POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Supporting the Development of Infrastructure as Code Using Ansible: a Smart IDE Integrating External Sources

TESI DI LAUREA MAGISTRALE IN
COMPUTER SCIENCE AND ENGINEERING - INGEGNERIA INFORMATICA

Author: **Michail Bachras**

Student ID: 938860
Advisor: Elisabetta Di Nitto
Academic Year: 2020-2021

Abstract

Infrastructure as Code (IaC) is a process that enables developers to provision and manage infrastructure resources through code rather than manually intervening. This paradigm has grown in popularity in recent years as it addresses the issue of inconsistency, which occurs when different people want to use the same configuration on different machines, a problem that cloud computing struggles to address. However, several impediments to IaC adoption exist, including a dispersed field of technologies and a lack of support within existing tools. SODALITE, a European project, aims to address these issues by providing an end-to-end solution for the deployment of a complex application. In this context, Ansible DSL, an abstraction of Ansible, and Ansible editor, a text editor, have been implemented to aid in the operationalization of the cloud application.

The contribution of this thesis is to extend the features of the Ansible editor in multiple directions. More specifically, we integrate disparate information sources into the Ansible editor to provide a large spectrum of information to the developers and enable direct access to preexisting, reusable elements, such as Ansible content, and TOSCA models. In this way, the Ansible editor exploits the nested information to provide semantic suggestions, validation mechanisms and code smell detection utilities to the developers, that ensure the quality of the Ansible scripts and lower the development costs.

Our second contribution is the seamless integration of the various phases of an application's deployment procedure into a single approach. More specifically, we enable smooth continuity between resource provisioning, which in SODALITE is performed by executing a deployment model written in the TOSCA language, and the creation and configuration of software layers on top of the provisioned resources, which in SODALITE is performed through the execution of Ansible scripts.

Overall, our approach enhances an already innovative development environment with features not available in other tools on the market, thereby moving the IaC paradigm one step closer to adoption.

Key-words: Infrastructure as Code, TOSCA, Ansible, IDE

Abstract in lingua italiana

Infrastructure as Code (IaC) è un processo che consente agli sviluppatori di fornire e gestire le risorse dell'infrastruttura tramite codice anziché intervenire manualmente. Questo paradigma è diventato popolare negli ultimi anni poiché affronta il problema dell'incoerenza, che si verifica quando persone diverse vogliono utilizzare la stessa configurazione su macchine diverse, un problema che il cloud computing fatica ad affrontare. Tuttavia, esistono diversi ostacoli all'adozione dell'IaC, tra cui un campo di tecnologie disperso e la mancanza di supporto all'interno degli strumenti esistenti. SODALITE, un progetto europeo, mira ad affrontare questi problemi fornendo una soluzione end-to-end per l'implementazione di un'applicazione complessa. In questo contesto, Ansible DSL, un'astrazione di Ansible, e Ansible editor, un editor di testo, sono stati implementati per facilitare l'operazionalizzazione dell'applicazione cloud.

Il contributo di questa tesi è di estendere le caratteristiche dell'editor Ansible in più direzioni. Più specificamente, integriamo diverse fonti di informazioni nell'editor Ansible per fornire un ampio spettro di informazioni agli sviluppatori e consentire l'accesso diretto a elementi preesistenti e riutilizzabili, come i contenuti Ansible e i modelli TOSCA. In questo modo, l'editor di Ansible sfrutta le informazioni nidificate per fornire agli sviluppatori suggerimenti semantici, meccanismi di convalida e utilità di rilevamento dell'odore del codice, che garantiscono la qualità degli script Ansible e riducono i costi di sviluppo.

Il nostro secondo contributo è la perfetta integrazione delle varie fasi della procedura di distribuzione di un'applicazione in un unico approccio. Più specificamente, consentiamo una continuità regolare tra il provisioning delle risorse, che in SODALITE viene eseguito eseguendo un modello di distribuzione scritto nel linguaggio TOSCA, e la creazione e configurazione di livelli software in aggiunta alle risorse fornite, che in SODALITE viene eseguita attraverso l'esecuzione di script Ansible.

Nel complesso, il nostro approccio migliora un ambiente di sviluppo già innovativo con funzionalità non disponibili in altri strumenti sul mercato, avvicinando così il paradigma IaC all'adozione.

Parole chiave: Infrastructure as Code, TOSCA, Ansible, IDE

Contents

Abstract i

Abstract in lingua italiana iii

Contents v

1. Introduction 1

1.1 Motivation 1

1.2 Scope of the thesis 6

1.3 Thesis Structure 7

2. Background and Related work 9

2.1 Iac Tools 9

2.1.1 Configuration Management Tools 9

2.1.2 Provisioning tools 15

2.2 IaC Research Projects 21

2.3 Research Gaps and Contribution 27

3. Problem Definition 31

3.1 Main Goal 31

3.2 Ansible editor overview 32

3.3 TOSCA and Ansible integration 33

3.4 Ansible Content 36

3.5 Ansible DSL and editor extensions 37

4. Solution Design 39

4.1 Ansible Editor Architecture and Code Organization 39

4.1.1 Xtext framework 39

4.1.2 Architecture 40

4.1.3 Implementation Structure ... 42
4.2 TOSCA and Ansible integration ... 43
4.2.1 Guided creation of abstract Ansible models ... 45
4.2.2 Connection of Ansible editor with SODALITE KB ... 47
4.2.3 Portability of Resource Models ... 49
4.3 Supporting the reuse of Ansible elements ... 51
4.3.1 Ansible content ... 51
4.3.2 Proposal Provider ... 53
4.3.3 Validator ... 58
4.3.4 Ansible Defect Predictor ... 62
5. Evaluation ... 65
5.1 Evaluation objectives ... 65
5.2 Evaluation procedure ... 66
5.3 Evaluation Results ... 70
6. Conclusion ... 73
6.1 Summary ... 73
6.2 Future Work ... 75
Bibliography ... 77
A. Appendix A ... 83
List of Figures ... 85
List of Tables ... 87
Acknowledgements ... 89

1. Introduction

1.1 Motivation

Nowadays, the IT market is dominated by the need to release software quickly and frequently in order to meet the constantly changing needs of customers and users. Consequently, IT organizations were scrambling to find new ways to meet their customers' expectations while remaining competitive and viable. Cloud computing [1] emerged in this direction, intending to address some of the challenges facing the IT industry.

Cloud computing offers new capabilities and opportunities that traditional IT solutions cannot, as well as a number of advantages to organizations that adopt this model [2]. The pay-per-use model and on-demand resource provisioning, which allow for dynamic resource adjustment based on service needs at any given period, are two of the primary reasons for adopting cloud computing. This is highly advantageous, especially for new and small businesses, because it eliminates the need to acquire resources in advance to meet peak usage and only pays for resources that will cover current demand. The client is also relieved of system maintenance, hardware upgrades, and data backups. These three administration procedures are the responsibility of cloud providers; they are frequently expensive and time-consuming, and they have a negative impact on the development cycle and decrease flexibility.

Cloud providers offer various types of services to their clients based on their needs and expertise [3]. There are three types of service models: software as a service (SaaS), platform as a service (PaaS), and infrastructure as a service (IaaS). In SaaS, cloud providers provide software services on the application layer, allowing clients to access software applications without worrying about deployment, configuration, or updates. PaaS is a computing platform that is delivered to clients as a web service [4]. The advantage of this type of service is that clients can build and deploy their applications in pre-configured development environments, utilizing virtualized hardware, dynamic resource allocation, and data redundancy. On the other hand, IaaS provides computational resources and infrastructure, such as CPUs, storage,

and networks, that can be "created," "destroyed," and managed as needed by the client. In this case, the client has complete control over the "rented" resources and can deploy and run applications and operating systems.

Another type of cloud computing classification is deployment models, which specify how cloud services are made available to consumers [5]. The four deployment models are public, private, hybrid, and community cloud. On the one hand, the public cloud's resources are accessible to the general public via a web service and are entirely controlled by the cloud provider who provides the cloud services. On the other hand, a private cloud consists of resources dedicated to and maintained by a single organization and can only be accessed by that entity. Finally, a community cloud is managed by a group of organizations that share computing resources, whereas a hybrid cloud is a combination of two or more deployment models that will operate as a unified system.

Regardless of the various service and deployment models that have been introduced, cloud computing is insufficient to address all of the major challenges for continuous and rapid software delivery. In this context, IT enterprises focused on using agile and lean methodologies throughout the software development cycle to reduce development time and increase responsiveness. However, these methodologies were primarily focused on the development side, ignoring the operations side [6], causing bottlenecks in the development process and delaying software delivery. A new paradigm known as DevOps emerges to bridge this gap and overcome barriers to effective communication between software development and operations teams.

Primary goal of DevOps is to shorten the release cycle and provide continuous delivery of high-quality software [7]. One major requirement for accomplishing this is to automate the entire process of configuring infrastructure components in a repeatable manner [8]. Infrastructure as Code, a fundamental principle of DevOps, is used to implement such an automated process [9]. IaC views IT infrastructure as software, allowing it to use software principles, methodologies, and tools to accelerate software operations.

When we consider the properties that IaC induces in the managed environment [10], the significance of this practice becomes clear:

1. Software-defined infrastructure: Computing resources are codified into simple text-based, machine-readable configuration files that describe the desired infrastructure state. This code-oriented approach allows the infrastructure to be deployed in stages and treats IaC as any other software development cycle. In addition, developers can test their applications in environments that are easily deployed and managed,

receiving immediate feedback on real-time metrics of their applications, such as security checking.

2. Consistency of configuration across multiple environments: Manual configuration of infrastructure components is error-prone, especially when the number of resources is significant, leading to configuration drift. In contrast, with IaC, the configuration procedure is automatic, ensuring that the same configuration is deployed across all desired components.
3. Auditability: By automating the provisioning/configuration procedure and using a software-defined approach, it is possible to version the configuration files and track the changes made during the evolution of the deployed scripts. As a result, unstable systems can be quickly reverted to previous versions and restored to operational status.
4. Reproducibility: Manually configuring an infrastructure is a time-consuming and labor-intensive process. One of the primary benefits of IaC is the ease with which an administrator can provision resources and configure each component to different environments.
5. Immutability: When using IaC, the configuration file is the only thing that matters, which means that the provisioned infrastructure changes only when the relevant configuration file changes. As a result, system administrators can maintain consistency in computing resources across multiple environments.

DevOps practices, particularly IaC, are not restricted to the cloud computing context but can also be applied to on-premise infrastructure with the same benefits described above. In contrast, combining cloud computing and IaC practices multiply each methodology's benefits and significantly improve software development acceleration and infrastructure automation. Businesses and organizations rode the new wave of automation, attempting to adapt the fundamental principles of IaC to their specific needs. As a result, a plethora of disparate technologies and IaC languages emerged, with requirements and goals frequently overlapping. Furthermore, some IaC languages and tools are tied to specific technologies and platforms, limiting organizations' flexibility in collaborating with different service providers.

OASIS developed an open standard called TOSCA [11] in this fragmented field to provide a common language for describing the relationships and dependencies between cloud-based services and applications in a vendor-neutral manner. TOSCA introduced a unified model of common cloud resources, allowing developers from various backgrounds to use a standard specification to describe the structure of a cloud service as well as its operational aspects. This improves interoperability among service providers, lowers costs, and stimulates innovation, bridging the gap between various IaC approaches.

TOSCA allows you to design the infrastructure topology model of a cloud application as a graph, with typed nodes and typed edges serving as building blocks. The nodes represent the application components of the service, whereas the edges represent the various types of relationships between two or more application components. TOSCA also defined deployment and management operations for each application component, which are used throughout the component's lifecycle. Nonetheless, TOSCA is unable to define how these operations are implemented, so artifacts containing the content required to realize these operations must be included. The artifacts content can be of various types (scripts, executable programs, libraries), but Ansible is one of the most popular options [12].

Ansible is a free and open-source automation tool that can be used for provisioning, configuration management, application deployment, and orchestration. It is written in Python, and commands are scripted using YAML syntax.

One significant issue with IaC languages and tools/methodologies such as TOSCA and Ansible is a lack of support from text editors and IDEs, making code development time-consuming. For example, if a developer wants to use Ansible to create a deployment file, the obvious choice is to use a text editor that supports YAML syntax, such as Vim. This is not the best approach because the developer receives no feedback on the written code or suggestions to speed up and guide the development process. Features like code suggestions and error messages are handy, especially for new developers, and lay the groundwork for broader adoption of the respective tools and, more broadly, IaC.

SODALITE [13] (SOftware Defined AppLication Infrastructures managemenT and Engineering) is a European H2020 project in which multiple IT companies and universities from across Europe collaborate to achieve a specific vision and contribute to IaC's widespread adoption. SODALITE aims to address the issues that have arisen due to the rapid growth of cloud computing and the emergence of a plethora of disparate technologies. Its goal is to create a common platform for developers to deploy and run modern cloud applications in heterogeneous execution environments such as HPC or cloud. Vision of SODALITE is as follows [13]:

"The SODALITE vision is to support Digital Transformation of European Industry through (1) increasing design and runtime effectiveness of software-defined infrastructures to ensure high-performance execution over dynamic heterogeneous execution environments; (2) increasing simplicity of modelling applications and infrastructures to improve manageability, collaboration, and time to market."

To accomplish these goals, SODALITE leverages the benefits of the IaC paradigm, such as increased automation and limited manual intervention, and focuses on

creating an environment that supports the entire IaC lifecycle, from resource provisioning to resource management and, finally, "destruction". In addition, it offers a toolkit that enables application developers and infrastructure operators to create high-level models by abstracting from a specific IaC technology, simplifying the modeling and deployment process.

TOSCA and Ansible, two well-known IaC methodologies, are used in the SODALITE framework. Their integration occurs as previously described, with TOSCA defining the application topology graph and Ansible implementing the deployment and management operations of the container application components. However, SODALITE does not directly provide TOSCA in its pure form , but rather a Domain Specific Language that is similar to TOSCA and follows the same principles. The framework includes a smart IDE that provides suggestions during the development process and flags potential syntax errors via a syntax validation process to support this TOSCA-like language.

The semantic validation that is performed in the defined abstract models is another essential feature of SODALITE IDE. These models are mapped into knowledge graphs and stored in a knowledge base, where logic-based inference is used to enrich and validate the models in missing TOSCA definitions and relationship requirements [14]. In addition to TOSCA abstract models, SODALITE IDE allows users to create Ansible models using an integrated editor that supports a Domain Specific Language, similar to Ansible but with another layer of abstraction with respect to the original language, and to generate Ansible scripts directly from these Ansible models. This is done to simplify the development process for the user by grouping Ansible elements that are related to the same aspect. The Ansible editor's innovation is syntax validation and content suggestions and the ability to write Ansible scripts within the context of a TOSCA topology model. This allows for direct interaction between the script that will implement the operation of an application component and the component itself.

The current project is housed within the SODALITE IDE, and our goal is to create a development environment that includes all of the features required to provide users with a smooth and straightforward experience when modeling an application's deployment procedure. More specifically, we extend the capabilities of the included Ansible editor by integrating a diverse set of knowledge sources into it, assisting developers, and providing a wide range of information within the same text editor. Furthermore, we bring TOSCA deployment models and Ansible scripts closer together and facilitate information exchange between them, reducing the development time required to transfer information from one model to the other. To the best of our knowledge, no other editor provides such a comprehensive set of

features, and as a result, our work advances the adoption of IaC by addressing these significant challenges.

1.2 Scope of the thesis

This thesis aims to extend SODALITE IDE already innovative capabilities and go a step further in meeting user needs by providing high-quality support throughout the cloud application modeling procedure. The IDE enables the creation of the so-called Resource Model, which consists of various types of application and infrastructure components and their relationships, as well as Ansible models that implement the deployment and management operations of a typed component. Furthermore, the end-user can combine the previously defined components to form an AADM topology graph representing the cloud application and specify the non-functional requirements that must be met at run-time. Our work focuses on the first two stages of the modeling procedure to make valuable contributions that will improve users' interaction with the IDE.

As inspiration, we will take modern IDEs that provide multiple features and streamline the development workflow. Autocompletion, error messages, and code suggestions are extremely useful for the user and significantly speed up the development process because he/she has all of the necessary information in one place without having to search through lengthy documentation. This is especially important in the case of Ansible, where Ansible collections, modules, and roles are dispersed across multiple repositories, requiring the user to strain for the desired information. As a result, our primary focus will be connecting SODALITE IDE with information sources that will provide the end-user with valuable suggestions and constant feedback. These information sources will be the already implemented Knowledge Base, which stores SODALITE Resource Models, and a database containing Ansible content from repositories such as Ansible Galaxy and Github.

A significant contribution of this project is the multiple validation mechanisms that check Ansible models for validity issues and provide clear and meaningful recommendations to the end-user on how to fix them. For example, the Ansible editor notifies the user if an Ansible collection name has the incorrect format and instructs the user how to correct it. Such errors cause problems during cloud application deployment and management and necessitate a significant amount of time to identify the errors in the source code. As a result, accurate error messages, accompanied by quick fixes whenever possible, save the end-user a significant amount of time, increase productivity, and increase user satisfaction.

Another important aspect of our work is the mechanisms that guide the user through defining an Ansible model and importing the generated scripts into the corresponding Resource Model. After creating a Resource Model, the Ansible scripts that implement its TOSCA operations must be imported into the RM. For this reason, we developed a standardized procedure that guides the user through the definition of the appropriate Ansible model, followed by the generation and integration of the corresponding Ansible script into the RM. In this way, the user is not lost among the various DSL provided by SODALITE IDE but instead remains in a specific chain of activities. These mechanisms, however, do not alter the existing transparency property of the Ansible script's origin. Transparency property enables the user to select an Ansible script written with an external editor for a TOSCA operation rather than firstly defining an abstract Ansible model (i.e., .ans file) and then generating the concrete Ansible script. Thus, the only requirement for importing an Ansible script is to specify the script's local path in the RM without restricting development to our Ansible editor.

1.3 Thesis Structure

In this section, we will go over the structure of the current thesis and briefly describe the chapters that follow.

Chapter 2 introduces the state of art, presenting various Infrastructure as Code tools and research projects developed to meet the needs of the DevOps field. Furthermore, it delivers an overview of the technologies used in this thesis to provide readers with the necessary context.

Chapter 3 describes the goal of this thesis and the three different directions in which we moved to develop our ideas.

Chapter 4 describes the implementation decisions we made and the features we created to solve the problems we identified, with examples.

Chapter 5 describes our work's evaluation procedure, in which external testers compared the implemented editors to an existing editor used by the community to develop Ansible scripts. The testers experimented with actual use cases and reported their findings via a questionnaire.

Chapter 6 summarizes the various aspects of this thesis and suggests potential future research topics.

2. Background and Related work

This chapter presents an overview of the most popular IaC tools and languages, and we document some of the recent research projects related to our work. Section 2.1 discusses some of the most commonly used configuration and provisioning tools, describes their key features and compares them. Section 2.2 examines the state of the art in IaC research projects and provides an overview of their purpose and features. Section 2.3 identifies research gaps in modern literature and where our work resides among the state of the art.

2.1 Iac Tools

In Chapter 1.1, we discussed the concept of Infrastructure as Code and the value it brings to the IT industry. DevOps engineers can use IaC to define all aspects of the desired infrastructure and build an infrastructure system from the ground up by leveraging a variety of IaC tools that streamline the development process and cover all aspects of the modeled infrastructure. DevOps engineers can select from a large pool of IaC tools, each with its own set of features and serving diverse purposes. Based on this, IaC tools can be divided into two broad categories: configuration management tools and provisioning tools. In the following sections both types of tools are presented.

2.1.1 Configuration Management Tools

Configuration management tools are designed to install software on existing infrastructure, verify its state, and make the necessary configurations on provisioned infrastructure to ensure software's smooth execution. They provide an automated method for deploying and configuring applications and environments across various infrastructure components without the need for manual intervention, which can be difficult when managing a large number of components. In our work, we study many of the configuration tools used by the community, but we focus on the four most popular: Ansible, Chef, Puppet, and Saltstack, and we examine each one's main

characteristics, highlighting the differences between them. Moreover, we dedicate a subsection to describe the fundamental concepts of Ansible, as it is the core of our work.

2.1.1.1 Ansible overview

Ansible is a free and open-source configuration management tool for provisioning and configuring infrastructure components as well as deploying applications. It employs a Python-based YAML syntax in which the developer specifies the exact step-by-step procedure for bringing the infrastructure to the desired state. Furthermore, it is agentless, so the system administrator does not have to set it up separately for each management interface component [15].

An Ansible file is known as a playbook, and it contains a list of sequential operations known as tasks. A playbook can be run on specific hosts chosen by the developer while the available hosts' names and IP addresses are saved in an inventory. Hosts can also be organized into groups, allowing a playbook to be executed only on hosts of a specific group.

Tasks are the primary units of action in Ansible and are executed on selected hosts from the inventory. Different sets of tasks within a playbook may need to be executed in different hosts, which is accomplished by organizing them in groups known as plays. For example, suppose we have a playbook with 20 tasks, and the developer wants to run the first ten tasks on a group of nodes called "first group" and the following ten tasks on a group of nodes called "second group". In this case, he/she will define two plays with the names "first group" and "second group," with each containing the corresponding tasks.

The module is another central concept in Ansible. A module is a reusable unit of code that, like a function in a programming language, performs a specific operation with the help of some parameters. A task can call a single module and pass the necessary arguments as parameters in the form of variables or values. Modules are organized into Ansible collections, which are packages that contain Ansible content. Ansible includes many collections [16], but users can also access Ansible Galaxy, a massive repository of Ansible content, from which they can download and install Ansible collections and use the contained modules.

Aside from modules, another reusable Ansible entity is the role. A role is a portable unit of Ansible code that, when executed, accomplishes a specific task. Each role is written in YAML and is essentially a subset of a play that can be imported directly into an Ansible playbook. The utility of roles is that they enable the automation of reusable sets of operations that can be shared among the community in a self-contained manner because roles include all the necessary tools for execution such as

variables, files, tasks, etc. Users can find and download Ansible Roles from Ansible Galaxy and upload their own to share with the community.

Figure 2.1 shows an example of a simple Ansible playbook.

```
1  ---
2    - name: Playbook
3      hosts: webservers
4      become: yes
5      tasks:
6        - name: ensure apache is at the latest version
7          yum:
8            name: httpd
9            state: latest
10       - name: ensure apache is running
11         service:
12           name: httpd
13           state: started
```

Figure 2.1: Ansible playbook

The goal of this playbook is to install an Apache Web server on multiple hosts. More specifically, in the second line, we define the playbook's name, and in the third line, we specify that the playbook will be executed on hosts belonging to the host group *webservers*. We tell Ansible that the playbook will be executed with elevated privileges in the fourth line, and in the fifth line, we begin the section where all the tasks will be defined. In the task section, we will define two tasks using two modules, the first of which is *yum* and the second of which is *service*. In the first task with the *yum* module, the state *latest* indicates that the package *httpd* should be installed if it is not already installed or that it should be upgraded to the most recent version available if it is already installed. On the second task with the *service* module, we use the *state: started* to ensure that the service named *httpd* is started and running. If the service is already started and running, Ansible will not restart it.

2.1.1.2 Other configuration management tools

Puppet [17],[18] is an open-source configuration management and deployment tool. It uses a customized Domain Specific Language implemented in Ruby and runs a master-agent setup, following a model-driven approach. The main elements of Puppet are Resources, Classes, Manifests, and Modules. Each Resource defines the desired state for a part of the system and are the fundamental building blocks to model the system state. Classes are collections of Resources, while Manifests are collections of Classes. Finally, Modules represent a particular automation task and contain all the previously described elements.

Chef [19] is another configuration management tool that helps automate the IT infrastructure. It uses a Ruby-based Domain Specific Language to write the configurations and follows master-agent architecture, where a logical Chef workstation controls the configurations from the master to the agents. The main components of Chef are the following [20]:

- Chef workstation: It is the main point of interaction between the user and the system. Through this component, users can author and test cookbooks and recipes, manage the system's physical and virtual nodes, and interact with the Chef server through Knife, a command-line tool.
- Chef client node: A node is a physical or virtual machine that Chef manages. A Chef client is installed on every node and performs all the defined configuration tasks to bring the component into the desired state. The configuration procedure is performed by a built-in tool called Ohai.
- Chef server: This component acts as a hub of information, where all the critical files and data are kept, such as cookbooks, recipes, policies, etc. Each node that runs a Chef-client uses a pull-based approach and periodically requests the Chef server for the necessary configuration data, applying the defined configuration.

Chef exploits the concepts of "recipes" and "cookbooks" to specify the exact steps to be followed that will bring the infrastructure components to the desired state. Cookbooks define the automation workflow that will be executed and organize the related recipes. Recipes are Ruby scripts that support the required resources and specify how each resource will be applied. Finally, roles contain a list of recipes, which, after their retrieval from the Chef server, are executed by the Chef client and create the necessary configuration.

Saltstack [21] is an open-source tool to configure IT infrastructure. It leverages a mixed model, where there are two distinct types of nodes, the Salt master and the Salt minion, while at the same time, it uses a decentralized approach. The system can have multiple masters to which groups of minions are connected and get configuration data. The user can push configurations and updates to the minions, but also minions are allowed to check for updates and pull them accordingly [22]. As a result, this hybrid architecture of Saltstack achieves high speed, low latency, high scalability and resiliency [23]. Other essential characteristics of Saltstack are the Salt reactors, which are responsible for listening to new events on the minions, the grains, which retrieve information from the client nodes; and the pillars, which serve data to minions in a confidential, secure and node-specific manner. In addition, Saltstack is language-agnostic as it can render scripts in multiple languages such as Python, YAML and JSON.

Parameters	**Ansible**	**Chef**	**Puppet**	**Saltstack**
Configuration Language	YAML	DSL based on Ruby	DSL	YAML
Language Type	Procedural	Procedural	Declarative	Declarative
Configuration file	Playbook	Recipe	Manifest	State file
Architecture	Master only	Master-Agent	Master-Agent	Master-Agent
Configuration method	Push	Pull	Pull	Push
Communication	SSH	HTTPS	HTTPS	ZeroMQ, SSH
Supported OS	Master: Linux, MacOS	Master: Linux, MacOS Agents: Linux, Windows, MacOS	Master: Linux Agents: Linux, Windows, MacOS	Master: Linux Agents:Linux, Windows, MacOS

Table 2.1: Comparison of configuration tools

Table 2.1 summarizes the previously described characteristics for the tools under consideration, making it easier for the reader to identify significant differences between them [24]. The following parameters were considered in order to make the comparison:

- The configuration language provided by each tool: In Ansible and Saltstack, users script commands in YAML, i.e., Yet Another Markup Language, a user-friendly syntax that significantly reduces the learning curve. On the contrary, Chef and Puppet use their dedicated Domain-Specific Languages, which increases the time required for a developer to learn and write commands in each tool.
- The language's style: Ansible and Chef use a procedural programming paradigm, in which the developer writes code that specifies how to achieve the desired end state step by step. Puppet and Saltstack use a

declarative approach in which the developer specifies the desired end state, and the configuration tool determines the most efficient way to achieve that state.

- What kind of configuration file each tool employs and how this file is called.
- The infrastructure component architecture that each tool applies to the system: Chef, Puppet, and Saltstack have a master-agent architecture, in which the main server runs on the master machine of the system, and every other related machine runs a client that acts as an agent for the communication with the master machine. Ansible has only a master running on the server machine but no agents running on the client machines.
- How each system node receives the appropriate configuration: Puppet and Chef adopt a pull configuration approach, whereas Ansible and Saltstack adopt a push configuration approach. In the first case, the client machines automatically pull all the configurations from the master server without any commands, while in the second case, all the configurations present in the central server are pushed to the client machines.
- The node-to-node communication protocol: The communication between system nodes in Puppet and Chef is based on the HTTPS protocol, whereas Ansible and Saltstack use SSH. Saltstack also supports ZeroMQ, a high-performance asynchronous messaging library that allows for faster communication.
- The supported operating systems by each node type.

In general, the emergence of numerous configuration management tools has resulted from the fact that no single tool completely meets the needs of the DevOps community. For this reason, as we saw, there have been numerous attempts to meet these needs with various characteristics and features. Therefore, the developers should review and examine the features provided by each tool before making a decision based on the needs of their system. For example, in a system with many nodes, a reasonable decision would be to select a tool with a push configuration approach that minimizes the communication overhead between the clients and the master compared to a pull configuration approach in which the clients constantly request potential updates from the master.

2.1.2 Provisioning tools

Provisioning tools are intended to allow for the automatic provisioning of servers, networks, users, and services via code without the need for manual intervention. In this way, the provisioning work required by developers when deploying a new application is significantly decreased and reduced to just a script to be executed that contains all of the infrastructure specifications. Among the tools investigated, we concentrated on TOSCA, Terraform, Openstack Heat, and CloudFormation and briefly compared their features. Given the significance of TOSCA in this thesis, we will devote a separate subsection to describe its objectives and characteristics.

2.1.2.1 TOSCA Overview

The Topology and Orchestration Specification for Cloud Applications(TOSCA) is an official OASIS standard whose primary goal is to standardize the description of cloud application's structure and automate their deployment and management [25]. In order to achieve this, TOSCA focuses on the following three sub-goals [26]:

1. Automated application deployment and management: TOSCA provides a Domain Specific Language that allows developers to describe a complex cloud application in an abstract and modular manner. Furthermore, developers are able to create management plans that describe best practices for deployment and management, making complex application supervision an automated and less error-prone procedure.
2. Portability of application descriptions and their management: TOSCA provides a standardized way to describe a multi-component application and exploits the portability of workflow languages, such as BPMN and BPEL, to describe deployment and management plans.
3. Interoperability and reusability of components: TOSCA provides a standardized approach to describe the components of cloud applications, which not only improves interoperability and reusability amongst different cloud providers but also reduces developers' effort to design a cloud application's topology.

TOSCA offers an YAML-based modeling language, which allows describing, abstractly, a cloud service's topology and its operational aspects through management plans. It encodes a cloud application in the concept of "Service Template", which consists of two different components, the "Topology Template" and the "Management Plans":

- Topology Template: The topology template is a typed topology graph representing the structure of a cloud application and capturing the relationships and dependencies between the application components.

Types, Templates, and Instances are the three levels of abstraction used to describe the topology graph. This layered approach promotes the reusability of application components (see sub-goal three above) through type definition and allows for the description of a cloud application at various levels of granularity.

  - Types are defined separately from the topology and specify the structure of a template's features, such as properties, interfaces, requirements, capabilities, and policies [27],[28]. Each type maintains a set of operations bundled in interfaces and enables the deployment and management of the respective component. These operations are either deployment operations, which can be deploying an application component on an application server, or management operations such as scale, configure, upgrade, etc.[26]. The former is implemented through deployment artifacts, which are needed to realize the actual component, while the latter is implemented through implementation artifacts [29]. TOSCA allows artifacts of various types. For instance, a node type may have a deployment operation implemented with a WAR file, and at the same time, a management operation implemented in Ansible. Therefore, each component defines, implements, and maintains its deployment and management in a self-contained way. As a result, each defined type is independent of concrete cloud providers, fulfilling second sub-goal of TOSCA.
  - Templates are instances of a type and are the main building blocks of the topology graph, representing either an application component, in the case of a node template, or a relationship between two different application components, in the case of a relationship template.
  - Instances depend on the runtime of TOSCA and represent the actual instances of templates. A template can be realized many times in a topology.

- Management plans: Management plans express higher-level management tasks and combine a series of deployment and management operations from various topology graph nodes and relationships, as well as external services. Management plans, in general, specify how to manage the associated service template throughout its lifetime [26],[27] having three different types: Build, modification and termination. In this manner, application creators can incorporate best management practices and reoccurring tasks into management plans, ensuring that best practice

knowledge is widely used. Furthermore, management plans relieve enterprise IT of the burden of management knowledge while concealing most of the technical details of management best practices. In this context, TOSCA does not introduce a new workflow language to set up management plans but instead recommends an existing workflow language such as BPMN or BPEL. In this way, TOSCA enhances interoperability and portability between different engines while also shortening the learning curve required to use a new modeling tool. Aside from the explicit modeling of a management plan directly from the application creator, TOSCA also supports another approach, in which the management plan is generated automatically if the topology graph contains enough semantics. This shifts the encoded semantic information from the plan to the topology graph.

During the execution and management of the cloud application, the TOSCA runtime environment must have access to the Service template and all related artifacts. TOSCA ensures that these files are available by defining a standardized archive format known as "Cloud Service Archive" (CSAR). A CSAR is a compressed zip file [30] containing files of various types (e.g., implementation and deployment artifacts, scripts, etc.) organized in hierarchical sub-directories specific to a given cloud application, where Definitions and TOSCA-Metadata are two required subdirectories. The former contains one or more Definitions.tosca documents that describe the cloud application, such as the Service Template file, whereas the latter contains the TOSCA.meta file, which describes the metadata of the CSAR and the packaged files. All TOSCA-compliant runtime environments can deploy CSAR packages, ensuring portability between different TOSCA-compliant environments. There are two methods for processing a CSAR package. The first approach is imperative processing, and it involves the TOSCA engine deploying the cloud application based on the workflow specified by the topology modeler. The second method is declarative processing, and it involves the TOSCA engine attempting to extract a valid workflow from the topology graph.

Figure 2.2 depicts the topology of an application in which a Nginx server is deployed on OpenStack infrastructure.

```yaml
tosca_definitions_version: tosca_simple_yaml_1_3

imports:
  - library/nginx/main.yaml
  - library/openstack/main.yaml

topology_template:
  node_templates:
    nginx:
      type: nginx.Server
      requirements:
        - host: vm

    vm:
      type: openstack.VM
      properties:
        name: nginx_host
        image: 9ea4856a-32b2-4553-b408-cfa4cb1bb40b
        flavor: d3046a41-245a-4042-862e-59568e81f8fa
        network: 753940e0-c2a7-4c9d-992e-4d5bd71f85aa
        security_groups: default
        key_name: tadej_borovsak

    site:
      type: nginx.Site
      requirements:
        - host: nginx
```

Figure 2.2: TOSCA service template

In the first line, we specify the TOSCA version to be used, and in the *imports* section, we include the node types required to form the desired topology. We define a topology with three nodes in the topology templates section:

1. The Nginx server application component is represented by the node *nginx*, with the requirement to host the provisioned server on another component called *vm*.
2. The node *vm* represents a provisioned virtual machine on the Openstack infrastructure, and we define its characteristics.
3. The node *site* represents the site's contents, which will be uploaded to the provisioned Nginx server.

2.1.2.2 Other Provisioning tools

OpenStack [31] is a free and open-source cloud IaaS platform that enables rapid deployment, management, and development of an IT infrastructure's computing, storage, and network resources. Heat [32] is one of the leading software components of Openstack and automates the deployment of infrastructure and applications in OpenStack Clouds. Heat uses template text files to describe the infrastructure for a cloud application in a human-friendly way and supports a variety of infrastructure resources such as servers, volumes, security groups, etc. Furthermore, the user can specify the relationships between the different resources of the infrastructure and potential dependencies between them. Heat can identify the implicit and explicit dependencies between the resources and provision the corresponding infrastructure components in the correct order. Another important point is how easily the user can update or change the infrastructure state completely by simply modifying the template file. Heat receives the updated template files and performs all the necessary steps to adjust the current infrastructure to the desired state, including deleting existing resources. It also provides a cross-compatible AWS Cloud Formation implementation to run existing Cloud Formation templates on OpenStack [33].

AWS CloudFormation [34] is a provisioning tool that automates infrastructure deployment in the AWS Cloud. Users use JSON or YAML-compliant template files to describe a collection of AWS resources, known as a stack. CloudFormation uses a declarative approach, which means that the platform deploys the requested resources and manages their dependencies without the user's intervention. Furthermore, it provides a preview of how proposed changes will affect already provisioned infrastructure resources and allows you to roll back to previous versions of the deployed infrastructure if errors are detected.

Terraform [35] is a free and open-source orchestration tool for infrastructure provisioning, scaling, de-provisioning, and operations. It supports multiple cloud service providers, can integrate other third-party services, and provides a wide range of features for dealing with extensive infrastructure for complex distributed applications. Terraform primary function is to describe the desired infrastructure resources in configuration files using a domain-specific language known as HCL. The platform then generates an execution plan, which details the exact steps that will be taken to create the desired infrastructure topology. Terraform also creates a graph of infrastructure resources and identifies the dependencies that exist between them. As a result, if the user accepts the generated execution plan, the platform executes the necessary operations to deploy the resources and can parallelize the creation of non-dependent resources. It also has the advantage of storing the state of the deployed

infrastructure in a separate file called terraform.tfstate, which can be easily shared and used to recreate the same infrastructure elsewhere instantly.

Parameters	**Terraform**	**CloudFormation**	**Heat**	**TOSCA**
Language	DSL(HCL)	YAML,JSON	DSL(HOT)	YAML
Language Type	Declarative	Declarative	Declarative	Declarative
Cloud Platform	Multiple	AWS	Openstack,AWS	Multiple
Source Code	Open-source	Closed-source	Open-source	Open-source
Interface	CLI	GUI	CLI	GUI, CLI

Table 2.2: Comparison of provisioning tools

Table 2.2 provides a quick overview of the investigated provisioning tools, allowing the reader to identify differences and similarities for five different parameters:

- The configuration language provided by each tool: Terraform and Heat use Domain-Specific Languages to provision resources, whereas TOSCA and CloudFormation use the YAML syntax. Furthermore, when writing provisioning scripts, CloudFormation supports the JSON syntax.
- The provided language's style: All the examined provisioning tools follow the declarative programming paradigm, which means that the developer states the system's desired state and the tool determines how to achieve the requested state.
- What cloud platforms are supported: Terraform and TOSCA support resource deployment across multiple cloud providers, whereas CloudFormation is vendor-locked and only supports Amazon AWS. Furthermore, Heat only supports Amazon AWS and private cloud infrastructure via OpenStack rather than a wide range of cloud providers.
- How the tool's source code is distributed.
- The type of user interface to which the user has access: CloudFormation and TOSCA provide a GUI that allows developers to write scripts more quickly, whereas Terraform and Heat provide only the option to write code via a terminal(Command Line Interface).

2.2 IaC Research Projects

In the previous section, we presented just a few of the most popular IaC tools and languages, and new technologies are being developed daily. However, this wide range of tools and technologies makes it difficult for practitioners to select the appropriate ones that meet their needs. Because no tool dominates the market and supports the IaC development and deployment procedure end-to-end, practitioners typically use a set of IaC tools to cover all aspects of IaC [36]. To make the process of selecting appropriate technologies easier, the authors of [37] performed a systematic mapping review and categorization on IaC-related tools and catalogued the topics studied in infrastructure as code-related publications.

Furthermore, the authors of [38] conduct a systematic literature review of DevOps concepts and DevOps tools, including deployment automation tools like Chef or Puppet, whereas [39] catalogued cloud deployment modeling languages and identified their primary purposes for each one. These studies confirmed that, despite the abundance of frameworks, tools, and languages, the field of IaC is still quite fragmented. As a result, it is pretty challenging to select the appropriate technology, compare the disparate features and mechanisms, and migrate from one technology to the other due to the domain knowledge required. These challenges put significant barriers to the widespread adoption of IaC and DevOps in general.

To address these concerns, the authors of [40] propose a deployment metamodel that abstracts from the specific complexities of each technology and introduces a general overview of all the essential parts that exist between different declarative deployment technologies. This enables researchers working with various IaC technologies to share a common understanding and develop technology-agnostic models. As a result, migration between environments becomes more accessible because the user can identify commonalities and changes that must be made. Furthermore, the metamodel and the standard features of the technologies pave the way for exploiting MDA methods that will automatically transform the deployment artifacts from one environment to another, making application migration an automated procedure.

Based on this concept, the same team created TOSCA Light [41], a TOSCA subset used to develop technology-agnostic deployment models that can then be translated into a concrete IaC deployment artifact. In this manner, the developer creates an application deployment model only once and then reuses it through model transformations into all EDMM framework-supported deployment technologies, enhancing portability between heterogeneous environments.

Another kind of model-driven approach is RADON [42]. RADON proposes a DevOps framework for developing and managing cloud applications that utilize the microservices architectural style and the serverless Function as a Service (FaaS) paradigm. The user controls the autoscaling operations performed at the function level when using this framework, and the cloud application can be deployed across different cloud providers with heterogeneous capacities. Regarding the modeling language, RADON extends TOSCA with a constraint definition language (CDL) to specify temporal behavior as well as functional and non-functional requirements of a cloud application. Using TOSCA as the baseline modeling language allows professionals already familiar with TOSCA to learn this new modeling environment in less time. Another important aspect is that TOSCA-compliant tools, such as Winery, can be easily integrated into the framework and enrich the modeling environment at no additional cost to the RADON developers. After defining the application model and creating the TOSCA templates, the user can use the provided Verification and Defect Prediction tool to analyze, verify, detect, and correct defective IaC blueprints and incorrectly defined requirements.

Goal of RADON is to provide a solution that allows developers to effectively manage the entire lifecycle of a complex application that adheres to the serverless and microservices paradigms. However, while it provides quality assurance tools for checking IaC code for code smells and anti-patterns as well as serverless functions or microservices for business logic errors, it does not provide any quality review for configuration management scripts imported into TOSCA templates. Furthermore, it does not provide features that aid in developing configuration management scripts; instead, RADON assumes that they were created with an external tool.

An interesting model-driven project is ARGON [43], a framework for modeling the cloud resources that an application will require when distributed across many cloud providers. To accomplish this, ARGON provides a Domain Specific Language that allows users to abstractly represent the needed infrastructure without relying on a specific IaC language. As a result, DevOps professionals spend less time dealing with an extensive DevOps toolset and more time modeling the required infrastructure with a DSL, which abstracts the specificities of each tool and reduces modeling complexity. Moreover, after defining the infrastructure model, ARGON performs model to model and model to text transformations to build concrete scripts, such as Ansible or Puppet, which are subsequently deployed and executed on a cloud platform. Following this approach, the user may model the required infrastructure without expert knowledge in a specific configuration/provisioning language, significantly automating the infrastructure management and facilitating the communication between different DevOps professionals.

One thing to keep in mind is that ARGON has a limited expressive capability because it is limited to the supported abstract ARGON elements [44] and their transformation rules. Assume a developer wants to write complex Ansible scripts that use multiple Ansible modules to perform a variety of software configurations. In that case, due to ARGON's element library limitations, it may not be possible to do it entirely with ARGON, as it may not generate the desired Ansible modules. As a result, there is a tradeoff between development simplicity due to abstraction from technical details and tool expressiveness.

DICE [45] is another model-driven approach that enables users to create language-independent models that are then translated into concrete IaC scripts in the context of Data-Intensive Applications(DIA). DIA typically necessitates complex infrastructures, making it challenging to deploy, configure, and operate one, especially when the user wishes to combine more than one framework. To address these issues, the authors propose a Domain Specific Language (DSL) based on UML with a high level of abstraction, in which the user can model the provisioning, deployment, and configuration of a DIA without having to worry about the specific technical details for each framework. It also includes a library of standard IaC components, such as Hadoop, Storm, and Spark, for deploying DIA, as well as an open-source tool called DICER, which takes the defined abstract model of the DSL as input and generates runnable IaC scripts, which are written in Chef or TOSCA. DICER aims to allow users to create IaC in parallel with software development by unifying common abstractions between IaC and software design and enabling rapid deployment, testing, and redeployment by automating the procedure to generate concrete, deployable IaC scripts. Furthermore, because of the level of abstraction provided by DICER regarding the infrastructure, a user can exploit pre-existing IaC components without knowing the technical details behind them.

In general, DICE implements the same ideas as SODALITE, where there is a level of abstraction from the selected IaC languages. However, like ARGON, there is the limitation that the users cannot develop infrastructural code outside the scope of the provided library. As DICE focuses on DIA, it contains a specific set of reusable components of data-intensive architectures for TOSCA topologies and Chef recipes, supporting the most popular frameworks. This limitation is also reported by the writers of DICE with the intention to extend the provided library in the future.

MELODIC [46] follows another direction and exploits the concept of utility functions. MELODIC delivers powerful autonomous middleware that acts as automatic DevOps for a specific cloud application and changes the necessary deployment configuration at runtime. This framework aims to enable multi-cloud application deployment while also providing additional flexibility, capacity, and elasticity

during runtime. The suggested framework selects an appropriate initial deployment for the cloud application, then monitors its condition, and performs the necessary adaptations throughout its entire life cycle. These adjustments are either optimizations in the application's configuration or its redeployment, which includes the provision of new infrastructure. This method adheres to the concepts of autonomic computing, in which the application can self-configure, self-heal, self-optimize, and self-protect without the need for human interaction. As a result, the deployment and management of a cloud application become a fully automated activity, where even the cloud providers that will host the application are selected automatically based on defined requirements.

MELODIC uses the CAMEL language (Cloud Application Modeling and Execution Language) to describe the application's requirements and infrastructure, similar to TOSCA but enhanced with the ability to support the specification of instances at run-time beyond the definition of types and templates at design-time. CAMEL, like TOSCA, necessitates the use of implementation scripts to provide the necessary resources and configure the appropriate software. However, MELODIC does not support the development of such implementation scripts within the framework; instead, the user must create them with an external editor and then import them into the application model.

PIACERE [47] is an ongoing project whose primary goal is to simplify the development of infrastructure code, increase its quality, security, and reliability, and provide autonomic computing features such as self-healing, self-configuration, and self-optimization that ensure its uninterrupted business continuity. To accomplish these tasks, PIACERE provides a framework that DevSecOps [48] professionals can use as an end-to-end solution, from modeling the required infrastructure and specifying the functional and non-functional requirements to deploying the application on the defined infrastructure and monitoring it during runtime. These activities are aided by the modular approach of PIACERE, which combines multiple tools within the same framework. The core is an IDE that allows developers to create infrastructure code for various infrastructure environments using a domain-specific language called DOML, which hides the peculiarities of each environment through abstraction layers. After completing the infrastructure model, PIACERE runs the Verification Tool, which checks the model for errors and inconsistencies. If the model is verified, a component called Infrastructure Code Generator is used to generate concrete IaC scripts in various IaC languages such as Terraform, TOSCA, Ansible, etc. Another valuable feature of PIACERE is testing infrastructure code in a simulated environment and performing checks to identify potential vulnerabilities and bottlenecks before the code is deployed. In addition, during the pre-deployment phase, PIACERE optimizes application deployment by determining the best

combination of available services (IaaS, XaaS), infrastructure resources, and defined requirements. PIACERE also monitors the entire IaC execution run in real-time and provides insightful and continuous feedback to the DevSecOps team on security, deployment, execution, and provisioning. These data are fed into PIACERE's autonomic computing mechanisms, which self-adapt the deployed application in order to meet the defined QoS conditions at all times.

SODALITE is a European project that provides an environment to support the definition of complex deployment models across heterogeneous resources and frameworks and simplifies application deployment modeling and execution across multiple, heterogeneous infrastructure resources. Our work lies in the context of SODALITE, and in order to give the readers a clear view of our contribution, we describe the main concepts and purpose of SODALITE thoroughly.Mainly, we will focus on the components of SODALITE related to our work and its modeling aspects. The references that we will exploit are the official website of SODALITE [13] and [14].

SODALITE provides a robust toolkit that aids in deploying an application throughout the stages of its modeling procedure. TOSCA and Ansible are key components of this toolset, but not in their pure form. SODALITE provides Domain-Specific Languages that adhere to the principles of TOSCA and Ansible but differ in critical ways to simplify deployment models and increase modularity. DevOps engineers can define deployment models by using an abstraction of TOSCA that is then translated into TOSCA. Two Domain-Specific Languages (DSLs) have been implemented to accomplish this: Resource Model DSL and Abstract Application Deployment Model DSL. The provided DSLs decouple the definition of an application model from the resources that it will use. When developing a TOSCA-based model, these concepts are bound together, whereas SODALITE creates two different development environments to support each development procedure separately and more efficiently.

Application Ops Experts (AOEs) can define application topology models using the AADM DSL, which is represented as a connected graph of application components. AOEs define node templates in the same way they do in TOSCA by instantiating imported node types, stating their relationships, and declaring their requirements. Another model, called the Resource Model, defines the node types and their relationships. A Resource Model represents infrastructure resources and contains the building blocks of an application topology model (AADM). The so-called Resource Experts (RE) are in charge of creating RMs and modeling classes of infrastructure resources as well as their capabilities, requirements, interfaces, and relationships. The completed RMs can then be uploaded to Knowledge Base of SODALITE and made

available for import into an AADM. As a result, the two types of models remain distinct, though closely linked. This concept is critical when new technologies emerge because it enables AOEs to exploit novel infrastructure resources uploaded from REs for their application. After submitting an AADM for deployment, SODALITE transforms the model and generates an equivalent TOSCA blueprint ready for execution on SODALITE's orchestrator [49].

Each operation defined in an RM must be accompanied by a corresponding implementation script that performs the required configuration and management tasks. SODALITE provides an Ansible abstraction called Ansible DSL, through which REs can define Ansible Models. Ansible DSL adheres to and supports the same conceptual attributes as Ansible, but it groups several attributes that are 'included' in the same semantic category to simplify user interaction and better organize the code. The Ansible Models are transformed into concrete Ansible scripts, which can be associated then with operations of an RM, integrating Ansible with TOSCA.

The Optimization Model DSL is another DSL provided by SODALITE, in which Optimization Experts (OEs) design optimization recipes that are associated with an AADM. Then, when the AADM is submitted for deployment, an application optimizer goes through the optimization settings and applies them to the target hardware.

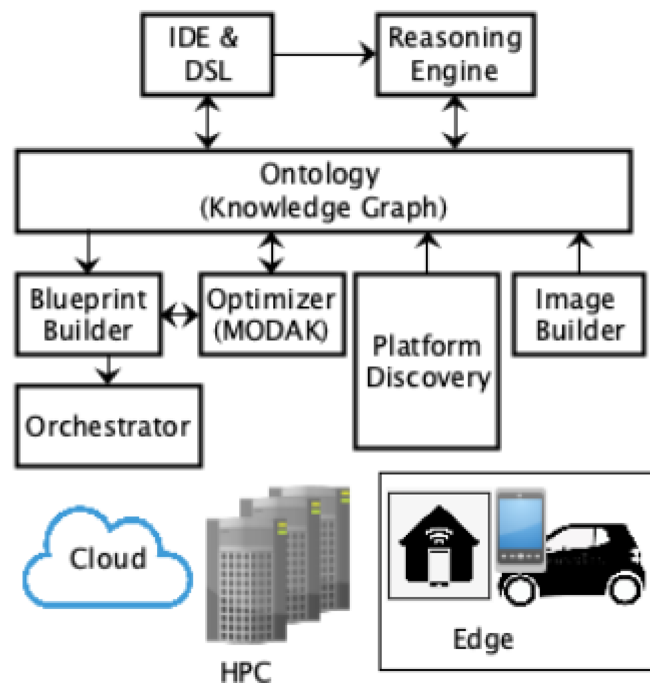


Figure 2.3: SODALITE overview

Modeling capabilities of SODALITE are delivered to end-users through a user interface known as SODALITE IDE. Users can use a textual and graphical editor to create AADM, as well as a textual editor for all other DSL. Textual editors provide a wide range of advanced features that aid in developing the cited models. For example, they provide content assistance mechanisms that propose elements at each stage of the development process to assist in the development of correct implementation scripts not only syntactically but also semantically. They also provide syntactic and semantic validation mechanisms, which alert the user about code errors and prevent the submission of deployment models with faulty properties.

However, the IDE depends on other components to serve some of the features and function properly. Figure 2.3 depicts a high-level overview of SODALITE architecture. SODALITE incorporates an ontology layer that maps the defined models into entities of a 3-tier ontology graph, implemented with GraphDB. The Reasoning Engine interacts with the ontology layer and performs logic-based inference, presenting the extracted information to the IDE through a RESTFUL API. The Blueprint Builder is in charge of generating concrete TOSCA artifacts from submitted AADM, interacting with the Optimizer taking into account associated Optimization Models, while the Orchestrator deploys the generated blueprint and manages the lifecycle of the application. Finally, the Image Builder creates runtime images that are stored in the image registry, and the Platform Discovery assists REs in the definition of node types by collecting information for the underlying infrastructure environment such as HPC or OpenStack.

2.3 Research Gaps and Contribution

This chapter introduced a plethora of IaC tools for resource provisioning and configuration management. Furthermore, we investigated cutting-edge research projects aimed at facilitating IaC adoption. At this point, it is critical to report where our contribution lies in relation to these research efforts and tools by making some observations. The research projects under consideration seek to address two significant issues in the field of IaC: the large number of tools and the enhancement of support of cloud applications that exploit specific programming paradigms.

In the first case, the large number of IaC tools with varying structures, syntax, and properties, combined with the lack of a dominant tool on the market, make selecting the appropriate one, time-consuming with ambiguous outcomes. To tackle this challenge, research projects like ARGON, DICE, and PIACERE aim to add abstraction layers to selected IaC languages (TOSCA, Ansible, Chef) in order to remove specific technical details, reduce the learning curve for different languages,

and lessen the effort required when a developer migrates the same script between languages. In this way, the developer only needs to write one abstract IaC script, and the framework will translate it into concrete IaC code, increasing flexibility. However, despite the simplicity that the abstraction layers offer to the users, one significant weakness is the limitations of the provided translation mechanisms concerning the supported target languages. Moreover, the abstraction layers hide some language-specific details from the resulted IaC script, reducing the expressiveness of the tool. Such an example can be the usage of Ansible modules that are not supported by the translation mechanisms and therefore are not included in the final, concrete, Ansible script.

Regarding the second issue, some research projects, such as DICE and MELODIC, focus on a specific set of infrastructure resources (e.g., FaaS) and a specific phase of the lifecycle of the IaC (provisioning, deployment, configuration). As a result, users have to exploit external tools in order to automate all the steps of the deployment procedure of an application and different tools for different kinds of applications. For example, in the case of MELODIC, the user has to develop with an external editor the implementation scripts that are necessary for the realization of the defined resources. Alternatively, in the case of DICE, the supported set of infrastructure resources is limited to those related to popular Big Data frameworks.

As we can conclude, numerous tools, languages, and frameworks focus on resolving a specific issue while failing to provide an end-to-end solution for various types of applications. Thus, in many cases, users must employ multiple tools to complete the general task of properly deploying a relatively complex application across heterogeneous resources and ensure the expressiveness of the deployment model. To address this challenge, the SODALITE project was created to provide a method and tools to make users' lives easier. SODALITE focuses on assisting users in the creation of TOSCA artifacts that define the overall structure of an application from a deployment standpoint. This approach, however, is insufficient to automate the deployment tasks from start to finish entirely. The reason is the assumption that the implementation scripts used to realize the defined components were written outside of the SODALITE framework and related to the Resources and Application Deployment models in an implicit way.

Our work aims to fill this gap by complementing the work done in [50]. We enhance the existed development environment with features that assists users in writing Ansible scripts and guides them in relating such scripts to other parts of the deployment specification, such as resource provisioning, thereby facilitating end-to-end modeling of a cloud application from the definition of each deployment topology characteristic to the implementation scripts that realize its management and

deployment operations. The main goal is to create a hub where various information sources can be brought together, and pre-existing elements can be reused (Ansible collections, Ansible modules, Ansible roles, TOSCA models). Users can also utilize the provided validation and defect prediction mechanisms to ensure the correctness and quality of the Ansible scripts.

In comparison to state-of-the-art, we are aware of no other tool on the market that provides all these features in one place and assists users in such a wide range of aspects. These features enable the end-users to reduce their development effort, thus allowing them to use their time and resources in other activities and saving costs.

3. Problem Definition

This chapter presents the thesis' objectives, explaining the problems we attempt to solve and providing a general idea of the proposed solutions without delving into technical details. Section 3.1 introduces our work's main goal and breaks it down into smaller, more concrete objectives. Section 3.2 provides an overview of the Ansible editor's initial version, while Section 3.3 discusses the TOSCA and Ansible integration issue in the context of the Ansible editor. Section 3.4 describes how to assist users in selecting the appropriate Ansible content, and Section 3.5 provides a preview of the new features we will implement.

3.1 Main Goal

This thesis aims to support the adoption of the IaC paradigm as a DevOps practice by bringing together two popular IaC tools, TOSCA and Ansible, to facilitate the writing of IaC code and increase practitioner productivity. In addition, modern development environments, such as IntelliJ [51] and VisualStudio [52], serve as inspiration for our work because they provide advanced features to developers such as code completion, language support, integrated plugins, etc.

Our work will be done within the context of the SODALITE IDE and will extend the functionality and capabilities of the already implemented Ansible editor, which laid the groundwork for an innovative and supportive IaC development environment. As described in Chapter 2, SODALITE allows users to model TOSCA node types and include Ansible scripts that implement each node type's various deployment and management operations. However, the RE had to use an external editor to write the necessary Ansible script before the Ansible editor was released [50]. As a result, incorporating this editor into SODALITE added significant value and allowed for a more tightly coupled relationship between these two powerful tools.

Having the current Ansible editor as a starting point, we aim to enrich its capabilities and facilitate the end-to-end modeling of a cloud application, from the definition of each characteristic regarding the deployment topology, to the implementation scripts

that realize its management and deployment operations. To accomplish this task, we divided it into four smaller objectives:

1. Facilitate the combination of TOSCA and Ansible further and provide the necessary environment for the seamless integration of these two IaC tools.
2. Extend the features of the already implemented Ansible editor and make meaningful changes to Ansible DSL that will enhance the user's experience.
3. Integrate Ansible content from various resources and repositories into Ansible editor.
4. Assess the quality of Ansible content and create a meaningful subset that will bring value to the SODALITE framework in a secure and quality way.

3.2 Ansible editor overview

The current Ansible editor incorporates a content assistant component that provides valuable visual suggestions to the end-user while developing an abstract Ansible model. More specifically, it supports the following features:

- Syntax errors identification: The Ansible editor can detect syntax errors made by the user, such as misspelling a keyword or using the wrong variable name, and triggers a detection mechanism to notify the user of the errors by underlining the corresponding text in red.
- Keywords' meaning: The addition of a new level of abstraction in comparison to Ansible requires Ansible editor users to learn and understand the syntactical and semantic differences between the two languages. To aid in this process, the Ansible editor displays clear indications for each keyword and explains the logical meaning of each keyword option and the attributes that can be defined in it in a small window.
- Declared variables suggestions: On numerous occasions, the user must refer to a variable that holds a specific value, which must be used at another point in the code. In this case, the user must traverse the entire text to identify the appropriate variable to be referenced, which may result in errors and time waste, especially if the code is long and complex. To make this process easier, the Ansible editor suggests available variables that can be referenced directly and displays the type of variable as well as its name identifier.
- Suggest information from local Resource Model: As we described in chapter 2, a TOSCA operation belongs to a TOSCA interface, which in turn is contained inside a TOSCA node type. Each abstract Ansible model

aims to implement the logic behind a TOSCA operation and perform the necessary tasks to deploy or configure the corresponding node type. Still, the Ansible model may require some information as inputs from the corresponding RM, which can come either from an interface or from an operation. For this reason, the Ansible editor allows users to reference these inputs from the locally connected RM and exploit them as if they were variables declared in the Ansible model.

The Scope Provider is another critical component of the Ansible editor. This component provides hyperlinking, a feature that allows you to navigate between references. For example, if a user declares an entity of any type in the Ansible model (defined variable, registered variable, handler, topic, 'fact set' variable) and then refers to it elsewhere in the code, he/she can directly jump to the corresponding definition of the entity using hyperlinking. Furthermore, it supports cross-references between entities from different models. In SODALITE, this means that an Ansible model can refer to elements contained within a local RM. Hyperlinking is especially useful because the user has direct access to all the entities and files associated with an Ansible model and does not have to waste time searching for the location of the corresponding file or the definition of the entity within the code, primarily when the code extends in hundreds of lines.

One of the most critical features of the Ansible editor is the generation of concrete Ansible playbooks from an abstract Ansible model. A component called Generator performs the necessary steps by taking an abstract syntax tree (AST) as input and generating concrete Ansible playbooks ready for deployment in the orchestrator. The abstract syntax tree represents the Ansible model's abstract syntactic structure and is generated during parsing, storing it in memory for later processing. The Generator component can process the AST to create valid, concrete Ansible playbooks, which can be included in the RMs. The user defines the necessary information for generating concrete Ansible scripts within the abstract Ansible model in a self-contained manner, which means that the Generator can produce valid playbooks without the need to retrieve information from other resources, greatly simplifying the entire procedure.

3.3 TOSCA and Ansible integration

The integration of Ansible and TOSCA is one of the main aspects of this project and one of the main features of the Ansible editor. Currently, the Ansible editor supports suggestions and cross-references between local RM and Ansible models, allowing the user to refer to inputs defined by a TOSCA interface or operation. However, this approach only supports RM stored in the developer's file system, which may be a

problem when different people develop a RM and its implementation scripts simultaneously. For example, in Figure 3.2, the developer has imported the node type *sodalite.nodes.AWS.Keypair* into the Ansible model from a local Resource Model. In order to use this Resource Model file from different developers, it is necessary to distribute it via external communication channels, adding overhead to the development procedure. With this in mind, we must use every available tool at our disposal to facilitate the development process. As a result, we thought it would be beneficial to link SODALITE's Knowledge Base with the Ansible editor. In this way, Ansible developers will have direct access to TOSCA files stored in KB (Figure 3.1) and will be able to reference information without downloading these files. Furthermore, distributing the same file among different developers may result in deviations from the original version, impacting the developed Ansible scripts. Thus, referring to the same RM through a centralized knowledge base eliminates this risk and ensures consistency within the same group of practitioners.

KB Content	Description
RMs	
docker	
docker_certificate.rm	
docker_registry.rm	
docker_component.rm	
docker_volume.rm	
openstack	
openstack_vm.rm	specification of resources for Openstack VM
openstack_security_rule.rm	
snow	
snow_v2.rm	

Figure 3.1: Content of the Knowledge Base

```
1  playbook_name:"Create keypair"
2  used_by:
3      node_type: "sodalite.nodes.AWS.Keypair"
4      interface: "Standard"
5      operation: "create"
6  plays:
7      play:
8          collections:
9              -amazon.aws
10         tasks_list:
11
12             task_to_execute:
13                 task_name: "Create EC2 key pair"
14                 module: ec2_key
15                     parameters:
16                         name: "EC2 key"
17                         region: {{operation_input: "region"}}
18                         aws_access_key: {{operation_input: "aws_access_key"}}
19                         aws_secret_key: {{operation_input: "aws_secret_key"}}
```

Figure 3.2: Ansible Model importing a node type from a local RM

Along with the definition of a RM, the RE must include an implementation script for each defined deployment or management operation. To accomplish this, the RE can either use an Ansible script written with an external editor or an Ansible script generated from an abstract Ansible model defined with the Ansible editor of SODALITE. Nonetheless, suppose the RM contains many node types and operations, including all of the different Ansible files in the RM. In that case, it can be quite an error-prone and time-consuming task that can result in unexpected behavior at runtime. For this reason, it would be helpful to provide some guidelines to the end-user to create a clear picture of what the next step would be and simplify the management of the Ansible scripts for each operation. For example, the Ansible editor can provide Ansible script templates for each operation as a starting point for further development, indicating to the user that extending these scripts should be the next development step. Furthermore, grouping Ansible files (Ansible abstract models, Ansible scripts) by interface and node type matches Ansible scripts to specific TOSCA operations, establishing the prerequisites for a structured development environment. These types of mechanisms will be discussed in greater detail in Chapter 4.

In this context, it is essential to maintain the portability of RM stored in KB between different users. A RE includes the required Ansible playbooks within a RM by setting their path in the local file system. With this approach, when a RM is stored in the KB, it depends on the local file system, where the RM was developed (Figure 3.3). For instance, if a RE writes a RM that includes multiple Ansible scripts, then when another user retrieves that specific RM from the KB, he/she will have to manually

replace all the paths that refer to each Ansible script and are dedicated to another folder structure, wasting much time in unnecessary errors. As we can see from the example in Figure 3.3, there is an error for the *delete* operation's implementation script due to its dependency to a local file system that does not exist in the current system. Therefore, we must devise a mechanism to ensure interoperability among machines that download the same RM. This change will benefit not only experienced SODALITE users but also newcomers because when they download RMs from KB to familiarise themselves with the environment and the TOSCA models, they will not have to deal with errors right away, which otherwise will make their first experience with the framework unpleasant. Another thing that limits the portability of the RMs is that when a user retrieves an RM from the KB, the related Ansible files are not accessible. Thus, the user must find these Ansible files through other channels, if this is possible, creating problems to the development procedure.

```
131        delete:
132          inputs:
133            id:
134              type: string
135              default: get_attribute:
136                entity: SELF
137                attribute: aws/sodalite.nodes.AWS.VM.id
138            aws_access_key:
139              type: string
140              default: get_property:
141                entity: SELF
142                property: aws/sodalite.nodes.AWS.VM.aws_access_key
143            aws_secret_key:
144              type: string
145              default: get_property:
146                entity: SELF
147                property: aws/sodalite.nodes.AWS.VM.aws_secret_key
148            vm_name:
149              type: string
150              default: get_property:
151                entity: SELF
152                property: aws/sodalite.nodes.AWS.VM.name
153          implementation:
154            primary: "/workspace/iac-modules/aws/playbooks/vm_delete.yml"
155
```

Figure 3.3: Error after retrieve RM from KB

3.4 Ansible Content

Ansible modules and roles are two of the essential aspects of Ansible. They are primarily reusable Ansible content that allows the distribution of execution logic among different users. A module is a discrete unit of code that performs a specific operation, whereas a role is a grouping component with a set of tasks that reuse

common configuration steps. Furthermore, in Ansible 2.8, a new concept called 'Ansible collection' was introduced, a package of Ansible content that can include playbooks, roles, modules, and plugins. These elements enable the user to write playbooks in a clean and readable manner, avoiding the need to rewrite code for tasks that have already been implemented.

Ansible collections and roles can be shared among users via Ansible Galaxy, the official Ansible content hub. Users can publish their Ansible artifacts in Ansible Galaxy and provide documentation for them, in which the purpose and functionality of each artifact, as well as its requirements, are described. However, as Ansible has grown in popularity, the number of available collections and roles stored in Ansible Galaxy has increased significantly, and many of them overlap in terms of purpose and functionality. This complicates the selection of the appropriate artifact because the developer does not know which one is best suited to his/her playbook. Therefore, the documentation quality is a good strategy [53] for selecting the right Ansible artifact, which must include a quickstart tutorial for installation as well as a detailed reference guide for the contained elements, such as modules, plugins, etc. Beyond documentation, the developer should look at the Ansible artifact's issue tracker to see if the creator maintains the artifact regularly and fixes old issues and bugs that users report. Another criterion is the rating given to the artifact by other users regarding code quality, community score, and the number of downloads. When these metrics are high, we can assume that the Ansible artifact is trustworthy and serves its intended purpose, as described in the documentation.

As we can see, the selection procedure is time-consuming, and even after taking into account all of these criteria, the outcome may not be as desired. As a result, we evaluated the content in Ansible Galaxy and chose a valid subset of it to be used in Ansible editor, which we believe will meet the needs of SODALITE users and simplify the selection of the appropriate Ansible module or Ansible role. Our first criterion for selection is security. After submitting the corresponding TOSCA file, SODALITE executes the Ansible playbooks with xOpera orchestrator, as described in Chapter 2. Keeping this in mind, we must ensure that the running playbooks and, as a result, the imported Ansible content from Ansible Galaxy do not pose any security risks to the SODALITE infrastructure.

3.5 Ansible DSL and editor extensions

The Ansible editor, as described in Section 3.2, offers a wide range of features aimed at making it easier to develop Ansible models and thus create Ansible scripts with a lesser amount of effort. Syntax validation, autocompletion, content assistance, and hyperlinking are vital features that bring innovation and take the editor a step

further than other Ansible editors such as Atom. In this context, the Ansible editor focuses primarily on Ansible's syntactical concepts and provides the necessary guidelines for writing syntactically correct Ansible scripts without considering the semantics of each Ansible entity. In addition, the collaborative nature of Ansible in terms of content has resulted in the development of a large volume of Ansible collections, modules, and roles from various developers, which can be shared among the community, as described in section 3.4. At this point, we believe there is plenty of room for improvement in order to bring this content closer to developers when writing Ansible models, providing mechanisms that will exploit this information, and bringing advanced features to the editor.

The new features that we will implement will give users access to information from various repositories, allow them to search for relevant Ansible content, and receive continuous feedback on their choices. In addition, the suggestions and validation mechanisms presented to the end-user will aid in understanding the purpose of the various Ansible collections, modules, and roles, as well as their functionality and how to use them correctly.

To extend the capabilities of the Ansible editor, we will use the Xtext framework [54], which provides the necessary environment for implementing the desired features, such as the validator component, which is in charge of detecting errors in an Ansible model and coloring the corresponding area in red.

4. Solution Design

In this chapter, we discuss the design decisions we made to extend the framework's functionality, as well as the features and components we introduced within the SODALITE ecosystem to improve the user experience. We also present some of our work's implementation details. Section 4.1 provides a general overview of the Ansible components and presents the project's organization. Section 4.2 describes how TOSCA and Ansible are integrated into the context of SODALITE, whereas Section 4.3 explains how Ansible content was collected and organized into a structured MongoDB. Finally, Section 4.4 discusses the Ansible DSL and Ansible editor additions and extensions.

4.1 Ansible Editor Architecture and Code Organization

4.1.1 Xtext framework

Eclipse Xtext [54] is a free and open-source framework for creating programming languages and domain-specific languages. It provides all the required components for a complete language infrastructure, including a serializer, linker, code formatter, compiler, interpreter, and syntax checker. Furthermore, through various functionalities, Xtext provides the ability to create an editor for the designed language, which the Eclipse IDE supports. Syntax highlighting, suggestions, scope providers, wizards, and code generators are among the features available.

One of the most critical characteristics of Xtext is the integration with EMF and Eclipse. For each defined piece of code written with the DSL, Xtext generates an EMF model, which abstracts from the syntactic details of the DSL and is used later on by Xtext for validation, compilation, or interpretation. As a result, the textual model that the user defines is transformed and maintained by Xtext as an Abstract Syntax Tree (AST). The generated AST conforms to a specific metamodel inferred from the DSL's grammar defined in a language called Ecore. In addition, EMF contains a code

generation mechanism that transforms each entity in the Ecore metamodel into a concrete Java class, whereas each entity in the metamodel results from a grammar-defined production rule. Therefore, each element of the EMF model is an instance of one of the generated Java classes that implement Ecore's primary interface, called EObject.

Beyond the tools that allow the definition of a DSL, Xtext offers, through the integration with Eclipse, the ability to create powerful editors for the defined DSL with advanced features such as validation,auto-completion options, syntax checking, quick-fixes, hyperlinking, etc. Thus, developers are relieved of creating these features from scratch for an editor that supports the development of textual models in the provided DSL.

SODALITE IDE takes advantage of all these features and offers a collection of Xtext textual editors for different DSLs, aiming to model infrastructure resources, application deployment topologies, application optimization, and operation implementations [55].

4.1.2 Architecture

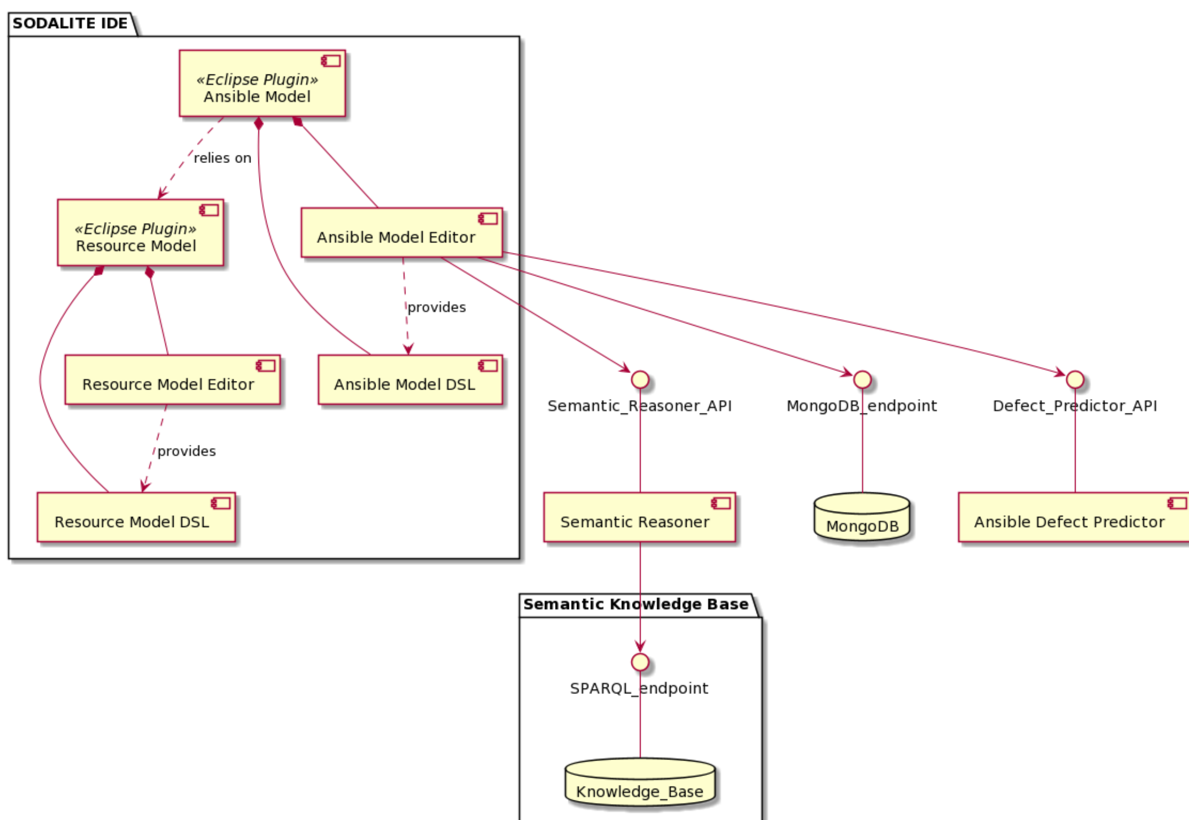


Figure 4.1: Architecture

Figure 4.1 highlights the elements of the Ansible editor and its interaction with the neighbor components. The Ansible editor is integrated into the SODALITE IDE as an Eclipse Plugin built with Xtext and uses Ansible DSL, which allows the users to create abstract Ansible playbooks that are then translated into concrete Ansible scripts. This Ansible Eclipse Plugin is based on the Resource Model Eclipse Plugin, which includes a DSL and a user interface for defining TOSCA reusable entities and their details such as attributes, properties, provided capabilities, and requirements. Ansible editor provides various services by interacting with four different software components: Semantic Reasoner, Knowledge Base, the MongoDB database, and the Ansible defect predictor, either implicitly or explicitly.

The semantic Knowledge Base is a semantic repository containing structured data from various domain aspects of SODALITE. SODALITE users (Application Ops Experts, Resource Model Experts, Quality Experts) can store and retrieve domain models from the KB. These are internally represented as RDF-based knowledge graphs. The Semantic Reasoner, an intermediary between the Knowledge Base and the SODALITE IDE, provides access to this knowledge. It consists of two components: the Semantic Reasoning Engine and the Semantic Population Engine. The former communicates with the Knowledge Base to retrieve information, whereas the latter populates the Knowledge Base with TOSCA entities defined by end-users. In addition, SODALITE IDE can communicate with the Semantic Reasoner via a REST API called Semantic Reasoner API, which allows it to import and retrieve data from the Knowledge Base and use its intelligent inference services.

The Ansible editor interacts with the Semantic Reasoner API to retrieve information about Resource Models stored in the Knowledge Base. These can be the various Resource Models stored in KB, the TOSCA interfaces and operations defined within a specific Resource model, and the data that each interface and operation inputs to the appropriate implementation script to perform the necessary tasks. The Ansible editor presents such information to the end-user to allow for the coherent and consistent development of Ansible models within the context of the chosen resource type and TOSCA operation.

The MongoDB database delivers Ansible content to Ansible editor end-users, including Ansible collections, modules, and roles gathered from various repositories in the Ansible Galaxy. To facilitate the development of an Ansible model, the Ansible editor communicates with the MongoDB database endpoint and serves the collected Ansible content to the end-user through suggestion and validation mechanisms.

Finally, the Ansible defect predictor is a module that identifies code smells and bugs in Ansible scripts and alerts the user to potential problems in their code. The Ansible editor communicates with the defect predictor via the Defect Predictor API, where it

sends Ansible scripts for analysis and receives the results, which then displays to the user via the user interface.

4.1.3 Implementation Structure

This subsection presents the structure of different projects related to Ansible DSL within SODALITE IDE and explains the changes and extensions we have made compared to the initial version.

The SODALITE IDE contains many Xtext projects that implement different DSL and their corresponding user interfaces in order to fulfill the requirements and purposes of the SODALITE project. In this context, our work mainly focuses on the projects related to Ansible and, more specifically, on three distinct components: the Ansible DSL, the Ansible editor, and the Resource Model editor. Our goal is to build upon the already implemented projects and provide meaningful extensions and features that will facilitate the development procedure of Ansible scripts.

In order to provide users with a better understanding of how the project is structured and where our contribution resides, we present the changes and additions we made in the Ansible-related projects of the SODALITE IDE and briefly explain the purpose for each one.

- org.sodalite.dsl.ansible:
  - AnsibleDsl.xtext: We extended the grammar of the Ansible DSL to support Ansible concepts such as namespaces, collections, modules, and role names not as strings only but as entities.
  - AnsibleDslGenerator: We changed the generation procedure of the concrete Ansible playbooks, which are now stored in the same folder as the corresponding, abstract Ansible model.
  - AnsibleHelper: We created a collection of reusable functions that are valuable for the functionality of the Ansible editor.
  - AnsibleDslValidator: We implemented many different validation mechanisms that alarm the user for errors in the abstract Ansible model.
- org.sodalite.dsl.ansible.ui:
  - AnsibleDslProposalProvider.xtend: We introduced multiple content proposal features that aim to facilitate the development of an Ansible model and provide clear guidelines to the users.
- org.sodalite.dsl.RM:
  - RMValidator.xtend: We added a validation mechanism to verify the paths of imported Ansible scripts.

    - RMGenerator.xtend: We extended this component to support the generation of Ansible models and Ansible scripts for each specified operation within the Resource Model.
- org.sodalite.dsl.RM.ui:
    - RMProposalProvider.xtend: We introduced suggestion mechanisms that facilitate the import of Ansible scripts into a Resource Model.
    - RMQuickfixProvider.xtend: We added a quickfix mechanism that corrects potential errors in the path of the imported Ansible scripts.
    - GenerationHandler.java: We added the option to generate Ansible files from a RM.
    - RMBackendProxy.java: This component was extended to implement the backend services that allow the generation of Ansible files from a RM.
    - RMHelper.java: We introduced some reusable functions required when importing Ansible scripts into a Resource Model.
- org.sodalite.dsl.kb_reasoner_client:
    - InterfaceDefinitionJsonDeserializer.java: We extended this component to deserialize all the information stored in the Knowledge Base related to a specific TOSCA node type, such as interface and operation data.
- org.sodalite.dsl.preferences:
    - SodaliteBackendProxy.java: We connected the IDE with the MongoDB, which contains Ansible Galaxy content, and the Ansible defect predictor, which identifies ansible code smells in the Ansible model.

More details about the briefly described will be presented in the following sections of this chapter.

4.2 TOSCA and Ansible integration

This section will go over the various mechanisms that we put in place to make it easier to develop Ansible models in the context of a Resource Model. The extensions we've introduced are integrated into the RM editor and the Ansible editor, and they allow for a more efficient exchange of information between an Ansible model and the related Resource Model.

As we described in Chapter 3, we mainly focus on three different aspects:

- Connection of Ansible editor with SODALITE KB: The goal of developing an Ansible model is to implement a TOSCA operation defined in a Resource model. However, during the development process, the user can only refer to the information contained in local Resource Models, with no access to models stored in SODALITE KB. As a result, the Ansible editor should communicate with the KB in some way and provide appropriate mechanisms for users to interact with its contents.
- Guided creation of abstract Ansible models: When a RE creates a Resource Model, it must associate each defined TOSCA operation with an implementation script containing the necessary business logic. However, managing all of the related implementation scripts in the case of a complex Resource Model with a large number of node types, interfaces, and operations is a difficult task. As a result, we believe it would be beneficial to enhance the development environment with features that help users organize and implement abstract Ansible models and the corresponding concrete scripts.
- Portability of Resource Models between different users of SODALITE: The association of a TOSCA operation with an Ansible script is achieved by defining the absolute path of the script in the local file system. However, this approach makes sharing Resource Models between different users a difficult task because each user who downloads a Resource Model from the KB must change all absolute paths that are incompatible with his/her local file system. Furthermore, when a user retrieves a RM from the KB, he/she does not have access to the Ansible files associated with it, which limits the portability, mainly when multiple practitioners develop the same Resource Model. For this reason, we present a mechanism that improves the portability of these models, allowing for the seamless exchange of Resource Models.

In the following subsections, we will detail the implemented features.

```
module: aws

node_types:
    sodalite.nodes.AWS.VM:
        derived_from: tosca.nodes.Compute
        interfaces:
            Standard:
                type: tosca.interfaces.node.lifecycle.Standard
                operations:
                    create:
                        inputs:
                            aws_access_key:
                                type: string
                                default: get_property:
                                    entity:SELF
                                    property: aws/sodalite.nodes.AWS.VM.aws_access_key

    sodalite.nodes.AWS.SecurityRules:
        derived_from: tosca.nodes.Root
        interfaces:
            Standard:
                type:tosca.interfaces.node.lifecycle.Standard
                operations:
                    delete:
                        inputs:
                            rules:
                                type: string
                                default: get_property:
                                    entity: SELF
                                    property: aws/sodalite.nodes.AWS.SecurityRules.rules
```

Figure 4.2: Resource Model example

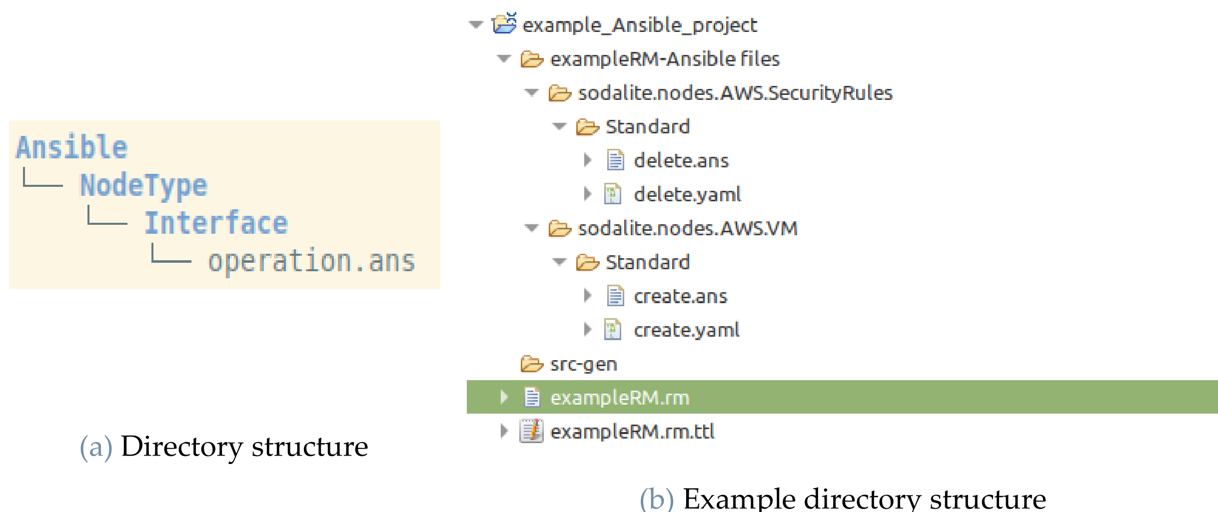
4.2.1 Guided creation of abstract Ansible models

The Resource Model editor has been enhanced to provide users with clear guidelines on which Ansible files should be created. To achieve this, the Resource Model editor creates a specific directory structure, where the implementation scripts are organized in a structured way, easing the management of different Ansible files, within the same project.

For each specified operation within the Resource model, the user can generate the abstract Ansible model and the corresponding Ansible script for further development. Initially, these files do not contain any specific content, and their purpose is to provide clear indications of which Ansible models should be created and for which operations. The Ansible files are generated in the 'Ansible' folder and follow the folder structure depicted in Figure 4.3a. A folder is created for each defined node type, and a folder with the name of each TOSCA interface is generated in the appropriate node type folder. Finally, for each TOSCA operation, a .ans file (Ansible model) and the corresponding .yaml file are generated and placed in the proper interface folder.

Figure 4.2 shows an example of a Resource Model named *exampleRM.rm*, and Figure 4.3: Folder organization shows the generated folder structure. We have two defined node types in the provided Resource Model: *sodalite.nodes.AWS.VM* and *sodalite.nodes.AWS.SecurityRules*. For each node type, two folders are generated with the same names (Figure 4.3b). Furthermore, each one of the node types contains an interface with the name *Standard*, and in each node type folder, another folder with the name *Standard* is generated. Every folder related to an interface contains the Ansible files for the operations that belong to that specific interface. In this case, for the node type *sodalite.nodes.AWS.VM* two different files are generated for the operation *create*(create.ans, create.yaml) and stored on the folder Standard, the parent interface of the operation. The same also holds for the node type *sodalite.nodes.AWS.SecurityRule* where two files related to the operation *delete*(delete.ans, delete.yaml) are generated and stored at the corresponding folder Standard. In the example above, both node types happened to have an interface with the same name, the name Standard.

We want to emphasize that generating Ansible files through the RM editor is not required. The user can use Ansible scripts written with an external editor without first developing the abstract Ansible model (i.e. .ans file). In this way, the RM editor maintains transparency regarding the origin of the implementation scripts. Section 4.2.3 will go into greater detail.



(a) Directory structure

(b) Example directory structure

Figure 4.3: Folder organization

4.2.2 Connection of Ansible editor with SODALITE KB

The Ansible editor has been extended to support the communication and information exchange between the provided user interface and Knowledge Base content. Through this interaction, the user has access to the contents of the Knowledge Base and can refer to information gathered from the stored Resource Models.

In the first version of the Ansible editor [50], the user creates an Ansible model in the context of a local Resource Model, specifically a TOSCA operation, and can refer to its inputs as declared variables within the Ansible model. We expanded on this concept by developing mechanisms that enable the development of an Ansible model within the context of a Resource Model stored in KB. As a result of these mechanisms, the user can now refer to the inputs defined by the selected TOSCA operation as well as the inputs defined within the Ansible interface that contains the selected TOSCA operation. The presentation of this information is performed by the content assistant component that provides visual suggestions by using the "CTRL+Space" combination.

Figure 4.4 depicts how the Ansible editor displays the resource types in the Knowledge Base. Each resource type belongs to a distinct namespace, which aids in the organization of the Knowledge Base's various Resource Models. The Ansible editor preserves this organizational structure by providing the full list of resource types, organized by namespace, to assist the user in making the best choice.

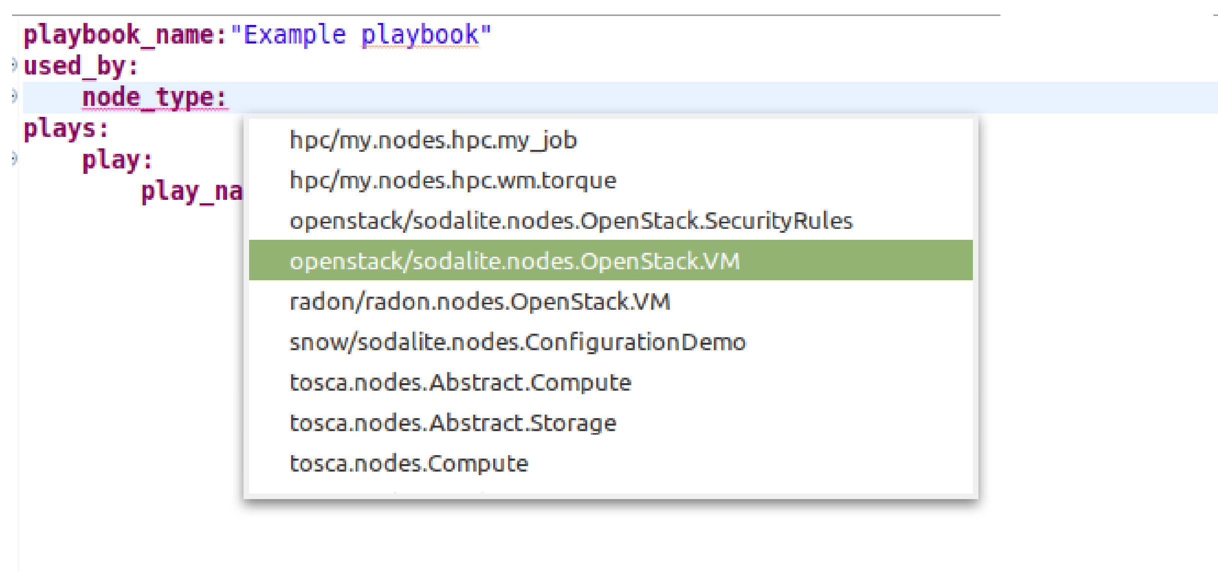


Figure 4.4: Select a resource type from a Resource Model stored in KB

Following the selection of a node type, the user can select one of the interfaces contained within it, as shown in Figure 4.5

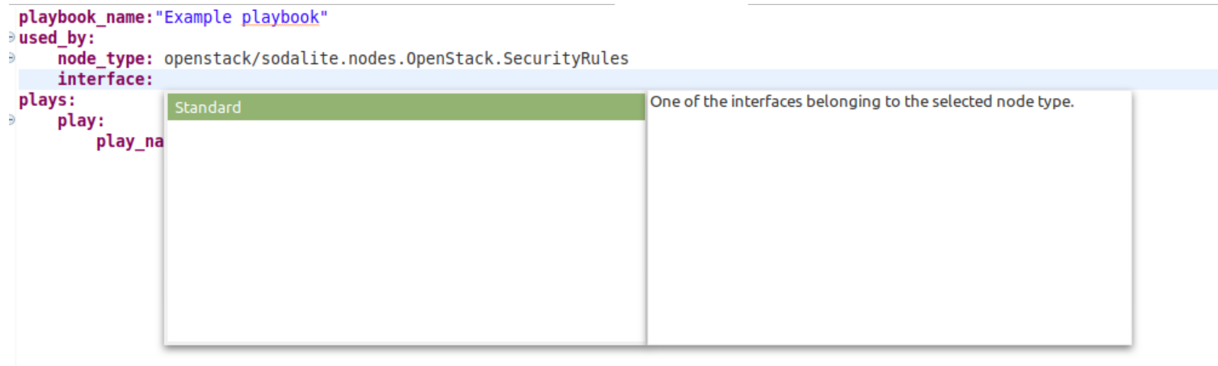


Figure 4.5: Select interface

The same holds for operation selection, where the user can choose between the operations that belong to the selected interface, as shown in Figure 4.6.

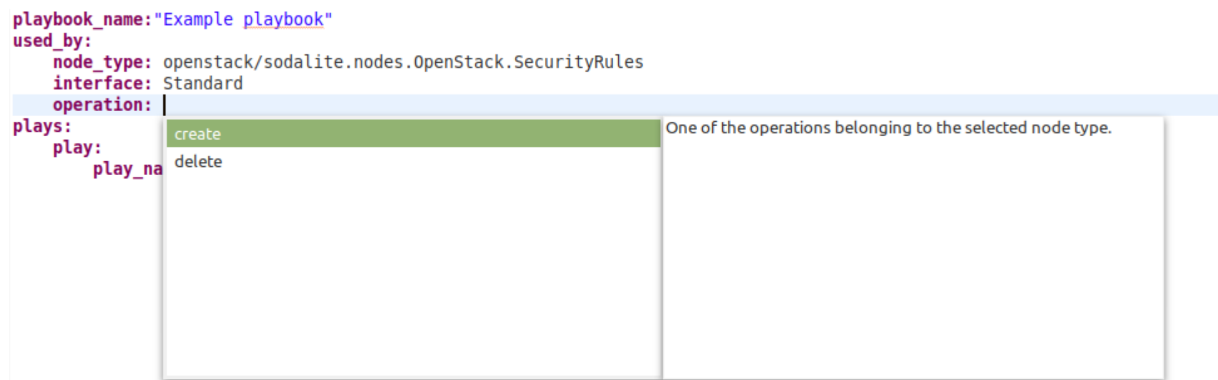


Figure 4.6: Select operation

At this point, we would like to emphasize why, in comparison to the first version of the Ansible editor, we added the 'interface' option when selecting the appropriate TOSCA operation. Previously, when a user chose a node type with many interfaces and operations, the Ansible editor generated a large pool of suggested operations from all the contained interfaces. In this case, a large pool of proposals may confuse the user and complicate the selection process. As a result, filtering the proposed operations by interface simplifies the decision and reduces the time required to select the appropriate operation.

After selecting the interface and the included operation, the corresponding interface and operation inputs can be used as variable references within the Ansible model.

Figure 4.7 shows an example where the user selects an operation input as a value for an attribute.

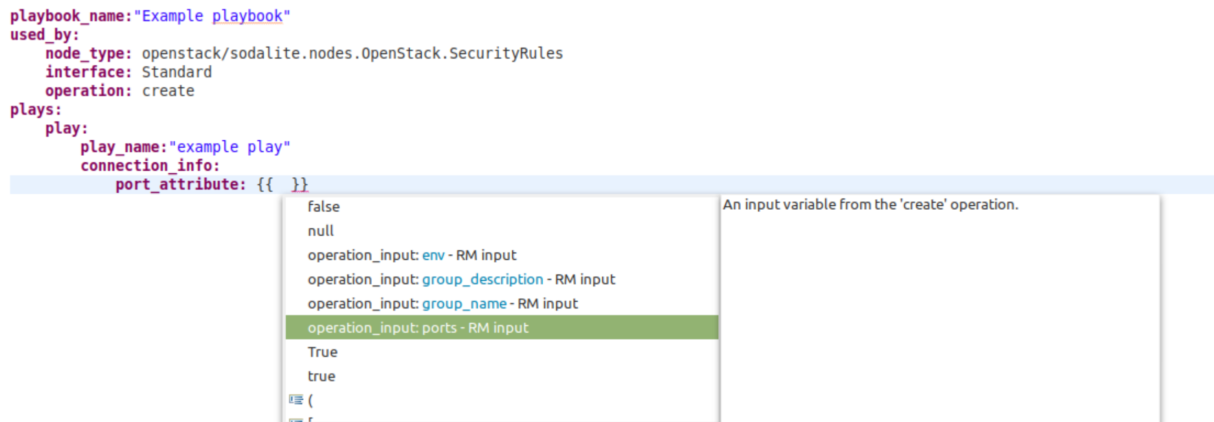


Figure 4.7: Select operation input as variable

4.2.3 Portability of Resource Models

As described in the section 4.2.1, the RM editor creates a specific directory structure in which the various Ansible files of a RM are stored. We take advantage of this structure to improve the portability of Resource Models between users and enable the seamless exchange of Resource Models with all their related Ansible files together.

We implement a feature that allows users to bind a TOSCA operation defined in an RM with an Ansible script by using the script's relative path to the RM's location. Thus, when a RE uploads an RM to the KB, the paths of the implementation scripts will be relative to the RM's location, with no reliance on any local file system. When a user retrieves an RM from the KB, the IDE will ask if he/she wants to retrieve all the related Ansible files. If the answer is yes, the IDE will create the same directory structure used during the RM's development, keeping the defined relative paths consistent. Otherwise, only the RM will be stored locally. Figure 4.8 depicts an example where the developer binds the TOSCA operation with Ansible scripts using their relative paths.

```
module: aws

node_types:
    sodalite.nodes.AWS.VM:
        derived_from: tosca.nodes.Compute
        interfaces:
            Standard:
                type: tosca.interfaces.node.lifecycle.Standard
                operations:
                    create:
                        inputs:
                            aws_access_key:
                                type: string
                                default: get_property:
                                    entity:SELF
                                    property: aws/sodalite.nodes.AWS.VM.aws_access_key
                        implementation:
                            primary:"./exampleRM-Ansible files/sodalite.nodes.AWS.VM/Standard/create.yaml"

    sodalite.nodes.AWS.SecurityRules:
        derived_from: tosca.nodes.Root
        interfaces:
            Standard:
                type:tosca.interfaces.node.lifecycle.Standard
                operations:
                    delete:
                        inputs:
                            rules:
                                type: string
                                default: get_property:
                                    entity: SELF
                                    property: aws/sodalite.nodes.AWS.SecurityRules.rules
                        implementation:
                            primary:"./exampleRM-Ansible files/sodalite.nodes.AWS.SecurityRules/Standard/delete.yaml"
```

Figure 4.8: Resource Model example

When a user retrieves the Resource Model in Figure 4.2 from SODALITE Knowledge Base, the IDE will recreate the directory structure shown in Figure 4.3b. Therefore, the relative paths indicating the location of Ansible scripts remain consistent across all machines that download the RM.

Suppose now the user selects an implementation script that is not in the specified directory structure. In that case, the IDE copies it into the appropriate folder and renames it with the name of the related TOSCA operation. In this manner, the Ansible script is always encountered in the specified directory structure, and we avoid inserting inconsistencies within the RM. For example, assume a user has an implementation script named *script.yaml* that implements the business logic of the *create* operation in figure 4.8 and is located in the desktop folder. When the user selects this file, the IDE copies it to the following directory */ExampleRM-Ansible files/sodalite.nodes.AWS.VM/Standard/* and renames it as *create.yaml*.

4.3 Supporting the reuse of Ansible elements

This section describes how we collected Ansible content from various sources and how we formulated a database that structures the collected information. Moreover, we present the features we added to the Ansible editor that exploits the Ansible content as well as the main components that provide the new features, which are the Xtext framework's Proposal Provider and Validator and the Ansible Defect Predictor as an external component.

4.3.1 Ansible content

One of the main contributions of our work is the integrated use of Ansible content, such as Ansible collections, modules, and roles within the SODALITE Ansible editor. As described in Section 3.4, the procedure for selecting the appropriate pre-packaged Ansible unit from Ansible Galaxy is not simple, and the developer must consider a number of factors. As a result, in order to mitigate these issues from the developers' standpoint, we established some criteria that can verify that the chosen Ansible component guarantees the provided functionality and meets some minimum quality standards. The Ansible content was filtered using the following criteria: documentation quality, maintenance, popularity, developer satisfaction, and security, as we are dealing with reusable units of code developed by verified and unverified sources.

As a first step, we examined the documentation of all the Ansible collections stored in Ansible Galaxy and investigated how many of the included modules adhere to the official Ansible documentation. The outcome was that 911 modules out of 36882 had incomplete documentation or no documentation at all. Furthermore, many of the examined collections had few or no downloads, raising concerns about future maintenance efforts. Finally, because Ansible Galaxy is a public repository where anyone can upload Ansible collections, we were hesitant to include all of the Ansible collections that were uploaded for security reasons.

Taking all of this into account, we decided to select only Ansible collections that are related to the official Ansible documentation [16] and are RedHat certified [56]. We examined these Ansible collections to check the documentation of the included modules and discovered that only 10 modules out of 7211 had minor issues with their documentation. Furthermore, they provide a wide range of functionalities to the end-user and make up the vast majority of the most popular collections in Ansible Galaxy. Finally, the fact that official creators and companies develop them ensures that the developed modules are secure and maintained in the future.

Regarding the standalone Ansible roles of Ansible Galaxy, we followed a different approach. We could not find any certified content from the official Ansible documentation or RedHat, so we evaluated the quality of each role based on the content score that each role received from users on Ansible Galaxy [57]. The weighted average of the community and quality scores is used to calculate the content score. The former is based on surveys submitted by users in Ansible Galaxy, while the latter is based on automatic validation from the yamllint and ansible-lint tools. We chose Ansible roles with a download counter greater than 20000 and a content score greater than 4.0. As a result, we can ensure that many users have used this role and are satisfied with its function and support.

Ansible's content is stored in a well-structured MongoDB database, called AnsibleDB, and its detailed database schema is depicted in Figure 4.9. The database contains two MongoDB collections: *Ansible_Galaxy_Collections* and *Standalone_Ansible_Galaxy_Roles*. *Ansible_Galaxy_Collections* contains information about Ansible collections, including modules and roles, as well as the taking parameters and their details for each module, such as type, description, available value choices, and default value. *Standalone_Ansible_Galaxy_Roles* is a MongoDB collection that contains information about the standalone roles, and for each role, the database includes details such as the number of downloads, the user satisfaction score, the role description, and how a user can refer to each role (role name, role namespace). Aside from the contents of MongoDB, one critical aspect is the database's performance and how it handles incoming requests. In this context, we must consider how the Xtext framework provides validation and static analysis to the DSL. The static analysis provided can be used in three ways: whenever the file is modified, when the file is saved, or when the user explicitly validates the file from the menu option. The first option, in our opinion, is the most suitable because it allows for real-time validation of the user's inputs. This method, however, necessitates a large number of database requests each time a user makes a change to a local file, which can cause problems if multiple people use the Ansible editor simultaneously. To address this issue, we decided to provide an automated way that deploys the MongoDB database locally within a container, considering its small size. As a result, the database is relieved of managing a large number of queries, increasing overall system performance and the latency of the validation mechanisms.

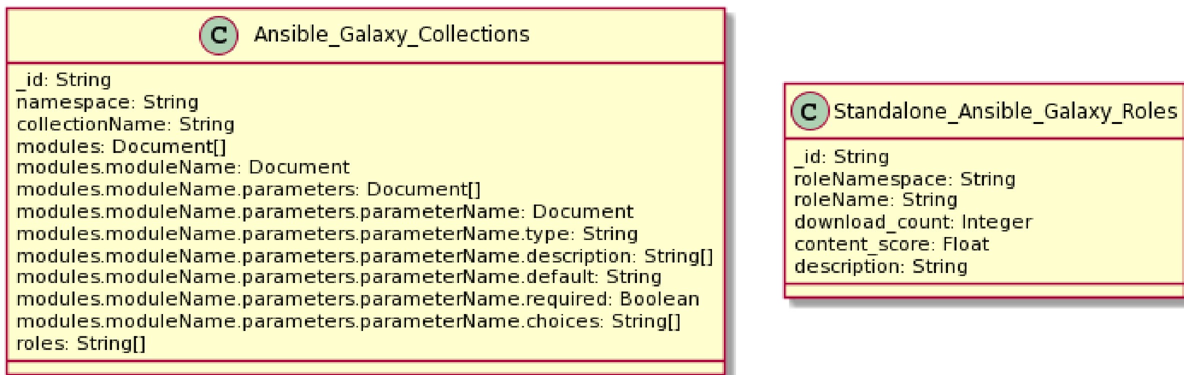


Figure 4.9: Database schema

4.3.2 Proposal Provider

The Proposal provider is a component offered by the Xtext framework that provides visual suggestions to the user while writing code by pressing 'CTRL+Space'. More specifically, we extended the Proposal Provider to enable users to navigate the Ansible content stored in the related MongoDB database, named AnsibleDB. The newly introduced features are the following:

- **Import Ansible collections**
  When a user needs to import a specific Ansible collection, the Proposal Provider was extended to suggest namespaces and collection names. Users can navigate the list of available namespaces (Figure 4.10), and after selecting one, the available collection list is filtered, and the user can select one of the collections contained within the chosen namespace (Figure 4.11).

```
playbook_name: "hello world playbook"
used_by:
    node_type:openstack/sodalite.nodes.OpenStack.SecurityRules
    interface: Standard
    operation: create

plays:

    play:
        play_name: "example play"
        connection_info:
            port_attribute: {{ operation_input: ports }}
        collections:
            - |
```

amazon - Namespace	Collection Namespace as it is depicted in Ansible Galaxy
ansible - Namespace	
arista - Namespace	
awx - Namespace	
azure - Namespace	
check_point - Namespace	
chocolatey - Namespace	
cisco - Namespace	
cloudscale_ch - Namespace	

Figure 4.10: Select Ansible namespace

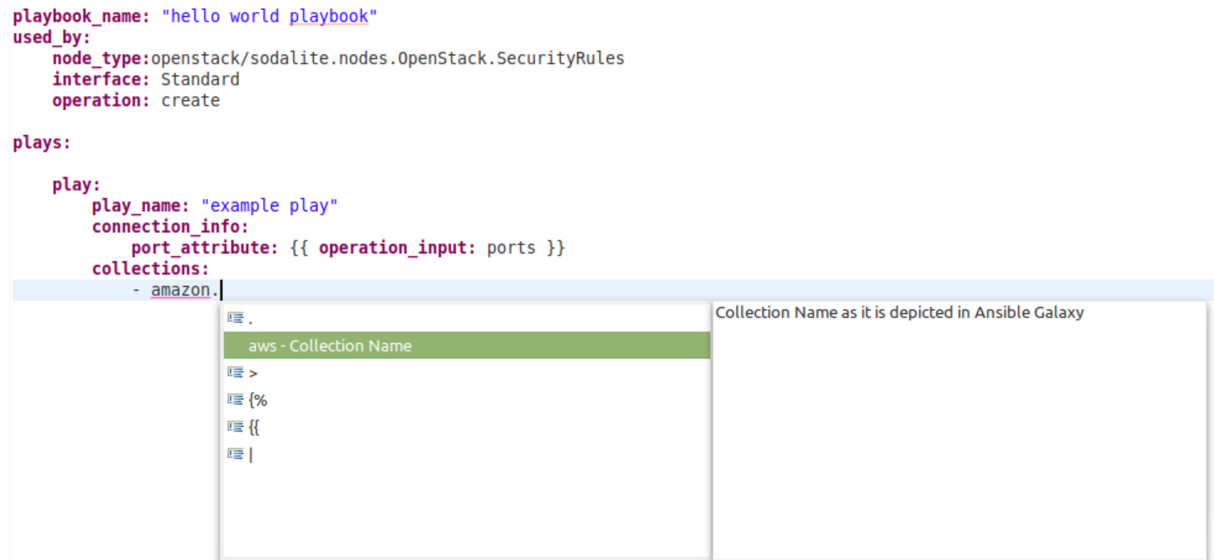


Figure 4.11: Select Ansible collection

- **Module selection**
  A significant addition to the current arsenal of features is the suggestion of modules. The majority of modules are hosted in Ansible collections, and so we thought it would be reasonable to extend the Proposal Provider in a way that suggests modules from the pool of the imported collections (Figure 4.12). Furthermore, the users can refer to Ansible modules through its fully qualified name by determining its three parts: the namespace, the collection, and the module's name. The Ansible editor provides suggestions for each part of the fully qualified name (Figure 4.13), filtering the options based on the previously determined part.

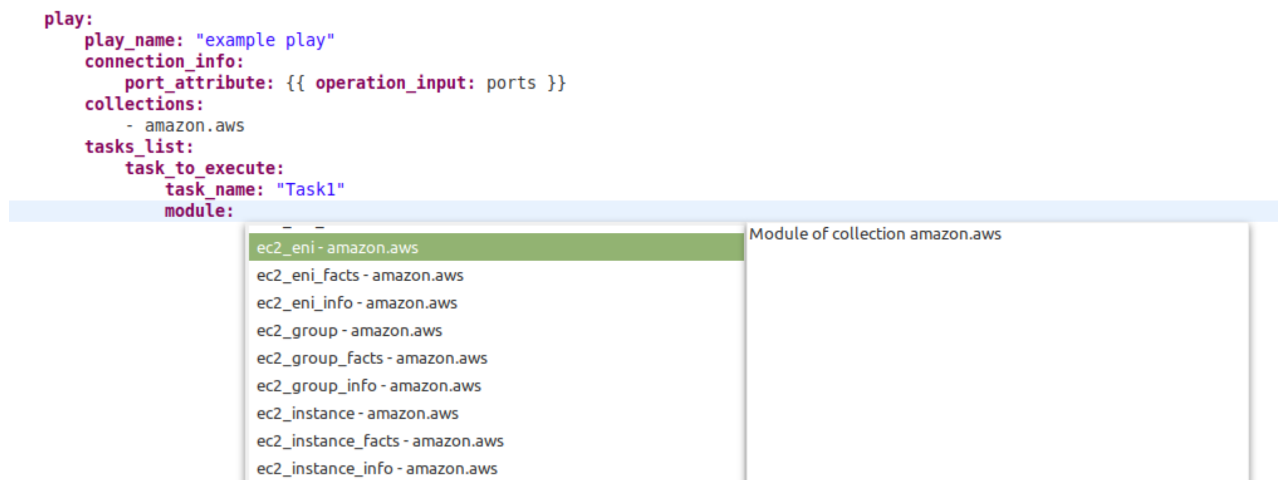


Figure 4.12: Select module from the pool of the imported collections

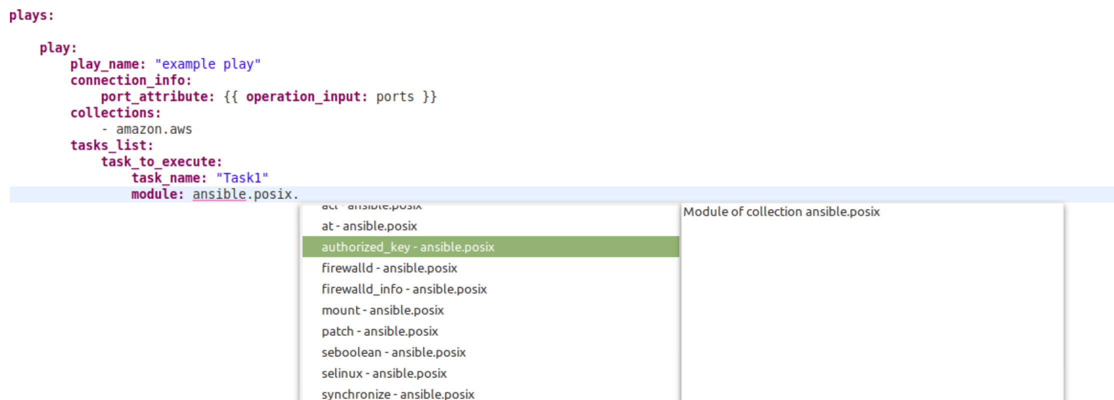


Figure 4.13: Select module through its fully qualified name

- **Parameters**
  The procedure for defining parameters to be passed in a module is time-consuming because the user must constantly check the module's documentation for the required parameters as well as the type and allowed values for each parameter. Moreover, some complex parameters necessitate the definition of a set of subparameters in order for the module to function properly. Because all of this information is stored in the MongoDB database, we gave users access to it through the suggestions provided by the Proposal Provider. As a result, the Ansible editor provides content assistance for each Ansible module's parameters, emphasizing the required ones (Figure 4.14) and the default value, if such exists (Figure 4.15). It also informs the user about inserting values for each parameter by displaying the value type that each parameter expects and presenting the acceptable values (Figure 4.17) and the official description that helps the user understand its purpose. Finally, it enumerates the possible subparameters that can be defined (Figure 4.16).

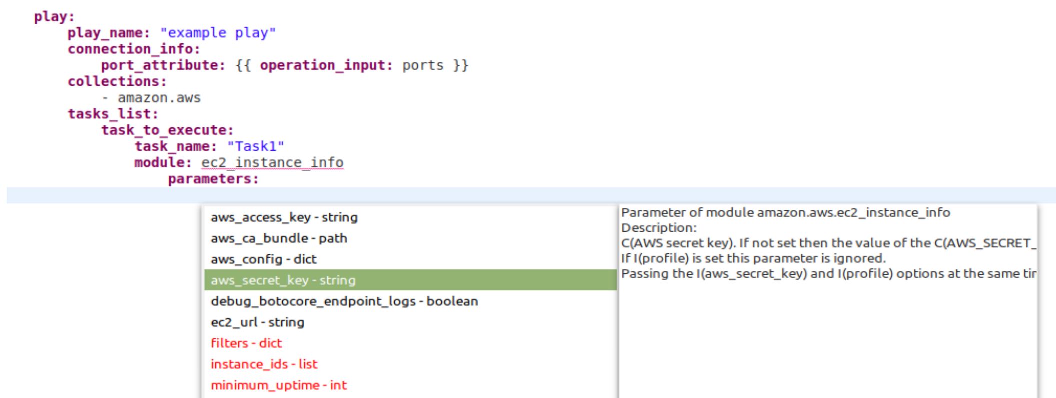


Figure 4.14: Select parameter for the chosen module

```
plays:

    play:
        play_name: "example play"
        connection_info:
            port_attribute: {{ operation_input: ports }}
        collections:
            - amazon.aws
        tasks_list:
            task_to_execute:
                task_name: "Task1"
                module: ec2
                    parameters:
                        wait_timeout:
```

"String" - STRING
0 - NUMBER
300 - Default value
False
false
no
null
True
true

A number

Figure 4.15: Default value of the selected parameter

```
plays:

    play:
        play_name: "example play"
        connection_info:
            port_attribute: {{ operation_input: ports }}
        collections:
            - amazon.aws
        tasks_list:
            task_to_execute:
                task_name: "Task1"
                module: ec2
                    parameters:
                        volumes:
```

delete_on_termination - bool
device_name - str
encrypted - bool
ephemeral - str
iops - int
Name - ID
snapshot - str
volume_size - int
volume_type - str

Subparameter of volumes
Description:
Whether the volume should be automatically deleted when the ins

Figure 4.16: Subparameters for complex parameters

```
        parameters:
            name:"String"
            resource_group: "String"
            profiles:
                fixed_date_end : "String"
                count : 4
                name: "String"
                rules:
                    metric_name:"String"
                    operator:
                    statistic
                    time_grai
                    time_wind

    task_to_execute:
        module: amazon.aws.aws_s3
        parameters:
            bucket:"String"
            encryption_mode:
            mode:"create"
            permission: [priv

    task_to_execute:
```

:
Equals
GreaterThan - Default value
GreaterThanOrEqual
LessThan
LessThanOrEqual
NotEquals
>
[

Figure 4.17: Acceptable values of the requested parameter

- **Role selection**

  Developers widely use roles in order to automate a set of tasks and make it reusable among the community. Thus, a suggestion mechanism that allows developers to select the appropriate Ansible role directly through the Ansible editor would be beneficial. For this reason, we took advantage of the Ansible Galaxy content and enriched the Proposal Provider with role selection suggestion mechanisms. These roles can be part of an Ansible collection or stand-alone without being associated with a collection. In the first case, the user can choose the desired role from the pool created by the imported collections (Figure 4.18) or by defining its fully qualified name as in Ansible modules (Figure 4.19), whereas in the second case, the user must define the namespace to which the role belongs (Figure 4.20) as well as its simple name (Figure 4.21).

```
plays:

    play:
        play_name: "example play"
        connection_info:
            port_attribute: {{ operation_input: ports }}
        collections:
            - amazon.aws
            - community.mongodb
        roles_inclusions:
            role:
                role_name:
        tasks_list:
            task_to_execute
                task_name:
                module: ec2
                    paramet
                        wai
```

```
mongodb_auth - community.mongodb
mongodb_config - community.mongodb
mongodb_install - community.mongodb
mongodb_linux - community.mongodb
mongodb_mongod - community.mongodb
mongodb_mongos - community.mongodb
mongodb_repository - community.mongodb
mongodb_selinux - community.mongodb
>

Role of collection community.mongodb
```

Figure 4.18: Select role from the pool of the imported collections

```
plays:

    play:
        play_name: "example play"
        connection_info:
            port_attribute: {{ operation_input: ports }}
        collections:
            - amazon.aws
        roles_inclusions:
            role:
                role_name: community.mongodb.
        tasks_list:
            task_to_execute:
                task_name: "Task1"
                module: ec2
                    parameters:
                        wait_timeout: 300
```

```
.
mongodb_auth - community.mongodb
mongodb_config - community.mongodb
mongodb_install - community.mongodb
mongodb_linux - community.mongodb
mongodb_mongod - community.mongodb
mongodb_mongos - community.mongodb
mongodb_repository - community.mongodb
mongodb_selinux - community.mongodb

Role of collection community.mongodb
```

Figure 4.19: Select role belonging to an Ansible collection through its fully qualified name

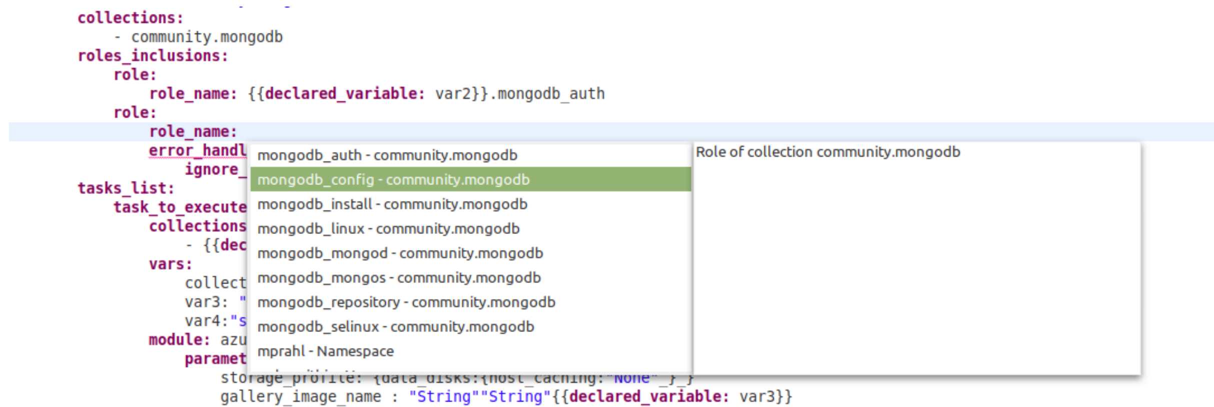


Figure 4.20: Select Ansible namespace

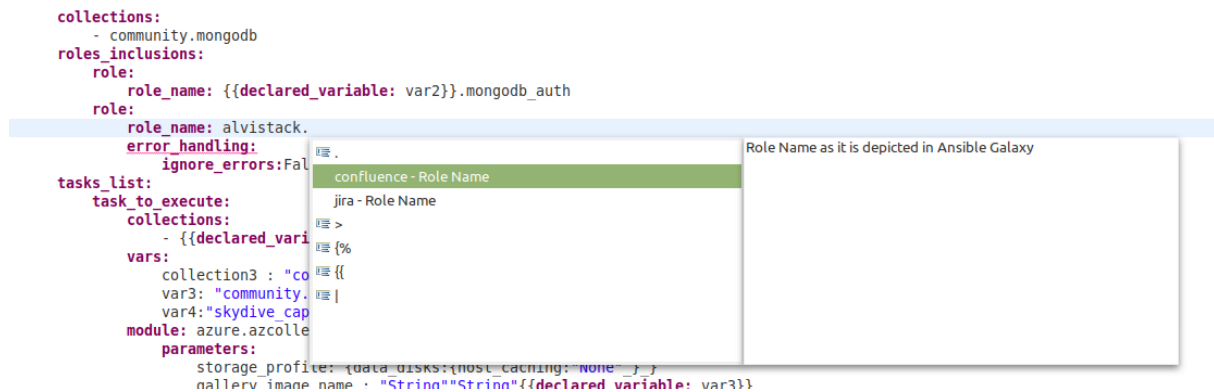


Figure 4.21: Select role contained in the chosen Ansible namespace

4.3.3 Validator

Aside from proposals, we can use the information stored in the MongoDB database for validation and provide the user with real-time feedback on potential errors. The validation mechanisms that have been implemented are as follows:

- **Collection and Role support**
  For the reasons stated in Section 4.3, SODALITE does not support the execution of all Ansible collections and roles. As a result, it is necessary to notify the user if an imported collection or role is supported and to use the provided suggestion mechanisms to choose the appropriate Ansible component (Figure 4.22).

```
vars:
    var1: {{ operation_input: env}}
    collection1: "amazon.aws"
    var2:"community.mongodb"
    var5:"abims_sbr"
collections:
    - community.mongodby
```

```
vars:
    var1: {{ operation_input: env}}
    collection1: "amazon.aws"
    var2:"community.mongodb"
    var5:"abims_sbr"
collections:
The collection community.mongodby is not supported
```

Figure 4.22: Unsupported Ansible collection

- **Collection and Module name format**

  Ansible developers frequently make the mistake of defining an imported collectio (Figure 4.23) or module (Figure 4.24) incorrectly, which is discovered only after the Ansible playbook is executed. To avoid this, we enhanced the Validator to detect these errors and alert the user to correct them.

```
plays:

    play:
        play_name: "example play"
        connection_info:
            port_attribute: {{ operation_input: ports }}
        collections:
            - amazon
        roles_  Collection name is missing.
            ro     Press 'F2' for focus.mongodb.mongodb_auth
        tasks_list:
            task_to_execute:
                task_name: "Task1"
                module: ec2
                    parameters:
                        wait_timeout: 300
```

Figure 4.23: Wrong Ansible collection name format

Figure 4.24: Wrong Ansible module name format

- **Parameters' type and values**
  Checking for parameter values and type correctness is another important feature that we introduced. An Ansible playbook may contain a large number of different parameters and subparameters, which can be a source of multiple errors that developers may not be able to detect.
  An automated validation mechanism is useful for developers and has the potential to reduce development costs significantly. As a result, we extended the Validator component to determine whether or not the inserted value for a parameter is correct. For example, if a user enters an integer value into a string parameter, the Ansible editor will highlight the parameter and inform the user of the error details (Figure 4.25).

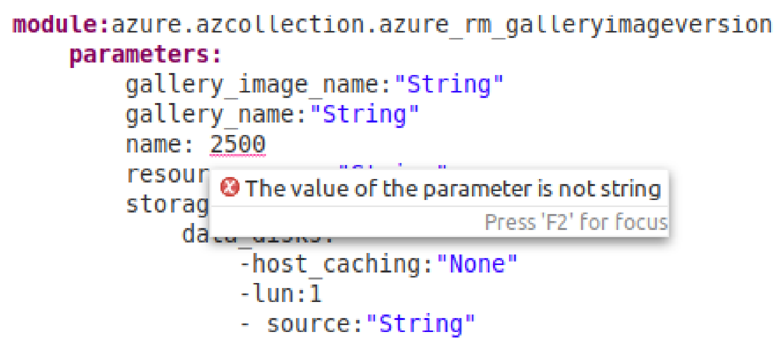


Figure 4.25: Wrong value type

Moreover, some parameters and subparameters accept only specific values as input, causing issues during playbook execution even if the developer has inserted the same input type. To prevent this, the Ansible editor has been enhanced to check for acceptable values and generate an error message that displays all of the appropriate values that can be assigned (Figure 4.26). In addition, some modules require

mandatory parameters to be passed to be executed. In light of this, we implemented a validation mechanism that alerts the user when mandatory parameters and subparameters are missing (Figure 4.27,Figure 4.28). Finally, the Ansible editor has been expanded to accept only parameters and subparameters valid for the related Ansible module (Figure 4.29).

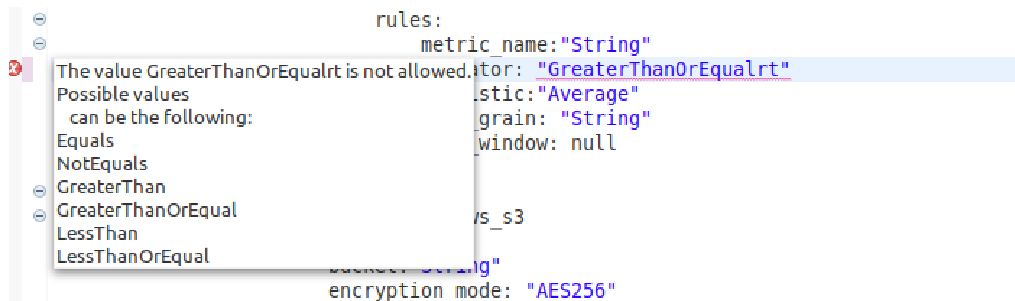


Figure 4.26: Available values to be assigned

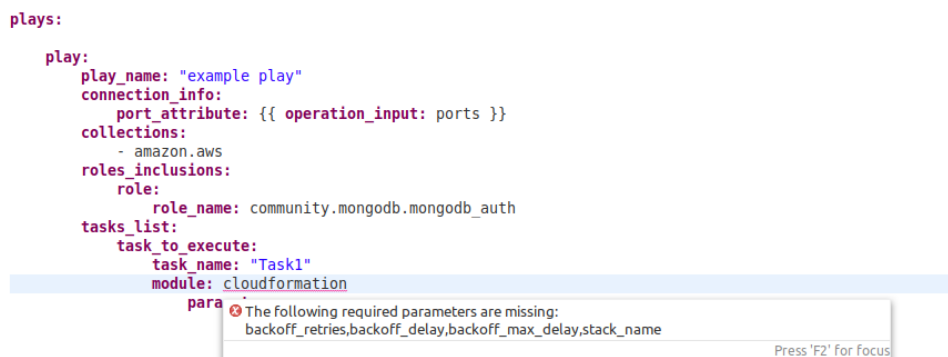


Figure 4.27: Missing mandatory parameters

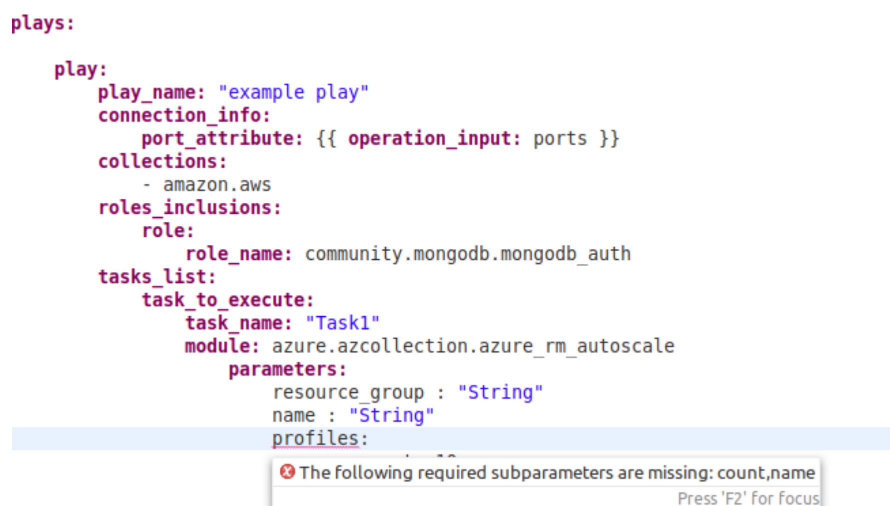


Figure 4.28: Missing mandatory subparameters

```
module:azure.azcollection.azure_rm_galleryimageversion
    parameters:
        gallery_image_name:"String"
        gallery_name:"String"
        nameer:"String"
```

```
The parameter nameer is not included in module
azure.azcollection.azure_rm_galleryimageversion
                                                  Press 'F2' for focus
```

```
                -lun:1
                - source:"String"
```

Figure 4.29: Invalid parameter

4.3.4 Ansible Defect Predictor

Ansible scripts can provision and configure development environments and servers on a large scale by adhering to software engineering principles and best practices. On the other hand, developers frequently violate best practices and introduce code smells, negatively impacting maintainability and code quality. To address this issue and provide continuous feedback to the user, we connected the Ansible defect predictor developed as part of the SODALITE project [58] with the Ansible editor. When a user saves an abstract Ansible model, the Ansible editor sends the equivalent concrete Ansible script to the Ansible defect predictor, which is deployed in SODALITE public testbed for analysis. As a response, the Ansible defect predictor analyzes the received Ansible script and returns all detected code smells. Then, the identified code smells are presented to the user via the Ansible editor at the level of play or task, indicating the existing code smells as well as their type (Figure 4.31). As a result, the user can quickly identify 'smelly' pieces of code and refactor the abstract Ansible model.

```
13        tasks_list:
14            task_to_execute:
15                task_name: "Add nodes to dse-cluster group (from"{{outer_loop.dc_name}}")"
16                module: add_host
17                    parameters:
18                        name:{{ hostvars[local_item].inventory_hostname}}
19                        ansible_host: {{ hostvars[local_item].ansible_host|default(hostvars[local_item].ansible_ssh_host) }}
20                        ansible_user:{{ hostvars[local_item].ansible_user|default('root') }}
21                        ansible_ssh_pass:{{ hostvars[local_item].ansible_ssh_pass|default('') }}
22                        ansible_become_user:root
23                        ansible_become_pass: {{ hostvars[local_item].ansible_ssh_pass|default('') }}
24                        groups: [dse-cluster]
25
26                when: "{{ outer_loop.options.workloads is defined }}"
27                tags_attribute:
28                    - "always"
29                loop:
30                    loop_over: "String"
31                    loop_control:
32                        loop_var: local
```

Figure 4.30: Example playbook

```
13          tasks_list:
14  Multiple markers at this line
15  - There is the following code smell: EmptyPassword     dse-cluster group (from"{{outer_loop.dc_name}}")"
16  - There is the following code smell: HardCodedSecrets
17  - There is the following code smell: AdminByDefault
18  - There is the following code smell: No Jinja2 in when  local_item].inventory_hostname}}
19                      ansible_host: {{ hostvars[local_item].ansible_host|default(hostvars[local_item].ansible_ssh_host) }}
20                      ansible_user:{{ hostvars[local_item].ansible_user|default('root') }}
21                      ansible_ssh_pass:{{ hostvars[local_item].ansible_ssh_pass|default('') }}
22                      ansible_become_user:root
23                      ansible_become_pass: {{ hostvars[local_item].ansible_ssh_pass|default('') }}
24                      groups: [dse-cluster]
25
26              when: "{{ outer_loop.options.workloads is defined }}"
27              tags_attribute:
28                  - "always"
29              loop:
30                  loop_over: "String"
31                  loop_control:
32                      loop_var: local
```

Figure 4.31: Code smells

5. Evaluation

This chapter aims to present the evaluation procedure of the new version of Ansible DSL/Editor, identify how the newly added extensions assist the user in writing quality implementation scripts, and compare it with the Atom text editor. First, we describe the evaluation objectives of our procedure in Section 5.1. Then, in Section 5.2, we briefly summarize the evaluation process in the initial version of Ansible DSL and Editor, what this evaluation procedure adds to the previous one, how we structured the questionnaire, and the purpose of each question, with the work done in [50] as a starting point.

5.1 Evaluation objectives

As we have seen so far, our work aims to make it easier for users to write Ansible blueprints and reuse pre-existing elements like collections, roles, and modules. However, it is critical to assess how the new features affect end-users and whether they are helpful to the development community. As a result, we need to collect information from a group of professionals through an evaluation procedure to assess their satisfaction with our system and receive feedback that can be used to improve and expand our work in the future.

Our primary goal during this evaluation procedure is to determine whether or not our system actually helps users by measuring some quantitative and qualitative metrics and checking if there is an improvement in comparison to the experience a user has with the Atom text editor, one of the most popular editors in the community. These metrics would provide us with a clear picture of the participants' opinions and beliefs after they have used our system as well as Atom and experienced all of their features. Furthermore, during this procedure, we would be able to record how our system works in real-world scenarios and test for potential bugs. Finally, in order to contextualize the collected responses, we would catalog participants' attributes such as working experience and current working role.

5.2 Evaluation procedure

The authors of the initial version of the Ansible editor [50] followed a three-step evaluation procedure to check the correctness of the Ansible DSL and receive feedback from users on the provided Ansible editor. In the first step, code coverage tools were used to check the correspondence between the expected generated playbooks and the actual generated playbooks, with 98 percent test coverage. The authors then tested the expressive power of Ansible DSL to ensure that users can develop any Ansible playbook using the Ansible DSL. To accomplish this, they chose a large set of Ansible playbooks and wrote the corresponding Ansible models, demonstrating that all Ansible playbooks can be written as Ansible models. Finally, they compared the Ansible editor with the Atom text editor and created a questionnaire where they asked engineering students about their experience using the Ansible editor.

Using this as a starting point, we will concentrate on the final step of this procedure, broaden the scope of the questionnaire, and assess how the extensions we have made encourage developers to use the Ansible DSL and editor in comparison to the solution offered by Atom.

In order to evaluate our work and check if the defined objectives were met, we conducted a survey among seven professionals from various backgrounds. First, we asked them to write four Ansible playbooks with varying characteristics using our Ansible editor and the Atom text editor's Ansible language plugin, as this editor is one of the most widely used for development in the Ansible community. More specifically, users created Ansible playbooks for 1) a simple LAMP application, 2) a VM on the AWS cloud infrastructure, 3) a Docker container running an Apache server, and 4) a MongoDB database. We provided instructions in the form of exercises for all the playbooks to help users understand the purpose of each playbook. The requested exercises have been designed to enable users to explore all the provided features of the Ansible editor.

The first playbook is simple and introductory to familiarize users with the Ansible editor's UI and give them a glimpse of the features available. The second playbook is created in the context of a Resource Model to show users how the integration between TOSCA models and Ansible works in the Ansible editor and how users can use information stored in the Knowledge Base. Finally, the last two playbooks share a common goal: to demonstrate the Ansible editor's capabilities in relation to the collected Ansible content via suggestion and validation mechanisms in two different scenarios.

Users developed the requested playbooks in Ansible DSL via the Ansible editor and then automatically generated the corresponding Ansible scripts. Then, they developed the playbooks in Ansible via the Atom text editor. In this way, the users will be able to compare the features of the Ansible editor and Atom and identify the different approaches of the two solutions.

Following that, we asked them to complete a questionnaire designed to investigate and document various factors for each editor and provided language gleaned from the related literature. Each question is related to a different factor and focuses on the user's satisfaction with that aspect. To obtain the necessary feedback from the users, we included Likert-scaled questions with a scale of 1 to 5 points, where 1 point corresponds to the most negative answer and 5 to the most positive answer, and open-ended questions.

The authors in [59] propose specific factors that measure how valuable a DSL is, as well as which characteristics contribute to its success. We used these factors to assess users' satisfaction with various aspects of Ansible DSL and the related editor and draw conclusions about the utility of the new extensions we implemented compared to Atom and Ansible. More specifically, the factors are the following:

- Expressiveness: The capability of the Ansible DSL to describe the Ansible playbooks sufficiently and concretely.
- Learnability (L): How much effort and time do the developers need to learn the new Ansible DSL.
- Reliability (R): What features does each editor provide to the users to make the development procedure less faulty and improve the quality of the delivered code.
- Usability (U): How does each editor and language ease developing an Ansible script.
- Implementation costs (C): If and how much each editor and language reduces the development time.
- Reusability (RE): How helpful is the integration of Ansible and already implemented TOSCA models in the context of the new Ansible editor and Atom.

We did not measure the factor of expressiveness in our questionnaire because it was investigated in [50] while the factor of learnability was measured only for the Ansible DSL in order to identify how easy its adoption would be.

Every question in the questionnaire is related to one or more of these factors and is designed to evaluate various aspects of the two solutions. Table 5.1 presents the questions that have been chosen along with the factor that each question measures:

	Questions	Measured factors
Q1	How many years do you work as a software developer?	L
Q2	What is your current role?	L
Q3	What is your experience with Ansible?	L
Q4	How do you know Ansible?	L
Q5	How easy was it to learn the Ansible DSL?	L
Q6	How easy was it to install the editor?	U
Q7	How much time did it take on average to create an Ansible playbook with the editor?	C
Q8	How intuitive was it to use the language?	U
Q9	How do you rate the provided user interface?	U
Q10	How do you assess the language's ability to allow you to organize the code in a clear and logical manner?	R
Q11	How do you evaluate the integration of TOSCA models with Ansible?	RE
Q12	How complex do you find the development procedure?	U
Q13	How do you rate the error mechanisms	R
Q14	Which two features do you think are the most useful?	R, U
Q15	Rate features of Ansible editor based on how useful they are.	R, U
Q16	Name two advantages and two disadvantages for the editor	R, U

Table 5.1: Questions of the evaluation procedure

Questions Q1-Q4 are related to each professional's background and are used to document each professional's experience with software development and Ansible. Question Q5 was designed to assess how easy it is to learn Ansible DSL independently of Ansible, whereas questions Q6-Q13 and Q16 are intended to assess the user's satisfaction while developing Ansible code with both editors. Finally, questions Q14 and Q15 identifies the most useful features of the Ansible editor. Table 5.2 presents the questionnaire that the testers completed.

Questions	**Answer Options**
Q1	☐ 0-3 ☐ 3-5 ☐ 5-8 ☐ 8+
Q2	Text
Q3	☐ None ☐ Beginner ☐ Intermediate ☐ Advanced ☐ Expert
Q4	Text
Q5	☐ Very easy ☐ Easy ☐ Normal ☐ Difficult ☐ Very difficult
Q6	☐ Very easy ☐ Easy ☐ Normal ☐ Difficult ☐ Very difficult
Q7	☐ 0-15 ☐ 15-30 ☐ 30-45 ☐ 45-60 ☐ 60+
Q8	☐ Very straightforward ☐ Straightforward ☐ Neither straightforward nor difficult ☐ Difficult ☐ Very difficult
Q9	☐ Very poor ☐ Poor ☐ Fair ☐ Good ☐ Exceptional

Q10	☐ Very poor ☐ Fair ☐ Exceptional	☐ Poor ☐ Good
Q11	☐ Not helpful at all ☐ Somewhat helpful ☐ Extremely helpful	☐ Not very helpful ☐ Very helpful
Q12	☐ Very simple ☐ Neither simple nor complex ☐ Very complex	☐ Simple ☐ Complex
Q13	☐ Not helpful at all ☐ Somewhat helpful ☐ Extremely helpful	☐ Not very helpful ☐ Very helpful
Q14	☐ Ansible Galaxy content proposal ☐ Syntax proposal ☐ Integration with TOSCA ☐ Code smells detection	☐ Content validation mechanisms ☐ Syntax highlighting ☐ Abstraction from Ansible
Q15	☐ Ansible Galaxy content proposal ☐ Syntax proposal ☐ Integration with TOSCA ☐ Code smells detection	☐ Content validation mechanisms ☐ Syntax highlighting ☐ Abstraction from Ansible
Q16	Text	

Table 5.2: Questionnaire

5.3 Evaluation Results

In this section, we present the feedback we received from the 7 testers who experienced the development of Ansible playbooks with the Ansible DSL and Atom text editor. The responses from the testers are presented in detail in Appendix A. Regarding the background profile of the participants, the majority of the testers (five out of seven) had 0 to 3 years of experience in the IT industry, with only two having more. Furthermore, all the testers currently work in different roles, ranging from

network engineers to front-end developers, and only two of them had previous experience with Ansible prior to this survey. Finally, 6 out of 7 testers found Ansible DSL learning to be easy or very easy, while only one found it normal.
Going to the core of our questionnaire, we asked the testers the questions Q6 until Q13 to measure the users' satisfaction in four different factors that assess various aspects of each editor and the provided language.
Table 5.3 contains the average results from questions Q6 until Q13 that allow us to compare the two editors.

Questionnaire		
Questions	**Ansible editor**	**Atom**
Q6	2.7	5
Q7(in minutes)	31.7	43
Q8	4	3.6
Q9	3.4	1.6
Q10	3.9	2.29
Q11	4.3	1
Q12	4.1	2.86
Q13	4.6	1

Table 5.3: Average Results of the Evaluation

As we can observe from the responses, the Ansible editor receives a larger satisfaction score on most of the questions than the Atom text editor, achieving the primary goal of our work, which is to facilitate developers when writing Ansible playbooks.

More specifically, we can see from Q7 that users needed 20% less time on average to develop Ansible playbooks with our Ansible editor than with the Atom text editor, which is to be expected given the toolset that the Ansible editor provides. The preference for the Ansible editor can also be seen in questions Q8, Q9, Q10, Q12, and Q13, where users gave very positive satisfaction scores compared to Atom in the features that improve the editor's usability and assist users in avoiding code errors. On the other hand, the Atom editor provides only features that check mainly for

syntax errors, such as the correct indentation, without checking for the semantics of the code, such as the type of the defined parameters.

Another important conclusion that we can make from question Q11 is that the users found the integration of our Ansible editor with TOSCA and the ability to exchange information with TOSCA models very valuable. On the contrary, the Atom editor does not enable users to work in the context of a TOSCA model and retrieve information directly via the editor.

However, users stated in question Q6 that installing the Ansible editor and the corresponding MongoDB was quite time-consuming when compared to the Atom editor, which only requires the installation of a software package. This is the tradeoff we have to pay between the editor's provided features and the complexity of its configuration procedure.

We also asked users to rate some of the Ansible editor's features in addition to the Likert-based questions. According to the testers, the most important feature was the integration with TOSCA, followed by the syntax proposal, the content validation mechanisms, and the Ansible Galaxy content proposal. Finally, the detection of code smells and syntax highlighting were the least preferred features from this pool of options.

Finally, for completeness, we asked the testers to report some of the advantages and disadvantages of our Ansible editor compared to Atom in open text questions. In this way, we can receive feedback beyond our questionnaire and see potential ideas for future improvements. More specifically, most users reported that the Ansible editor facilitates them to write faster and better Ansible playbooks by providing direct documentation for various concepts such as modules, collections, and roles, as well as suggesting Ansible Galaxy content. The Eclipse framework, on the other hand, in which the Ansible editor is housed, has been described as heavyweight, requiring a lot of RAM and more time to configure. Furthermore, some users stated that Eclipse is not a modern development environment and is not the best UI solution.

Threat to validity: A vital point that needs to be mentioned for our evaluation procedure is that all the testers have zero or little experience with Ansible. As a result, in a more experienced audience, the satisfaction scores may be different, leading us to different conclusions

6. Conclusion

6.1 Summary

Our work is part of the SODALITE project, which provides a rich toolkit for simplifying the deployment of a complex application. SODALITE assists users in resource provisioning by generating TOSCA artifacts in which the user defines the general structure of an application from a deployment standpoint. This approach, however, is insufficient for fully automating deployment tasks because it lacks support for creating and configuring software layers on top of provisioned resources. At this point, this thesis extends the SODALITE approach by forming a modeling environment in which users can create configuration scripts and seamlessly connect them to other parts of the deployment specification. Therefore, our contribution aims to supplement the deployment procedure of a complex application after provisioning the necessary resources, with a particular emphasis on improving the user's experience when developing Ansible scripts. To accomplish this, we enhanced the capabilities of the Ansible editor developed in the context of SODALITE by integrating external components and information sources in one place. In this manner, developers have access to a large pool of tools provided by the same editor, offering many different advanced features and receiving constant real-time feedback.

Having the initial version as a starting point, we expanded the Ansible editor's capabilities and focused on providing valuable features that streamline the development workflow. For example, autocompletion, error messages, and code suggestions are extremely useful for the user and significantly speed up the development process because he/she has all the necessary information in a single location without having to search through lengthy documentation. This is especially important in the case of Ansible, where Ansible collections, modules, and roles are dispersed across multiple repositories, requiring the user to strain for the desired information. As a result, our primary focus has been to connect the SODALITE IDE with information sources that provide the end-user with valuable suggestions and constant feedback. These sources of information are offered through intelligent

reasoning services running upon the Knowledge Base where SODALITE Resource Models are stored and a database containing Ansible content from Ansible Galaxy.

A significant extension we have introduced is the set of multiple validation mechanisms that check Ansible models for validity issues and provide clear and meaningful recommendations to the end-user on how to fix them. For example, the Ansible editor notifies the users if an Ansible collection name has the incorrect format and instructs them how to correct it. Such errors cause issues during the deployment and management of the cloud application and necessitate a significant amount of time to identify the mistakes in the source code. As a result, accurate error messages, accompanied by quick fixes whenever possible, can save the end-user time, increase their productivity, and boost users' satisfaction.

Taking semantic validation, a step further, we were inspired by static analyzers used in programming languages, which assess code quality by checking whether the developer followed some good development practices, and we wanted to provide this type of feedback via the Ansible editor. To accomplish this, we used another component developed in the context of SODALITE called Defect Predictor, which analyzes Ansible scripts and alerts on potential bad practices in them. Thus, the user can receive feedback on issues that do not directly affect the execution of the code but have an impact on its quality and maintenance.

In the same vein, we implemented code suggestions features to help users with the Ansible development process by indicating the appropriate options to write at each step. The content proposal component that manages this feature makes use of the KB and the MongoDB database to provide semantic code suggestions such as Ansible modules, parameters, and Ansible roles, among other things. In this way, users with little or no experience with Ansible syntax can create Ansible models by following the editor's instructions.

Another key feature of our work is the set of mechanisms that help the user define an Ansible model and import the generated scripts into the corresponding Resource Model. Following the definition of a Resource Model, the Ansible scripts that implement its TOSCA operations must be imported into the RM. As a result, we developed a standardized procedure that guides the user through the definition of the appropriate Ansible model, followed by the generation and integration of the corresponding Ansible script into the RM. In this manner, the user is not lost among the various DSLs that SODALITE IDE provides but instead remains in a specific chain of activities. One critical point to highlight here is that our work does not alter the transparency property of the Ansible script's origin. This property enables the user to select an Ansible script written with an external editor for a TOSCA operation rather than firstly defining an abstract Ansible model (i.e., .ans file) and then

generating the concrete Ansible script. Thus, the only requirement for importing an Ansible script is to specify the script's local path in the RM without restricting development to our Ansible editor.

6.2 Future Work

In this section, we propose some future contributions to the Ansible editor that will aid in the development of Ansible scripts and broaden the scope of our work.

Reverse translation

The current version of the Ansible editor supports the generation of an Ansible script from an abstract Ansible model. A beneficial extension to the Ansible editor would be to support the generation of Ansible scripts into Ansible models, which will allow users to exploit the capabilities of the Ansible editor for already implemented Ansible playbooks by generating the equivalent Ansible model. Then, the users can manipulate the generated Ansible model, make the desired changes, and create a new version of the Ansible playbook through the Generator component.

Complex projects

Aside from testing the expressiveness of Ansible DSL and measuring developer satisfaction when creating Ansible playbooks with the Ansible editor, it would be beneficial to test our approach in real-world case studies that require more complex projects than just playbooks. In that case, we could identify the effectiveness of the Ansible editor and its impact on development costs and user satisfaction.

Bibliography

[1] P. Mell and T. Grance, "The NIST definition of cloud computing (SP 800-145)," *NIST Spec. Publ*, vol. 145, p. 7, 2011.

[2] C. T. S. Xue and F. T. W. Xin, "Benefits and challenges of the adoption of cloud computing in business," *International Journal on Cloud Computing: Services and Architecture*, vol. 6, no. 6, Art. no. 6, 2016.

[3] S. N. Brohi and M. A. Bamiah, "Challenges and benefits for adopting the paradigm of cloud computing," *International Journal of Advanced Engineering Sciences and Technologies*, vol. 8, no. 2, Art. no. 2, 2011.

[4] T. Devi and R. Ganesan, "Platform-as-a-Service (PaaS): model and security issues," *TELKOMNIKA Indonesian Journal of Electrical Engineering*, vol. 15, no. 1, Art. no. 1, 2015.

[5] T. Laszewski and P. Nauduri, *Migrating to the cloud: Oracle client/server modernization*. Waltham, MA: Syngress, 2012. Accessed: Feb. 28, 2022. [Online]. Available: http://www.books24x7.com/marc.asp?bookid=44727

[6] L. E. Lwakatare, P. Kuvaja, and M. Oivo, "Dimensions of devops," in *International conference on agile software development*, 2015, pp. 212–217.

[7] M. Loukides, *What is DevOps?* O'Reilly Media, Inc., 2012.

[8] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[9] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," *Information and software technology*, vol. 64, pp. 1–18, 2015.

[10] I. Kumara *et al.*, "The do's and don'ts of infrastructure code: A systematic gray literature review," *Information and Software Technology*, vol. 137, p. 106593, 2021.

[11] "TOSCA Version 1.3." http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/TOSCA-Simple-Profile-YAML-v1.3.html

[12] "Ansible overtakes Chef and Puppet as the top cloud configuration management tool." https://www.techrepublic.com/article/ansible-overtakes-chef-and-puppet-as-the-top-cloud-configuration-management-tool/

[13] "SODALITE." https://www.sodalite.eu/

[14] E. Di Nitto *et al.*, "An approach to support automated deployment of applications on heterogeneous cloud-hpc infrastructures," in *2020 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 2020, pp. 133–140.

[15] L. Hochstein and R. Moser, *Ansible: Up and Running: Automating configuration management and deployment the easy way*. O'Reilly Media, Inc., 2017.

[16] "Ansible Collections." https://docs.ansible.com/ansible/latest/collections/index.html

[17] "Puppet." https://puppet.com/docs/puppet/6/puppet_language.html

[18] J. Loope, *Managing infrastructure with puppet: configuration management at scale*. O'Reilly Media, Inc., 2011.

[19] M. Marschall, *Chef infrastructure automation cookbook*. Packt Publishing Ltd, 2015.

[20] "Chef Overview." http://man.hubwiz.com/docset/Chef.docset/Contents/Resources/Documents/docs.chef.io/chef_overview.html

[21] C. Myers, *Learning saltstack*. Packt Publishing Ltd, 2016.

[22] "SaltStack Overview." https://searchitoperations.techtarget.com/definition/SaltStack

[23] J. O. Benson, J. J. Prevost, and P. Rad, "Survey of automated software deployment for computational and engineering research," in *2016 Annual IEEE Systems Conference (SysCon)*, 2016, pp. 1–6.

[24] "Configuration management tools - Comparison." https://medium.com/successivetech/chef-vs-puppet-vs-ansible-vs-saltstack-a-complete- comparison-9af8f1790c0d

[25] L. Baresi, E. Di Nitto, P. Mitzias, D. Radolović, K. Meth, and Y. Gorroñogoitia, "D2.1 Requirements, KPIs, evaluation plan and architecture - First version", [Online]. Available: https://www.sodalite.eu/reports/d21-requirements-kpis-evaluation-plan-and-architecture-first-version

[26] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, "TOSCA: portable automated deployment and management of cloud applications," in *Advanced Web Services*, Springer, 2014, pp. 527–549.

[27] A. Brogi and J. Soldani, "Matching cloud services with TOSCA," in *European Conference on Service-Oriented and Cloud Computing*, 2013, pp. 218–232.

[28] T. Binz, G. Breiter, F. Leyman, and T. Spatzier, "Portable cloud services using tosca," *IEEE Internet Computing*, vol. 16, no. 3, Art. no. 3, 2012.

[29] M. Zimmermann, U. Breitenbücher, and F. Leymann, "A TOSCA-based Programming Model for Interacting Components of Automatically Deployed Cloud and IoT Applications:," in *Proceedings of the 19th International Conference on Enterprise Information Systems*, Porto, Portugal, 2017, pp. 121–131. doi: 10.5220/0006332501210131.

[30] A. Brogi, J. Soldani, and P. Wang, "TOSCA in a Nutshell: Promises and Perspectives," Berlin, Heidelberg, 2014, pp. 171–186.

[31] "OpenStack-IaaS platform," *Openstack*, Mar. 26, 2022. https://www.openstack.org/

[32] "Heat OpenStack." https://wiki.openstack.org/wiki/Heat

[33] G. Katsaros, M. Menzel, A. Lenk, J. Rake-Revelant, R. Skipp, and J. Eberhardt, "Cloud Application Portability with TOSCA, Chef and Openstack," in *2014 IEEE International Conference on Cloud Engineering*, Boston, MA, Mar. 2014, pp. 295–302. doi: 10.1109/IC2E.2014.27.

[34] "AWS CloudFormation." https://docs.aws.amazon.com/cloudformation/index.html

[35] "Terraform," *https://www.terraform.io/intro/index.html*.

[36] Mi. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, "Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Cleveland, OH, USA, Sep. 2019, pp. 580–589. doi: 10.1109/ICSME.2019.00092.

[37] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "A systematic mapping study of infrastructure as code research," *Information and Software Technology*, vol. 108, pp. 65–77, Apr. 2019, doi: 10.1016/j.infsof.2018.12.004.

[38] L. Leite, C. Rocha, F. Kon, D. Milojicic, and P. Meirelles, "A Survey of DevOps Concepts and Challenges," *ACM Comput. Surv.*, vol. 52, no. 6, Art. no. 6, Nov. 2020, doi: 10.1145/3359981.

[39] A. Bergmayr *et al.*, "A Systematic Review of Cloud Modeling Languages," *ACM Comput. Surv.*, vol. 51, no. 1, Art. no. 1, Jan. 2019, doi: 10.1145/3150227.

[40] M. Wurster *et al.*, "The essential deployment metamodel: a systematic review of deployment automation technologies," *SICS Softw.-Inensiv. Cyber-Phys. Syst.*, vol. 35, no. 1–2, Art. no. 1–2, Aug. 2020, doi: 10.1007/s00450-019-00412-x.

[41] M. Wurster, U. Breitenbücher, L. Harzenetter, F. Leymann, J. Soldani, and V. Yussupov, "TOSCA Light: Bridging the Gap between the TOSCA Specification and Production-ready Deployment Technologies:," in *Proceedings of the 10th International Conference on Cloud Computing and Services Science*, Prague, Czech Republic, 2020, pp. 216–226. doi: 10.5220/0009794302160226.

[42] G. Casale *et al.*, "RADON: rational decomposition and orchestration for serverless computing," *SICS Softw.-Inensiv. Cyber-Phys. Syst.*, vol. 35, no. 1–2, Art. no. 1–2, Aug. 2020, doi: 10.1007/s00450-019-00413-w.

[43] J. Sandobalin, E. Insfran, and S. Abrahao, "An Infrastructure Modeling Approach for Multi-Cloud Provisioning," 2018.

[44] J. Sandobalin, E. Insfran, and S. Abrahao, "ARGON: A Tool for Modeling Cloud Resources," in *Service-Oriented Computing – ICSOC 2017 Workshops*, vol. 10797, L. Braubach, J. M. Murillo, N. Kaviani, M. Lama, L. Burgueño, N. Moha, and M. Oriol, Eds. Cham: Springer International Publishing, 2018, pp. 393–397. doi: 10.1007/978-3-319-91764-1_37.

[45] M. Artac, T. Borovšak, E. Di Nitto, M. Guerriero, D. Perez-Palacin, and D. A. Tamburri, "Infrastructure-as-code for data-intensive architectures: A model-driven development approach," in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 156–15609.

[46] G. Horn and P. Skrzypek, "MELODIC: utility based cross cloud deployment optimisation," in *2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, 2018, pp. 360–367.

[47] J. Alonso, C. Joubert, L. Orue-Echevarria, M. Pradella, and D. Vladušič, "PIACERE: Programming trustworthy Infrastructure As Code in a Secure Framework," 2021.

[48] H. Myrbakken and R. Colomo-Palacios, "DevSecOps: a multivocal literature review," in *International Conference on Software Process Improvement and Capability Determination*, 2017, pp. 17–29.

[49] "xOpera Orchestrator." https://xlab-si.github.io/xopera-docs/01-intro.html

[50] E. Imperiali, “Providing high-quality support for Ansible development,” Master Thesis, Politecnico di Milano.

[51] “IntelliJ framework,” *IntelliJ IDEA*, Mar. 27, 2022. https://www.jetbrains.com/idea/

[52] “VisualStudio IDE,” *VisualStudio IDE*, Mar. 27, 2022. https://visualstudio.microsoft.com/

[53] “How to select an Ansible collection.” https://opensource.com/article/21/3/ansible-collections

[54] “Xtext framework,” Mar. 27, 2022. https://www.eclipse.org/Xtext/

[55] J. Gorronogoitia, Z. Vasileiou, E. Imperiali, I. Kumara, D. Radolovic, and G. Meditskos, “A Smart Development Environment for Infrastructure as Code,” presented at the CEUR Workshop, Virtual Conference, Mar. 2021.

[56] “RedHat’s certified Ansible collections.” https://access.redhat.com/articles/3642632

[57] “Content Scoring - Ansible Galaxy.” https://galaxy.ansible.com/docs/contributing/content_scoring.html

[58] “Defect Prediction.” https://github.com/SODALITE-EU/defect-prediction

[59] F. Hermans, M. Pinzger, and A. van Deursen, “Domain-specific languages in practice: A user study on the success factors,” in *International Conference on Model Driven Engineering Languages and Systems*, 2009, pp. 423–437.

A. Appendix A

This appendix presents the detailed responses we received from the participants of the survey. The Table A.1 outlines the profile of each tester while the Table A.2 and Table A.3 present the feedback we got related to the usage of Ansible editor and Atom editor respectively.

Testers' Background							
Questions	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**
Q1(in years)	0-3	0-3	3-5	0-3	0-3	3-5	0-3
Q2	Network engineer	Data Analyst	System Administrator	Junior Machine Learning Engineer	Full Stack Software Developer	Product manager	Front-end developer
Q3	Beginner	None	None	Beginner	None	None	None
Q4	Through CCNA preparation	None	From the present project.	From work	From the current thesis	Used to interact with companies as a cloud architect.	General knowledge

Table A.1: Background of the participants

Ansible editor - Questionnaire								
Questions	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**Average**
Q5	4	4	4	4	4	3	5	4
Q6	3	3	2	2	3	2	4	2.7
Q7(in minutes)	40	45	30	27	50	15	15	31.7
Q8	4	4	3	5	3	5	4	4
Q9	4	4	3	4	4	2	3	3.4
Q10	4	4	4	5	4	3	3	3.9
Q11	4	4	4	5	4	5	4	4.3
Q12	4	4	4	4	4	5	4	4.1
Q13	5	5	4	5	5	5	3	4.6

Table A.2: Ansible editor - Evaluation Results

Atom editor - Questionnaire								
Questions	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**Average**
Q6	5	5	5	5	5	5	5	5
Q7(in minutes)	60	60	45	35	60	24	18	43
Q8	4	4	4	4	3	3	3	3.6
Q9	1	1	1	1	3	2	2	1.6
Q10	3	2	2	3	3	2	1	2.29
Q11	1	1	1	1	1	1	1	1
Q12	2	2	3	4	3	2	4	2.86
Q13	1	1	1	1	1	1	1	1

Table A.3: Atom editor - Evaluation Results

List of Figures

Figure 2.1: Ansible playbook 11
Figure 2.2: TOSCA service template 18
Figure 2.3: SODALITE overview 26
Figure 3.1: Content of the Knowledge Base 34
Figure 3.2: Ansible Model importing a node type from a local RM 35
Figure 3.3: Error after retrieve RM from KB 36
Figure 4.1: Architecture 40
Figure 4.2: Resource Model example 45
Figure 4.3: Folder organization 46
Figure 4.4: Select a resource type from a Resource Model stored in KB 47
Figure 4.5: Select interface 48
Figure 4.6: Select operation 48
Figure 4.7: Select operation input as variable 49
Figure 4.8: Resource Model example 50
Figure 4.9: Database schema 53
Figure 4.10: Select Ansible namespace 53
Figure 4.11: Select Ansible collection 54
Figure 4.12: Select module from the pool of the imported collections 54
Figure 4.13: Select module through its fully qualified name 55
Figure 4.14: Select parameter for the chosen module 55
Figure 4.15: Default value of the selected parameter 56
Figure 4.16: Subparameters for complex parameters 56
Figure 4.17: Acceptable values of the requested parameter 56
Figure 4.18: Select role from the pool of the imported collections 57
Figure 4.19: Select role belonging to an Ansible collection through its fully qualified name 57

Figure 4.20: Select Ansible namespace 58
Figure 4.21: Select role contained in the chosen Ansible namespace 58
Figure 4.22: Unsupported Ansible collection 59
Figure 4.23: Wrong Ansible collection name format 59
Figure 4.24: Wrong Ansible module name format 60
Figure 4.25: Wrong value type 60
Figure 4.26: Available values to be assigned 61
Figure 4.27: Missing mandatory parameters 61
Figure 4.28: Missing mandatory subparameters 61
Figure 4.29: Invalid parameter 62
Figure 4.30: Example playbook 62
Figure 4.31: Code smells 63

List of Tables

Table 2.1: Comparison of configuration tools 13
Table 2.2: Comparison of provisioning tools.. 20
Table 5.1: Questions of the evaluation procedure 68
Table 5.2: Questionnaire 70
Table 5.3: Average Results of the Evaluation 71
Table A.1: Background of the participants 83
Table A.2: Ansible editor - Evaluation Results 84
Table A.3: Atom editor - Evaluation Results.. 84

Acknowledgements

I would like to thank my supervisor, Professor Elisabetta Di Nitto, for her guidance, support, and invaluable feedback throughout my thesis and for the opportunity she provided me to collaborate and experience with many different researchers throughout Europe in the context of a European project. I must also thank the project partners Yosu Gorroñogoitia and Zoe Vasileiou for their support and helpful advice throughout my thesis.

I am extremely grateful to my parents for the sacrifices they have made in order to support my dreams in the best possible way. Their love and support kept me motivated and confident during my studies' long and challenging journey. Thank you!