops modelling language (doml). Master's thesis, Politecnico di Milano, 2024. https://hdl.handle.net/10589/223937.

[3] M. Chiari, B. Xiang, S. Canzoneri, G. N. Nedeltcheva, E. Di Nitto, L. Blasi, D. Benedetto, L. Niculut, and I. Škof. Doml: A new modeling approach to infrastructure-as-code. *Information Systems*, 125:102422, 2024. ISSN 0306-4379. doi: https://doi.org/10.1016/j.is.2024.102422. URL https://www.sciencedirect.com/science/article/pii/S0306437924000802.

[4] Cloud Computing: Pricing Models. https://www.binadox.com/blog/cloud-computing-pricing-models/.

[5] T. D. A comparative study of resource management approaches in kubernetes and docker swarm: Efficiency and scalability. *Bulletin of Science and Practice*, 10:140–145, 11 2024. doi: 10.33619/2414-2948/108/18.

[6] L. R. de Carvalho and A. Patricia Favacho de Araujo. Performance comparison of terraform and cloudify as multicloud orchestrators. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 380–389, May 2020. doi: 10.1109/CCGrid49817.2020.00-55.

[7] DevOps lifecyle. https://basanagouda.medium.com/day-1-getting-started-with-devops-b78ccca0af8.

[8] E. Di Nitto, J. Gorroñogoitia Cruz, I. Kumara, D. Radolović, K. Tokmakov, and Z. Vasileiou. *Deployment and operation of complex software in heterogeneous execution environments: the SODALITE approach*. Springer Nature, 2022.

[9] Docker website. https://www.docker.com.

[10] DOML specification. https://git.code.tecnalia.com/piacere/public/

the-platform/doml/-/blob/70dfc1b30a1e2131df3e1eb119654a6b5880077f/specification/DOML_Specification_v3.1.pdf.

[11] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano. Devops. *IEEE Software*, 33: 94–100, 05 2016. doi: 10.1109/MS.2016.68.

[12] F. Fondement and R. Silaghi. Defining model driven engineering processes. 01 2004.

[13] S. L. Garfinkel and H. Abelson. *Architects of the Information Society: Thirty-Five Years of the Laboratory for Computer Science at MIT*. The MIT Press, 04 1999. ISBN 9780262266710. doi: 10.7551/mitpress/1341.001.0001. URL https://doi.org/10.7551/mitpress/1341.001.0001.

[14] L. Gerber. Containerization for hpc in the cloud: Docker vs singularity - a comparative performance benchmark, 2018. https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1277794&dswid=7933.

[15] Getting Started with Kubernetes. https://dzone.com/storage/attachments/14131598-dzone-kubernetes-bundle.pdf.

[16] J. Harry Katzan. The education value of cloud computing. *Contemporary Issues In Education Research*, 2010.

[17] High Performance Computing. https://www.ibm.com/topics/hpc.

[18] A. S. Ian Miell. *Docker in Practice, Second Edition*. Manning, 2019.

[19] A. Khan. Key characteristics of a container orchestration platform to enable a modern application. *IEEE Cloud Computing*, 4(5):42–48, Sep. 2017. ISSN 2325-6095. doi: 10.1109/MCC.2017.4250933.

[20] Kubernetes — Architecture Overview. https://medium.com/devops-mojo/kubernetes-architecture-overview-introduction-to-k8s-architecture-and-understanding

[21] G. M. Kurtzer, V. Sochat, and M. W. Bauer. Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5):1–20, 05 2017. doi: 10.1371/journal.pone.0177459. URL https://doi.org/10.1371/journal.pone.0177459.

[22] C. Lekkala. Deploying and managing containerized data workloads on amazon eks. *Journal of Artificial Intelligence and Cloud Computing*, 2(2):1–5, 12 2023. doi: 10.2139/ssrn.4908416. URL https://ssrn.com/abstract=4908416.

[23] M. Luksa. *Kubernetes in Action*. Manning Publications, 2017.

[24] A. I. Martyshkin and R. A. Biktashev. Research and analysis of computing cluster

configuration management systems. In A. A. Radionov and V. R. Gasiyarov, editors, *Advances in Automation IV*, pages 194–205, Cham, 2023. Springer International Publishing. ISBN 978-3-031-22311-2.

[25] K. Morris. *Infrastructure as Code*. O'Reilly media inc., 2021.

[26] N. Naik. Performance evaluation of distributed systems in multiple clouds using docker swarm. In *2021 IEEE International Systems Conference (SysCon)*, pages 1–6, April 2021. doi: 10.1109/SysCon48628.2021.9447123.

[27] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi. Cloud container technologies: A state-of-the-art review. *IEEE Transactions on Cloud Computing*, 7(3):677–692, July 2019. ISSN 2168-7161. doi: 10.1109/TCC.2017.2702586.

[28] Piacere Framework Architecture, . `https://piacere-project.eu/blog/piacere-2-0-framework-architecture/`.

[29] Piacere Framework Architecture, . `https://piacere-project.eu/blog/framework-architecture-of-piacere/`.

[30] A. Pirrotta. DOML-K8s ICG, Feb. 2025.

[31] M. Saraswat and R. Tripathi. Cloud computing: Comparison and analysis of cloud service providers-aws, microsoft and google. In *2020 9th International Conference System Modeling and Advancement in Research Trends (SMART)*, pages 281–285, Dec 2020. doi: 10.1109/SMART50582.2020.9337100.

[32] M. Shahin, M. Ali Babar, and L. Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017. ISSN 2169-3536. doi: 10.1109/ACCESS.2017.2685629.

[33] N. K. Singh, S. Thakur, H. Chaurasiya, and H. Nagdev. Automated provisioning of application in iaas cloud using ansible configuration management. In *2015 1st International Conference on Next Generation Computing Technologies (NGCT)*, pages 81–85, Sep. 2015. doi: 10.1109/NGCT.2015.7375087.

[34] S. Singh and I. Chana. Cloud resource provisioning: survey, status and future research directions. *Knowledge and Information Systems*, 49(3):1005–1069, Dec 2016. ISSN 0219-3116. doi: 10.1007/s10115-016-0922-3. URL `https://doi.org/10.1007/s10115-016-0922-3`.

[35] J. Surbiryala and C. Rong. Cloud computing: History and overview. In *2019 IEEE Cloud Summit*, pages 1–7, Aug 2019. doi: 10.1109/CloudSummit47114.2019.00007.

[36] Tecnalia. Piacere icg, 2014.

[37] Terraform website. https://www.terraform.io.

[38] Traditional and cloud datacenters. https://www.parkplacetechnologies.com/blog/traditional-data-center-vs-virtualization-differences/.

[39] What is Ansible and What Does it Do? https://www.liquidweb.com/blog/what-is-ansible/.

[40] M. Wurster, U. Breitenbücher, A. Brogi, G. Falazi, L. Harzenetter, F. Leymann, J. Soldani, and V. Yussupov. The edmm modeling and transformation system. In *Service-Oriented Computing–ICSOC 2019 Workshops: WESOACS, ASOCA, ISYCC, TBCE, and STRAPS, Toulouse, France, October 28–31, 2019, Revised Selected Papers 17*, pages 294–298. Springer, 2020.

[41] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, and J. Soldani. The essential deployment metamodel: a systematic review of deployment automation technologies. *SICS Software-Intensive Cyber-Physical Systems*, 35: 63–75, 2020.

[42] D. R. V. Wågbrant Samuel. Automated network configuration: A comparison between ansible, puppet, and saltstack for network configuration. Master's thesis, Mälardalen University, School of Innovation, Design and Engineering, 2022. https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1667034&dswid=5705.

A | Appendix A: Scalable Web Server use case

This Appendix contains the code used for the SWS use case.

Terraform File

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "= 5.84"
    }
  }
}

# Configure the AWS Provider
provider "aws" {
  # region = var.region_name
  # access_key = var.aws_access_key
  # secret_key = var.aws_secret_key
}


#VPC
resource "aws_vpc" "common_network" {
  cidr_block = "10.100.0.0/16"
  tags = {
    Name = "concrete_vpc"
  }
```

```
}



# Subnet
resource "aws_subnet" "eks_subnet1_subnet" {
  vpc_id = aws_vpc.common_network.id
  cidr_block = "10.100.1.0/24"
  availability_zone = "us-west-1a"
  map_public_ip_on_launch = false
  tags = {
    Name = "eks_subnet1"
  }
}
# Subnet
resource "aws_subnet" "eks_subnet2_subnet" {
  vpc_id = aws_vpc.common_network.id
  cidr_block = "10.100.2.0/24"
  availability_zone = "us-west-1b"
  map_public_ip_on_launch = false
  tags = {
    Name = "eks_subnet2"
  }
}







module "eks" {
  source = "terraform-aws-modules/eks/aws"
  version = "20.33.1"
  cluster_addons = {
    coredns = {
      most_recent = true
    }
    kube-proxy = {
      most_recent = true
    }
```

```
  vpc-cni = {
    most_recent = true
  }
  metrics-server = {}
  eks-pod-identity-agent = {}
}

# cluster identifier
cluster_name                = "Cluster_main"

# allows public access to the public endpoint of the cluster
cluster_endpoint_public_access = true
# list of ip that can access the public endpoint
cluster_endpoint_public_access_cidrs = ["0.0.0.0/0"]
enable_cluster_creator_admin_permissions = true


# id of the security group for the cluster
cluster_security_group_id = aws_security_group.sg_security_group.id

# id of the vpc and subnets in which the cluster is deployed
vpc_id                  = aws_vpc.common_network.id
subnet_ids              = [aws_subnet.eks_subnet1_subnet.id,
    aws_subnet.eks_subnet2_subnet.id]

# id of the security group for the nodes
node_security_group_id = aws_security_group.sg_security_group.id


# defaults for the node groups
eks_managed_node_group_defaults = {
  ami_type      = "AL2_x86_64"
  instance_types = ["t2.micro"]
}

# node groups
eks_managed_node_groups = {
  # list of node groups
  main_group = {
```

```
      min_size    = 1
      max_size    = 3
      desired_size = 1

      instance_types = ["m5.large"]
      capacity_type = "ON_DEMAND"
    }
  }
}



# CREATING SECURITY_GROUP
resource "aws_security_group" "sg_security_group" {
  name       = "sg"
  vpc_id     = aws_vpc.common_network.id
  egress {
    from_port  = -1
    to_port    = -1
    protocol = "ICMP"
    cidr_blocks = ["0.0.0.0/0", ]
  }
  ingress {
    from_port  = 22
    to_port    = 22
    protocol = "TCP"
    cidr_blocks = ["0.0.0.0/0", ]
  }
  ingress {
    from_port  = 80
    to_port    = 80
    protocol = "TCP"
    cidr_blocks = ["0.0.0.0/0", ]
  }
  ingress {
    from_port  = 443
    to_port    = 443
    protocol = "TCP"
    cidr_blocks = ["0.0.0.0/0", ]
```

```
  }
  ingress {
    from_port  = 0
    to_port    = 65535
    protocol = "TCP"
    cidr_blocks = ["0.0.0.0/0", ]
  }
  ingress {
    from_port  = 10250
    to_port    = 10250
    protocol = "TCP"
    cidr_blocks = ["0.0.0.0/0", ]
  }
  ingress {
    from_port  = 30000
    to_port    = 32767
    protocol = "TCP"
    cidr_blocks = ["0.0.0.0/0", ]
  }
  egress {
    from_port  = 0
    to_port    = 0
    protocol = "-1"
    cidr_blocks = ["0.0.0.0/0", ]
  }
}
```

Kubernetes Files

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sws-deployment
  labels:
    app: sws
  namespace: default
spec:
  replicas: 2
  selector:
```

```
    matchLabels:
      app: sws
  template:
    metadata:
      labels:
        app: sws
    spec:
      containers:
      - name: webserver
        image: andreathesis/sws
        ports:
        - containerPort: 80
          name: http
        resources:
          requests:
            cpu: 100m
          limits:
            cpu: 200m
---

apiVersion: v1
kind: Service
metadata:
  name: sws-service-lb
  namespace: default
spec:
  type: LoadBalancer
  selector:
    app: sws
  ports:
  - name: http
    protocol: TCP
    port: 80
    targetPort: 80



---

apiVersion: autoscaling/v2
```

```
kind: HorizontalPodAutoscaler
metadata:
  name: sws-hpa
  namespace: default
spec:
 scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: sws-deployment
 minReplicas: 2
 maxReplicas: 5
 metrics:
 - type: Resource
   resource:
    name: cpu
    target:
      type: Utilization
      averageUtilization: 50
```

```
aws eks update-kubeconfig --name Cluster_main --region us-west-1
kubectl apply -f sws_deployment.yml
```

B | Appendix B: Web Scraping use case

This Appendix contains the code used for the Selenium Grid Web Scraping use case.

Terraform File

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "= 5.84"
    }
  }
}

# Configure the AWS Provider
provider "aws" {
  # region = var.region_name
  # access_key = var.aws_access_key
  # secret_key = var.aws_secret_key
}


#VPC
resource "aws_vpc" "common_network" {
  cidr_block = "10.100.0.0/16"
  tags = {
    Name = "concrete_vpc"
  }
```

```
}



# Subnet
resource "aws_subnet" "eks_subnet1_subnet" {
  vpc_id = aws_vpc.common_network.id
  cidr_block = "10.100.1.0/24"
  availability_zone = "us-west-1a"
  map_public_ip_on_launch = true
  tags = {
    Name = "eks_subnet1"
  }
}
# Subnet
resource "aws_subnet" "eks_subnet2_subnet" {
  vpc_id = aws_vpc.common_network.id
  cidr_block = "10.100.2.0/24"
  availability_zone = "us-west-1b"
  map_public_ip_on_launch = true
  tags = {
    Name = "eks_subnet2"
  }
}


# Internet Gateway Subnet
resource "aws_internet_gateway" "internet_internet_gw_subnet" {
  vpc_id = aws_vpc.common_network.id
  tags = {
    Name = "internet"
  }
}

# Route table
resource "aws_route_table" "internet_route_table" {
  vpc_id = aws_vpc.common_network.id
  route {
    cidr_block = "0.0.0.0/0"
```

```
    gateway_id = aws_internet_gateway.internet_internet_gw_subnet.id
  }
  tags = {
    Name = "internet"
  }
}



# Route table association
resource "aws_route_table_association" "eks_subnet1_subnet_route_assoc" {
  subnet_id     = aws_subnet.eks_subnet1_subnet.id
  route_table_id = aws_route_table.internet_route_table.id
}
# Route table association
resource "aws_route_table_association" "eks_subnet2_subnet_route_assoc" {
  subnet_id     = aws_subnet.eks_subnet2_subnet.id
  route_table_id = aws_route_table.internet_route_table.id
}




module "eks" {
  source = "terraform-aws-modules/eks/aws"
  version = "20.33.1"
  cluster_addons = {
    coredns = {
      most_recent = true
    }
    kube-proxy = {
      most_recent = true
    }
    vpc-cni = {
      most_recent = true
    }
    metrics-server = {}
    eks-pod-identity-agent = {}
  }
```

```
# cluster identifier
cluster_name                = "Cluster_main"

# allows public access to the public endpoint of the cluster
cluster_endpoint_public_access = true
# list of ip that can access the public endpoint
cluster_endpoint_public_access_cidrs = ["0.0.0.0/0"]
enable_cluster_creator_admin_permissions = true



# id of the security group for the cluster
cluster_security_group_id = aws_security_group.sg_security_group.id

# id of the vpc and subnets in which the cluster is deployed
vpc_id                  = aws_vpc.common_network.id
subnet_ids              = [aws_subnet.eks_subnet1_subnet.id,
    aws_subnet.eks_subnet2_subnet.id]

# id of the security group for the nodes
node_security_group_id = aws_security_group.sg_security_group.id



# defaults for the node groups
eks_managed_node_group_defaults = {
  ami_type      = "AL2_x86_64"
  instance_types = ["t2.micro"]
}

# node groups
eks_managed_node_groups = {
  # list of node groups
  main_group = {
    min_size    = 1
    max_size    = 3
    desired_size = 1

    instance_types = ["t3.medium"]
    capacity_type = "ON_DEMAND"
  }
```

```
    second_group = {
      min_size    = 1
      max_size    = 4
      desired_size = 1

      instance_types = ["t3.medium"]
      capacity_type = "ON_DEMAND"
    }
  }
}



# CREATING SECURITY_GROUP
resource "aws_security_group" "sg_security_group" {
  name       = "sg"
  vpc_id     = aws_vpc.common_network.id
  egress {
    from_port  = -1
    to_port    = -1
    protocol = "ICMP"
    cidr_blocks = ["0.0.0.0/0", ]
  }
  ingress {
    from_port  = 22
    to_port    = 22
    protocol = "TCP"
    cidr_blocks = ["0.0.0.0/0", ]
  }
  ingress {
    from_port  = 80
    to_port    = 80
    protocol = "TCP"
    cidr_blocks = ["0.0.0.0/0", ]
  }
  ingress {
    from_port  = 443
    to_port    = 443
    protocol = "TCP"
```

```
    cidr_blocks = ["0.0.0.0/0", ]
  }
  ingress {
    from_port  = 0
    to_port    = 65535
    protocol = "TCP"
    cidr_blocks = ["0.0.0.0/0", ]
  }
  ingress {
    from_port  = 10250
    to_port    = 10250
    protocol = "TCP"
    cidr_blocks = ["0.0.0.0/0", ]
  }
  ingress {
    from_port  = 30000
    to_port    = 32767
    protocol = "TCP"
    cidr_blocks = ["0.0.0.0/0", ]
  }
  egress {
    from_port  = 0
    to_port    = 0
    protocol = "-1"
    cidr_blocks = ["0.0.0.0/0", ]
  }
}
```

Kubernetes Files

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: selenium-hub-deployment
  labels:
    app: selenium-hub
  namespace: selenium
spec:
  replicas: 1
```

```
  selector:
    matchLabels:
      app: selenium-hub
  template:
    metadata:
      labels:
        app: selenium-hub
    spec:
      containers:
      - name: selenium-hub
        image: selenium/hub:4.1.0
        ports:
        - containerPort: 4442
          name: port1
        - containerPort: 4443
          name: port2
        - containerPort: 4444
          name: port3

---

apiVersion: v1
kind: Service
metadata:
  name: selenium-hub-service-lb
  namespace: selenium
spec:
  type: LoadBalancer
  selector:
    app: selenium-hub
  ports:
  - name: port1
    protocol: TCP
    port: 4442
    targetPort: 4442
  - name: port2
    protocol: TCP
    port: 4443
    targetPort: 4443
```

```
  - name: port3
    protocol: TCP
    port: 4444
    targetPort: 4444
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: selenium-nodes-deployment
  labels:
    app: selenium-nodes
  namespace: selenium
spec:
  replicas: 2
  selector:
    matchLabels:
      app: selenium-nodes
  template:
    metadata:
      labels:
        app: selenium-nodes
    spec:
      containers:
      - name: selenium-node
        image: andreathesis/sel-node
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: selenium-grid-metric-exporter-deployment
  labels:
    app: selenium-grid-metric-exporter
  namespace: selenium
spec:
  replicas: 1
  selector:
    matchLabels:
      app: selenium-grid-metric-exporter
  template:
```

```
    metadata:
      labels:
        app: selenium-grid-metric-exporter
    spec:
      containers:
      - name: metric-exporter
        image: andreathesis/queue-size-exporter
        ports:
        - containerPort: 8080
          name: http


---

apiVersion: v1
kind: Service
metadata:
  name: selenium-grid-metric-exporter-service-ip
  namespace: selenium
spec:
  selector:
    app: selenium-grid-metric-exporter
  ports:
  - name: http
    protocol: TCP
    port: 8080
    targetPort: 8080
```

```
aws eks update-kubeconfig --name Cluster_main --region us-west-1
kubectl create namespace selenium
kubectl apply -f sws_deployment.yml
kubectl apply -f selenium-hub_deployment.yml
kubectl apply -f selenium-nodes_deployment.yml
kubectl apply -f selenium-grid-metric-exporter_deployment.yml
kubectl apply -f
    https://raw.githubusercontent.com/andrimega/DOML-EKS-validation-local/refs/heads/main/
kubectl apply -f
    https://raw.githubusercontent.com/andrimega/DOML-EKS-validation-local/refs/heads/main/
kubectl apply -f
```

```
   https://raw.githubusercontent.com/andrimega/DOML-EKS-validation-local/refs/heads/main/Selen
kubectl apply -f
   https://raw.githubusercontent.com/prometheus-operator/prometheus-operator/main/bundle.yaml
kubectl apply -f
   https://raw.githubusercontent.com/andrimega/DOML-EKS-validation-local/refs/heads/main/Selen
kubectl apply -f
   https://github.com/kedacore/keda/releases/download/v2.13.0/keda-2.13.0.yaml
kubectl apply -f
   https://raw.githubusercontent.com/andrimega/DOML-EKS-validation-local/refs/heads/main/Selen
```

List of Figures

1.1 Traditional Datacenters and Cloud Datacenters, image from [38] 6
1.2 DevOps process, image from [7] 10
1.3 Terraform workflow, image from the Terraform website [37] 13
1.4 Ansible configuration process, from [39] 16
1.5 PIACERE Framework Design Time components, from [29] 21
1.6 PIACERE Design Process, from [28] 22
1.7 Docker stack, from the Docker website [9] 24

2.1 Kubernetes architecture, from the article: Kubernetes — Architecture Overview [20] 35
2.2 ContainerNetwork, ContainerVolume, and related classes, from DOML specification [10] 39
2.3 AutoScalingGroup classes, from DOML specification [10] 39
2.4 Docker Swarm classes, from DOML specification [10] 40

3.1 DOML-E extension menu, from [2] 43
3.2 Logical division in what and where 45
3.3 Kubernetes DOML application design flow 48
3.4 Kubernetes DOML application code generation 49
3.5 New DOML classes 54
3.6 ICG architecture, from PIACERE deliverable [1] 55

4.1 SWS model diagram 68
4.2 SWS resources 69
4.3 SWS webpage 69
4.4 HPA utilization 69
4.5 SWS scaled resources 70
4.6 Selenium Grid Scaling Flow 73
4.7 Selenium Grid model diagram 75
4.8 Selenium resources 76
4.9 Selenium Grid UI 76

4.10 Prometheus resources 77
4.11 Keda resources 77
4.12 Selenium scaled resources 77

List of Tables

1.1 Kubernetes and Docker Swarm comparison 29

2.1 DOML and Kubernetes comparison 40

4.1 DOML-K8s Requirements 63
4.2 Requirements Validation 65

Acknowledgements

Here you might want to acknowledge someone.