TÉCNICO LISBOA

Int2IT: An Intent-based TOSCA IT Infrastructure Management Platform

Manuel Duarte Mascarenhas

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. José Carlos Martins Delgado
Doctor Rui António dos Santos Cruz

Examination Committee

Chairperson: Prof. António Manuel Ferreira Rito da Silva
Supervisor: Prof. José Carlos Martins Delgado
Member of the Committee: Prof. Rodrigo Fraga Barcelos Paulus Bruno

October 2022

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

Acknowledgments

I would like to thank my parents, brother and sister for their immense support and guidance, both at a personal and academic level. They made every major obstacle encountered bearable and every small advancement feel like a great victory. Without them this project would not be possible in any capacity.

I would also like to thank my university friends for the camaraderie we developed over these last five years. For the long hours of working late at night and for the immense fun and adventures we shared throughout our academic journey. May this community we have built together last for many more years. In particular, I would like to distinguish Miguel Levesinho, Gonçalo Freire and João Galamba for their never-ending and ever present support, friendship, guidance and trust.

I would also like to thank my friends who have been with me for the better part of a decade or even most of my life. Thank you for joining me through the best of times and for making the darkest of times endurable. A special thank you to José Pombal, Pedro Santos, Guilherme Fernandes, Beatriz Thomaz and Carolina Branco for all the support you have given me, especially in this last year on this thesis.

I would also like to thank my high school professors without whom I would not be the person that I am today. I thank them for their compassion, friendship and for the close relationship that persists to this day. Specifically, I want to distinguish Sara Figueira, Ana Morgado, Marta Mota, Pedro Manso and Pedro Castanheira. All of you have influenced me in many ways and left me great life lessons, some intentional and others by accident but all appreciated equally.

Last, but not least, I would also like to acknowledge my dissertation supervisors Prof. Rui Cruz and Prof. José Delgado for their insight, support and sharing of knowledge that has made this Thesis possible.

To each and every one of you – Thank you.

Abstract

The introduction and widespread adoption of cloud computing has opened the door to the possibility of designing and building large scale systems with tremendous raw computing capabilities for dealing with an ever-increasing volume of data. However, the infrastructure required to support these systems became too complex for manual efforts, so practices such as Infrastructure-as-Code (IaC) and Development and Operations (DevOps) methodologies with programmatic orchestration and provisioning of cloud infrastructures became increasingly common. This led to the creation of the OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) standard, which has seen extensive research by academia but infrequent adoption by the industry, as, despite simplifying partly the management of infrastructures, lacks the capacity of encapsulating the desired behaviour of the system and its business-level end-goals. Int2IT is presented here as a solution to this specific problem alongside a proof-of-concept implementation named Int2IT-Lite. It is an "intent-based" infrastructure management platform that incorporates autonomic computing concepts in order to manage cloud deployments autonomously, using a TOSCA-based cloud application description. The proposed solution will be able to capture the user's "intents", which describe the system's end-goals, and translate them into a TOSCA-based cloud application. As a result, it can then be deployed to the cloud and autonomously managed, by utilizing a Monitor-Analyze-Plan-Execute over a shared Knowledge (MAPE-K) autonomic deployment life-cycle capable of adapting to the outside environment, ensuring that the end-goals are met to the greatest extent possible.

Keywords

Infrastructure-as-Code (IaC); DevOps; Cloud Computing; Intent-Based; OASIS TOSCA.

Resumo

A introdução e ampla adoção da computação em nuvem abriu as portas para a construção de sistemas de grande escala, com enormes capacidades de computação e para lidar com volumes crescentes de dados. No entanto, a infraestrutura necessária para dar suporte a esses sistemas tornou-se manualmente muito complexa, pelo que as recentes práticas de *Infrastructure-as-Code (IaC)* e *Development and Operations (DevOps)*, com orquestração programática e provisionamento em núvem, tornaram-se cada vez mais comuns. Isso levou à criação do padrão OASIS *Topology and Orchestration Specification for Cloud Applications (TOSCA)*, que tem visto extensa investigação ao nível académico, mas pouca adoção pela indústria, pois, apesar de simplificar, a administração de infraestruturas, não tem a capacidade de encapsular o comportamento desejado do sistema e seus objetivos a nível de negócio. O Int2IT é apresentado como uma solução para esse problema, juntamente com a implementação de uma prova de conceito designada Int2IT-Lite. É uma plataforma de administração de infraestruturas à base de "*intents*" que incorpora conceitos de computação autonómica, usando uma descrição de aplicações baseada em TOSCA, sendo capaz de capturar as "*intents*"que descrevem os objetivos finais do sistema, e traduzi-las num projeto que pode ser lançado na núvem e gerido de forma autónoma, utilizando um ciclo de vida de administração *Monitor-Analyze-Plan-Execute over a shared Knowledge (MAPE-K)*, garantindo que os objetivos finais sejam cumpridos.

Palavras Chave

Infrastructure-as-Code (IaC); DevOps; Computação na Nuvem; *Intent-Based*; OASIS TOSCA.

Contents

List of Figures

List of Algorithms

Acronyms

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Program Interface
AWS	Amazon Web Services
BPMN	Business Process Model and Notation
CLI	Command Line Interface
CI/CD	Continuous Integration/Continuous Delivery
CISTI	Iberian Conference on Information Systems and Technologies
CP	Cloud Provider
CPU	Central Processing Unit
DevOps	Development and Operations
DSL	Domain Specific Language
EDMM	Essential Deployment Metamodel
GUI	Graphical User Interface
GCP	Google Cloud Platform
IBN	Intent Based Networking
IaC	Infrastructure-as-Code
IT	Information Technology
MAPE-K	Monitor-Analyze-Plan-Execute over a shared Knowledge
NLP	Natural Language Processing
OASIS	Organization for the Advancement of Structured Information Systems
QoS	Quality Of Service
REST	Representational State Transfer

TOSCA Topology and Orchestration Specification for Cloud Applications

XML Extensible Markup Language

YAML YAML Ain't Markup Language

1

Introduction

Contents

The cloud computing paradigm has forever changed the landscape of Information Technology (IT) in terms of raw processing capability and high availability services, which is evident by the ever growing number of organizations who, in the last decade, perceive the cloud as a cost-effective and scalable solution that contrasts with the traditional acquisition, provisioning and management of local infrastructure.

Cloud providers such as Google Cloud Platform (GCP), Amazon Web Services (AWS) and Microsoft Azure offer a multitude of resource virtualization services diverse in capacity and price, ranging from infrastructure provisioning to data-driven serverless computing capabilities. Naturally, this wide range of services resulted in an exponential growth in complexity, thus its management has become increasingly important, which led to the creation and development of tools and methodologies designed to scale down such complexity, allowing for a more automated control over the cloud environment and enabling a more systematic approach to application deployment supported by version control and modular releases.

The Development and Operations (DevOps) culture is the practice of blending the traditional Development and Operations teams into a unified unit operating under a Continuous Integration/Continuous Delivery (CI/CD) life-cycle, where there is a focus on frequent smaller releases, the "need-for-speed" alluded to as per Artac et al. [1]. This life-cycle is supplemented with continuous testing and version control through "as-Code" practices yielding in re-usability, greater error detection and error correction prior to full production deployment.

Infrastructure-as-Code (IaC) is one of these practices which attempts to enable descriptions of complex infrastructure deployments in a programmatic manner, typically in a declarative fashion, despite the initial trend geared towards traditional imperative programming, as per Bellendorf et al. [2]. In the past, it has been described as "the DevOps practice of describing complex and (usually) Cloud-based deployments by means of machine-readable code." as per Guerriero et al. [3]. The main advantage of this practice is that it allows for a systematic approach to the management of infrastructure provisioning, avoiding typical human induced errors whilst simultaneously enabling a "fast track" developmental approach for the provisioning and maintenance of machine instances. Nevertheless, this practice is not without its downsides, such as the lack of standardization at an industry level, the prevalence of bad practices such as "hardcoding" and the lack of design time tools with inherent testing capabilities.

The Organization for the Advancement of Structured Information Systems (OASIS) Topology and Orchestration Specification for Cloud Applications (TOSCA) presents itself as a standardized implementation of IaC widely studied in academia whilst having a small market share in business environments, despite appraisal and interest from the experts who have utilized it, according to Guerriero et al. [3]. TOSCA has three main objectives, as per Brogi et al. [4]:

- "Automated application deployment and management";
- "Portability of application descriptions and their management";

- “Interoperability and reusability of components”.

1.1 Motivation

Unfortunately, DevOps methodologies and IaC practices can only go so far and, while they have a measurable effect on diminishing impactful human-prone errors, they may have even increased the likelihood of “operator errors”, as per Vetter et al [5], due to the increase in deployment releases and to dependencies on low-level changes in requirements, for example, changes in ports to be used in an application context.

Despite the fact that these methodologies and practices enable a more systematic approach to the application life-cycle through continuous testing and version control, there is a looming reliance on systems administrators’ knowledge of specific details and requirements, diverting their attention away from the holistic perspective of said system. Furthermore, while IaC allows for a program-like description of the system, it fails to capture the system’s business-level holistic perspective; in other words, it does not allow for a description of the system’s requirements or the metrics that allow their verification. In today’s world, systems have grown to such complexity and scale that manual efforts to manage them have become unfeasible, error-prone, and inefficient, according to Kephart [6].

As a result, the adoption and development of autonomic systems, was required to shift some of the administrators’ responsibilities in order to achieve freedom from operation and maintenance details, while keeping sets of machines operating at their maximum capabilities at all times, as per Kephart [6].

Nonetheless, autonomic systems will need to be provided with intentions and policies, by their administrators, such that these constraints guide their actions towards the desired optimal system, based on high-level end-goals, as per Kephart [6].

These are the exact conditions which this dissertation attempts to contribute to solve, with the development of a proof-of-concept, named **In2IT**, for an autonomic intent-based OASIS TOSCA infrastructure management tool.

The developed tool aims to expand and demonstrate the work proposed in a paper that was published and presented in the Iberian Conference on Information Systems and Technologies (CISTI) under the same name as this dissertation [7].

1.2 Goals

Int2IT is an Intent-Based autonomic infrastructure management platform that uses OASIS TOSCA for IaC standardization and infrastructure provisioning. Inspired by the recent trend of Intent Based Networking (IBN), Int2IT adopts an intent-based approach which, in its essence, allows the creation of

infrastructure and respective policies based on a higher-level abstraction of descriptions which specify the end-goals to be achieved rather than the manner of achieving them.

Additionally, through the derived policies obtained, Int2IT is capable of self-adapting its modular infrastructure deployments, in an autonomic fashion based on the Monitor-Analyze-Plan-Execute over a shared Knowledge (MAPE-K) model, in order to match the intended end-goals specified by developer's, thus minimizing human intervention, and consequently mitigating the aforementioned operator errors.

Lastly, Int2IT aims at expanding previous works on TOSCA-based orchestration and infrastructure which have not focused on bridging business-level logic and pure academic research. Therefore, the goals of the dissertation can be summarized as:

- Introduce intent-based descriptions to infrastructure deployments;
- Adopt an autonomic perspective in the infrastructure deployments to provide adaptability so as to comply to business-level metrics;
- Expand on previous TOSCA orchestration related work while motivating further research into automation and simplification of the operational side to systems development.

1.3 Organization of the Document

The remainder of this document is organized as follows. Chapter 2 presents related work with varied important concepts and definitions which are essential for understanding Int2IT and Int2IT, as well as a timeline of TOSCA academic contributions on top of which Int2IT will be built upon and, finally, introducing challenges for future work which will not be covered in this project, but are nonetheless important and in need to be addressed in future works. Chapter 3 provides a full overview of the proposed architecture of Int2IT complemented by models and use cases whenever necessary. Chapter 4 showcases the implementation of a proof-of-concept of Int2IT, a bare-bones approach named Int2IT-Lite, depicting the main requirements and obstacles that need to be addressed. Chapter 5 presents the methodology utilized to demonstrate the capabilities of Int2IT-Lite. A use case is created, showcased and the chapter concludes by drawing observations from the results achieved, making clear the success of the implemented solution. Finally, Chapter 6 draws conclusions about the research work done so far and the forthcoming developments.

2

Background and Related Works

Contents

In this section some background concepts and clarifications are provided for the most relevant topics that this dissertation will contribute to, such as Intent-Based and Autonomic systems, as well as focusing on the primary topic of the OASIS TOSCA standard and its current state of the art, whilst at the same time, reviewing and introducing a subset of previous works and their contributions.

2.1 Intent-Based

Intents can be defined as a high-level abstraction for specifying a system's end goals without mentioning how those goals will be achieved or delving into concrete low-level details, and they are frequently defined in a more "natural" language rather than a more programmatic declaration. *Intents* can provide qualitative definitions of service quality as well as quantitative thresholds for the metrics associated with said services. Davoli et al [8] define both Quality Of Service (QoS) features and thresholds as, respectively, being qualitative towards the specified service, for example "guaranteed bit rate or limited delay" and quantitative towards a specific metric of interest, e.g., "minimum bit rate or a maximum delay value".

Nefkens [9] presents a great analogy where, when designing a large office building, the blueprints offer guidelines and a perspective of the future overall outline aspect of the building, nevertheless, it does not supply the contractor's with the specifications of the materials to be used during construction nor the functionality which the office's will provide. Since intentions do not specify the "how" to achieve the defined end-goals, there is a need to translate them into procedural steps which can then be applied so as to meet the end goal desired. One good example of this translation is provided, as well, by Nefkens [9]. Consider the intention of "taking out the trash in the kitchen". We can deconstruct this into several steps, for example:

- Taking trash from bin in kitchen and wrapping it;
- Carry it outside and dump it in the containers;
- Replenish the bin's trash bag.

Intents are frequently confused with the concept of policies, but conceptually, despite some similarities, they refer to different abstractions. As previously stated, *intents* are concerned with the high-level outcomes of a system without defining events or actions. Policies, on the other hand, are collections of rules that define actions to be taken when certain conditions are met but do not refer to desired outcomes. For example, a possible definition of an *intent* could be "The database server's Central Processing Unit (CPU) utilization should be maintained between 15 and 75 percent", whereas a policy could be *If the database server's CPU utilization exceeds 75 percent or is less than 15 percent, then modify the number of replicas by X*.

2.2 Autonomic Computing

Autonomic computing surges as a paradigm designed to handle the increasing complexity, heterogeneity and dynamism in services and applications by emulating strategies employed by biological systems, as per Hariri et al. [10]. Specifically it has its basis in the autonomous nervous system present in humans which is capable of adapting, in face of uncertainty, so as to ensure the survival of the system. Hariri et al. [10] also remark that the system, when conditioned, internally or externally, will attempt to return to a state of equilibrium.

Restoring the system to said state however, requires the capability of self-adaptation often described as the **self-*** properties which can be deconstructed into four elements, being: self-configuration, self-optimization, self-healing and self-protection, as per Kephart and Chess [6].

For the solution here proposed, it will mainly cover the capability of self-configuration and self-optimization, which is present in the automatic creation, provisioning and optimization of a specific deployment's infrastructure. Although not being heavily focused on, there is still some capabilities to self-heal in the form of recovery from certain failures, such as an operation which could not be completed successfully.

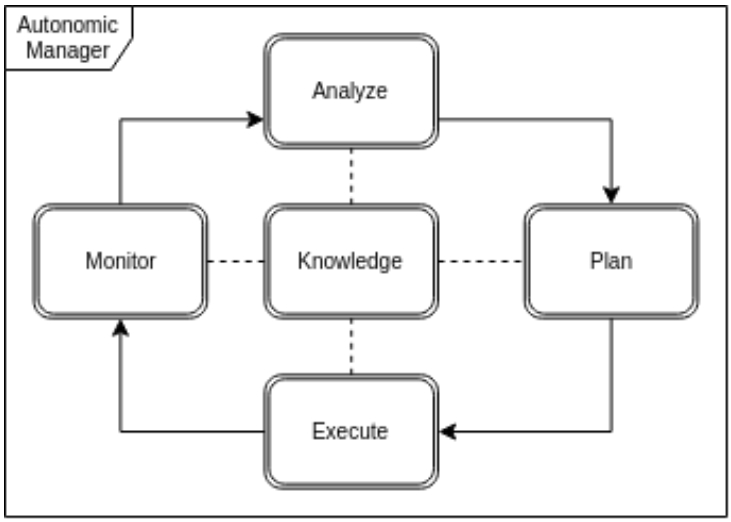

Figure 2.1: Representation of the MAPE-K model

While Int2IT is not an autonomic tool, it does leverage from some of the **self-*** principles previously mentioned, when managing cloud deployments. In particular, it employs a self-adapting life-cycle model similar to the well-known MAPE-K model, which represents the components of an autonomic manager that manages a single element. As shown in Figure 2.1, the model is made up of five components: Monitor, Analyze, Plan, Execute, and Knowledge. In Chapter 3, these will be explored in more detail as they are represented directly by the components in deployment management life-cycle.

2.3 OASIS TOSCA

The OASIS TOSCA standard presents itself as a language standardization in the realm of IaC , allowing for the definition and modelling applications and services through a declarative definition in a language named YAML Ain't Markup Language (YAML), despite originally geared towards a more Extensible Markup Language (XML)-like definition. TOSCA is capable of covering the entire life-cycle of cloud-based environments by depicting, as Lipton et al. [11] remarked, "their components, relationships, dependencies, requirements and capabilities of orchestrating software". Additionally, TOSCA also allows the definition of policies which can be used for the purposes of automatic scaling of applications, as presented in Cankar et al. [12]. At the same time, it is capable of specifying policies which cover, as Waizenegger et al. [13] demonstrated, non-functional requirements, such as cost or security.

TOSCA application topology is a directed graph structure made up of nodes and edges that serves as the foundation for TOSCA application description. A node in the graph represents a component of the application, regardless of the application layer to which it contributes, in other words, a node can refer to software elements or even physical components, whereas edges depict the relationships between nodes. The "Service Template" of TOSCA encompasses all the major elements that define a service which includes the application topology and the types of nodes and relationships, as well as the "Plans" that manage the complete service life-cycle, as per OASIS TOSCA standard [14] and shown in Figure 2.2(source: [14]).

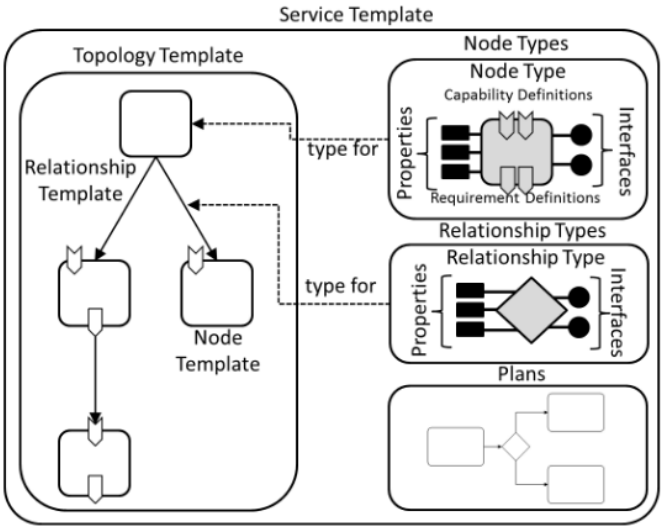

Figure 2.2: Representation of the Service Template from OASIS TOSCA standard version 1.0

Following this overview of some background information on the core concepts of TOSCA, a perspective on previous works and their main contributions will be presented, as well as some context on works done at a similar time.

2.4 Timeline of TOSCA Works

The following section intends to provide a high level overview of some of the most relevant works accomplished by some of the most influential authors and experts in the field of TOSCA IaC.

2.4.1 The Building Blocks (2012-2014)

Winery [1] is a web-based graphical TOSCA modeling tool that supports the entire OASIS TOSCA standard as well as type definition. It is made up of three parts: the Topology modeler, the Element Manager, and the Repository. Winery, as a tool, is intended to be used in conjunction with other tools that can benefit from its modeling capabilities such as OpenTOSCA, presented by Binz et al. [15], an imperative runtime for processing TOSCA applications, where the deployment and management flow is based on defined plans that are then executed by the engine.

Waizenegger et al. [13] present a formal policy definition based on the OASIS TOSCA standard, in which each policy is defined based on its impact, the stage of the cloud service life-cycle, the topology layer, and the effect to be produced. Furthermore, policies define the property over which the policy operates and store the desired value, such as a database encryption policy with an AES256 property denoting the encryption type. An offering consists of multiple policies and a formal TOSCA service template, which users will select based on whether it meets their needs. Finally, the authors present two approaches for policy-aware cloud provisioning: plan-based (P-Approach) and implementation artifact based (IA-Approach).

2.4.2 Advanced tooling and connectivity (2014-2018)

Brogi et al. [16] present systematic mappings of TOSCA interconnection constraints to formal conditions that must be guaranteed for a TOSCA application to be valid, as well as the *Sommelier* validation prototype.

The *Sommelier* validation prototype (created by Brogi et al. [17]), helps TOSCA application developers by validating application topologies and ensuring that they meet all interconnection restrictions, allowing for validation at design time, which was previously done manually.

Brogi et al. [17] also present a solution for improved support in the deployment and management of cloud applications based on TOSCA and Docker[2]. The authors contributions are twofold: a new TOSCA-based representation that allows users to depict solely the elements that compose the application and the associated software dependencies that each requires; the TosKerizer tool, which can find

[1] https://projects.eclipse.org/projects/soa.winery
[2] https://www.docker.com

a “best-fit” completion of the necessary Docker containers for the application, from incomplete TOSCA specifications.

Sampaio et al. [18] present a novel approach that uses Cloudify[3] and a Cloud Crawler to automate the performance of cloud topologies based on the OASIS TOSCA standard and their many possible configurations. Essentially, the user specifies the various topology specifications for that particular application, as well as the test parameters required for deployment evaluation, which is then broken down into scenarios by the Cloudify Manager and provided to Cloud Crawler, while Cloudify orchestrates the deployment to the specified Cloud Provider (CP). The Cloud Crawler is then in charge of evaluating each of the virtual machines that comprise the deployment using the specified parameters and reporting the results to the users.

2.4.3 Multi-component tooling and meta-analysis (2019-2021)

Tamburri et al. [19] provide an overview and key designs of TOSCA-based intent modeling concepts and their main properties provided in TOSCA using concise and representative examples: its substructural hierarchy, symmetric idempotence, Top-down intentionality, higher-order scope, meta-centric design, and, finally, resource-centric intent unfolding.

Wurster et al. [20] present TOSCA Light as a subset of TOSCA that is compliant with the Essential Deployment Metamodel (EDMM), with the goal of closing the gap between the state of academic research and the industry. EDMM’s have been shown to be able to be transformed into various deployment automation technologies, such as Terraform[4], therefore TOSCA Light being EDMM-compliant enables the translation of technology-agnostic descriptions into more tool specific descriptions. Furthermore, the authors present the TOSCA Light end-to-end toolchain, implemented as a proof-of-concept, which essentially depicts the flow from a TOSCA deployment model that is validated for TOSCA Light compliance, which is then transformed into a tool specific deployment model that can be executed by said tool.

Bellendorf et al. [2] present the findings of a systematic review of the literature on TOSCA in which they identify the main contributions to existing research, emerging trends in academic works, and potential topics for future research. Previous works, the authors conclude, have primarily focused on the application of TOSCA, methodologies for processing TOSCA models, or provided extensions to the standard.

Bogo et al. [21] present a novel approach for deploying multi-component applications defined by the OASIS TOSCA standard specification, while remaining decoupled from the containers that host each component, using existing (and industry standard) container orchestrators such as Docker Swarm

[3] https://cloudify.co
[4] https://www.terraform.io

and Kubernetes[5]. The authors present a toolchain comprised of the TOSKOSE Packager, Unit, and Manager, the first of which extracts Docker-based artifacts from TOSCA specifications, the second of which manages components present in a container, and the third of which manages the entire TOSKOSE Units that comprise the application.

TORCH, as presented by Tomarchio et al. [22], intends to address multi-cloud orchestration, a recent trend in cloud computing, while also combating vendor lock-in and promoting a more equitable marketplace in cloud services. The approach is primarily concerned with converting TOSCA application models into Business Process Model and Notation (BPMN) work and dataflows that can then be processed by any "off-the-shelf" BPMN engine. After processing the model, TORCH employs "pluggable connectors", as defined in the paper, that can be deployed on a variety of cloud providers. As a result, all coding efforts are now focused on developing these additional plugins, which are responsible for connecting BPMN workflows to platform-specific Application Program Interfaces (APIs). The connectors used and developed by the authors were specifically based on two of the industry's most popular orchestration tools, Kubernetes and Docker Swarm.

For Int2IT's purposes, the concept of connectors for various orchestration industry tools is quite beneficial as a means of deploying the plethora of TOSCA topologies generated, in different cloud providers, for the respective described intentions.

DesLauriers et al [23] utilize a TOSCA-based approach which serves as the basis for the description of an application in MiCADO, a framework for applications with multi-cloud and auto-scaling capabilities. In their paper, the authors demonstrate, in essence, the ease in portability and flexibility, provided by MiCADO, when changing from a deployment in AWS to a deployment on the private OpenStack cloud of the University of Westminster and, finally, to Microsoft Azure.

2.5 Summary of Related Work

Int2IT has as foundations three core concepts: (a) Intent-Based; (b) Autonomic Computing; and (c) the OASIS TOSCA standard. As such, Int2IT consumes high-level human-readable *Intents* that describe a cloud deployment; autonomously adapts so as to comply with said *Intents*; while following the OASIS TOSCA IaC standard.

Intent-based systems have been a recent growing trend in other fields, specifically in IBN, because they allow for a more goal-oriented system without having to be derailed by low-level implementation details.

Autonomous Computing is the concept of modelling technological systems after biological systems, concretely the human anatomical system. It focuses on creating resilient systems that adhere to four

[5] https://kubernetes.io

main principles: (a) self-configuration; (b) self-optimization; (c) self-healing; and (d) self-protecting.

TOSCA has been extensively studied in academia with initial works that provided fundamentally necessary tools to be able to fully take advantage of the standard. Afterwards, a period of development of advanced tooling ensued, bringing forth a collection of tools that provided advanced blueprint validation as well as some connectivity to other tools in the industry. Most recently, composite tools have been emerging that take advantage of the previous two eras to achieve more complex objectives such as addressing multi-cloud orchestration and auto-scaling of deployments.

Int2IT differs from previous approaches by creating an abstraction on top of TOSCA with a more human-readable format, shielding developers from implementation specific details. Moreover, it allows for an encapsulation of business-level goals that will be used as the metrics for the "*correctness*" of the deployment generated. Finally, it will provide adaptability of the deployment's capabilities by scaling nodes that are constrained by metrics.

3

Int2IT Architecture

Contents

This chapter, depicts the entire architecture of the ideal solution (Int2IT), as show in Figure 3.1, starting by providing an overview of the complete system and, afterwards, providing a more detailed description of each component, what functionality it provides and the reason behind its existence.

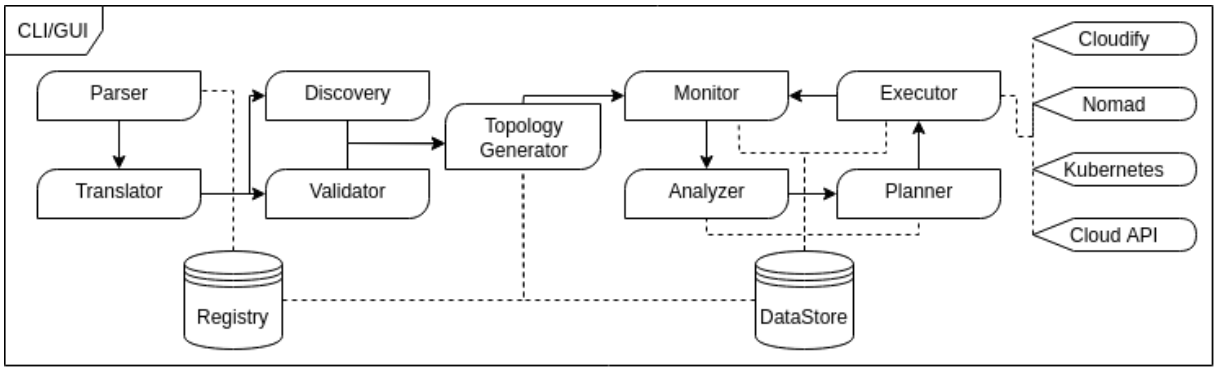

Figure 3.1: Int2IT deployment flow architectural overview.

3.1 Overview

Int2IT is composed of four different layers, *intent* processing, TOSCA processing, autonomic deployment cycle and orchestration, each serving a specific purpose in order to be able to achieve the different goals proposed for this proof-of-concept, as illustrated in Figure 3.2.

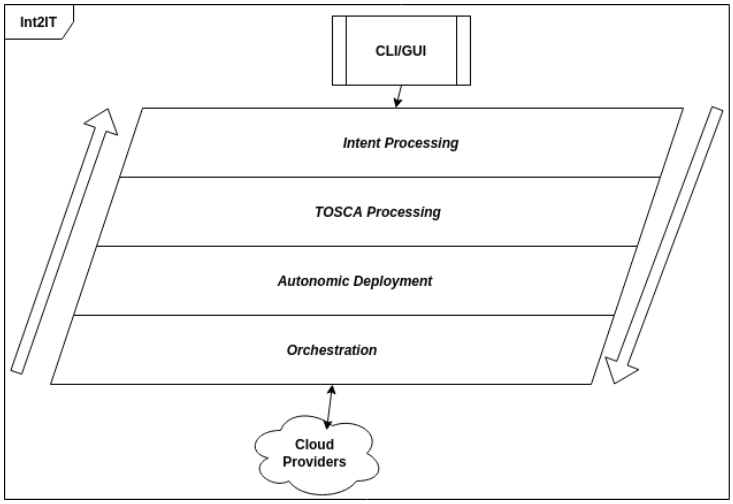

Figure 3.2: Int2IT architectural layers.

Each layer consists of a series of independent components, in other words, each component serves as a contained process which communicates with others in order to perform its functions, but will not be dependent, from a process health perspective, of the others. Consequently, any future implementation

of any of the components will be able to be "plugged in" without compromising the entire system and forcing changes in other components.

3.2 GUI and CLI

To achieve the main goals of this project, all that is required is a simple Graphical User Interface (GUI) capable of capturing the developers' *intents*, displaying the TOSCA topology representation, and displaying the current health status of each registered deployment. Furthermore, Int2IT is also able to be instructed via Command Line Interface (CLI) commands.

3.3 Intent Processing

The primary purpose of this stage is to process the captured *intents* via the GUI or CLI, transforming them into TOSCA definitions and representing them via an intermediary representation that serves as an abstraction from the extensive *intent* description, storing only the relevant information required, while being much more compact and simple than the complete TOSCA code definition.

3.3.1 Parser

The Parser component is in charge of accepting *intent* descriptions provided by the developer written in a more Natural-like language, and parsing those *intents*, into an intermediary representation (which is referred to as an *Order* from now on) where only the relevant infrastructure, policies, and metadata is stored. An *Order* serves as a bridge between the described *intents* and the entire TOSCA YAML infrastructure-as-code definition files, allowing for a more compact and simpler representation of what is necessary for that specific deployment. A simplified example of a possible *intent* and the *Order* representation derived from it, omitting several necessary details for a real deployment, is described as following:

```
1 Register a deployment in GCP with 2 instances: WebServer and DBServer.
2     - WebServer: computes an Apache server hosting a WordPress application;
3     - DBServer: computes a MySQL DBMS hosting a WordPress database.
```

The specified *intent* can now be parsed into an *Order* representation, which will be used later by the system so as to enable a procedural depiction of every step that needs to be taken to be able to fulfill said *intent*. As an example, the aforementioned *intent* can be represented by the following *Order*:

```
Deployment:
    provider: GCP
    components:
        WebServer:
            node: Compute
            hosts:
                ApacheServer:
                    node: WebServer.Apache
                    hosts:
                        WordPressApp:
                            node: WebApplication.WordPress
        DBServer:
            node: Compute
            hosts:
                MySQL:
                    node: DBMS.MySQL
                    hosts:
                        wordpress_db:
                            node: Database.MySQL
```

3.3.2 Translator

The Translator component is responsible for receiving the *Order* previously produced by the Parser and translate it into TOSCA YAML definitions following the recommendations and examples published in the OASIS documentations for TOSCA Simple YAML version 1.3. Following the recommendations (as well as further examples of community blueprint repositories), the Translator will create the deployments' IaC files, depicting all the infrastructure configurations as well as the policies and plugins required for the correct orchestration of the service.

It will be interesting for future works to address the possible implementation of an intelligent Natural Language Processing (NLP) agent as to allow more freedom and fluidity in the *intent* descriptions. Moreover, it will allow for a greater implementation of different services on-the-spot without having to alter the behaviour of component. As per Nefkens [9], the translation process of converting processed intents into procedures composed of repeatable executable steps is one of the most relevant aspects in intent-based designs.

3.4 TOSCA Processing

The primary goal of this stage is to validate the deployment in terms of authentication and any external resources required, such as plugins or tools, based on the TOSCA definition established in the previous phase. Finally, a set of topologies is generated based on the TOSCA definition, but also on the business-level goals associated with the deployment and the level of confidence provided, indicating how strictly Int2IT should adhere to these goals.

3.4.1 Discovery

The Discovery component is in charge of communicating with the various required services to determine their availability, as well as ensuring that the machine hosting Int2IT has the necessary plugins and tools specified in the *Order*. As a result, if the machine is found to be missing any of the required external components, the Discovery will either install them, if possible, or report the need for manual intervention by invalidating the processed deployment. An example could be the following: the Discovery component probes the host machine and determines that the GCP Cloudify plugin is missing. If possible, it attempts to install the plugin to correct this fault.

3.4.2 Validator

The Validator component is in charge of analyzing the TOSCA *Order* definition and ensuring that it contains all of the information required to deploy the specified infrastructure via a “dry-run”, in which the deployment is attempted, but without committing its creation in the cloud (which is already a feature on tools such as Terraform). Furthermore, the Validator should ensure the authentication side by using a tool such as *OAuth2* and, if using a public cloud such as AWS, confirming the existence of a “project” associated with the deployment.

3.4.3 Topology Generator

The Topology Generator component is responsible for generating various infrastructure topologies which meet the demands specified in the TOSCA infrastructure definition and, in addition, selects the one which, from *a priori* knowledge from works such as Sampaio et al. [18] and collected data from previous deployments, most likely results in a near optimal configuration, according to the specified level of confidence as mentioned previously.

For example, considering the deployment depicted in Figure 3.3, one possible topology would be to have three low-level cloud instances where two machines are hosting two *WebServer* application replicas and the third is hosting the *Database*, whereas an alternative could be to have one mid and

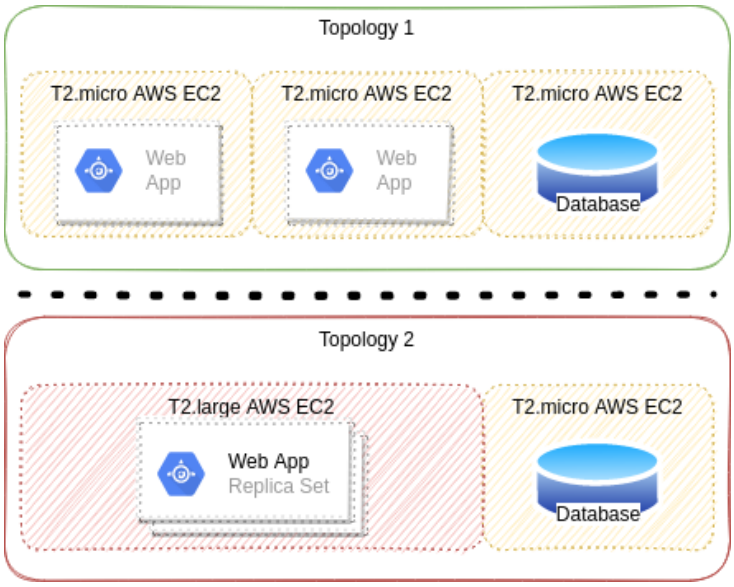

Figure 3.3: Representation of different topologies for the same deployment configuration of two ”Web App” and one database instances

one low-level instances, where the first is hosting all *WebServer* replicas and the latter is hosting the *Database*. Consequently, one topology could be preferable to other given the specific business-level goals captured from the developers *intent* description.

3.4.4 Registry

The Registry database is where the mappings between TOSCA blueprint definitions and original *intent* descriptions are stored. It also saves the relevant *Order* representation, as well as the current topology and health state of each deployment. This is encapsulated by a structure called *records* which is created upon the registration of a new deployment and will be updated according to the changes the deployment incurs either manually or automatically during the autonomic deployment life-cycle.

3.5 Autonomic Deployment

The primary goal of this stage is to apply autonomic computing concepts of **self-*** properties in order to autonomically manage each deployment which is currently online. Each deployment is then monitored, collecting data to be analyzed and, if necessary, applying mitigation actions to the infrastructure topology components to optimize the deployment and ensure compliance with the established high-level goals.

3.5.1 Monitor

The Monitor component is in charge of collecting data about the various infrastructure components used in each deployment in order to determine the current performance and health of said deployment. Traditionally, cloud environments provide native tools to monitor instances and other resources, such as AWS *CloudWatch*, however, in the case of TOSCA-based cloud applications, Sampaio et al. [18] proposes a method for evaluating the topologies of the applications by utilizing *CloudCrawler*. Additionally, other orchestration tools such as *Kubernetes* and *Docker Swarm* also provide the capability of self-monitoring. As a result, the Monitor component must use the most appropriate tool for data collection, based on the needs of the specific deployment, or even a combination of multiple tools.

3.5.2 Analyzer

The Analyzer component is in charge of analyzing and processing the data previously collected by the Monitor in order to determine if there is any breach of the end-goals specific to that deployment. For example, if the deployment requires a CPU utilization of 15-75%, the detection of an instance that is determined to be “overloaded” (or “underloaded”) must result in the execution of mitigation actions that will eventually correct that infraction.

3.5.3 Optimizer

The Optimizer component is analogous to the Plan stage in the typical MAPE-K model for autonomic systems. Its main goal is to plan which mitigation actions should be taken in order to resolve any breaches of end goals that the Analyzer has previously identified. It is important to note that the actions to be taken will have an impact on the system, therefore, they must be applied with some interval of time between them, allowing the system to stabilize and allowing the Monitor and Analyzer to verify if the breaches have been fixed or if additional actions are required.

3.5.4 Resource Manager

The Resource Manager is in charge of keeping the deployed infrastructure in the desired state by acting as an actuator manager, that is, it manages the actuators, which have the ability to perform actions that change the state of the deployment. As a result, just as the Optimizer is analogous to the planner stage, the Resource Manager in the MAPE-K model performs the functions of the executor stage. In particular, the Resource Manager will function as a connector manager, which, as the name implies, manages the available connectors and routes requests to the appropriate connector. To be more specific, a connector is similar to a single executor, whereas the resource manager serves as the central entity in charge of

the collection of executors.

3.5.5 Data Store

The Data Store database is where the monitor component stores the monitored data, which is associated with each deployment currently registered in the system, and where the analyzer component accesses it later. In concrete terms, the Data Store serves as the MAPE-K model's knowledge stage, containing a body of knowledge about the past state and performance of each deployment, as well as the state and performance of each infrastructure element within them. This component can be adequately implemented by a simple database instance with Atomicity, Consistency, Isolation, Durability (ACID) property guarantees for the purposes of this proof-of-concept, without requiring more complex configurations that would allow for better performance and availability. As an example, consider using a containerized replica set of *MongoDB* without the high horizontal scalability provided by "sharding", which allows the creation of subset of key-value pairs such that the database is effectively subdivided into smaller replica sets which are independent on one another.

3.6 Orchestration

The leading purpose of this phase is to connect the internal flow's artifacts produced by the system to the external services that will publish the end results and, essentially, make them available for consumption by the user, in this case the developers. The Orchestration stage will consume the set of instructions provided by the Resource Manager for each deployment and will handle communication with external entities such as cloud providers, e.g., AWS, Azure, or GCP.

In the overall autonomic deployment cycle, each Orchestration connector serves as a simple executor component, whereas the Resource Manager can be thought of as a collection of executors. Those connectors serve as interfaces between Int2IT's internal process flow and the orchestration tools' API. As a result, these connectors are expandable, and add support for a new Orchestration method that is as simple as programming a new connector interface and implementing all of the necessary features to connect the system to the new tool.

3.7 Operations

This subsection provides a detailed description of every required operation, supported by the system, depicting what it consists of and the procedure associated with said operation.

3.7.1 Register

The register operation allows to create a registry mapping (record) of a collection of *intents* described via the GUI or the CLI to an *Order* and, eventually, a TOSCA validated topology viewable via the GUI or as IaC. This *Order* is utilized by the Translator, which in turn generates a TOSCA YAML file for version control, portability, and IaC representation. Following that, the Discovery service will verify the required plugins and endpoints for that deployment, downloading them if they are not present or reporting an error if a specific endpoint cannot be reached. Following the validation of the TOSCA YAML, a dry-run of the deployment is performed to ensure that everything that is required has been obtained in previous steps. If the validation is successful, a TOSCA topology is generated and stored in the Registry, where it is associated with the corresponding *Order*. The steps can be formalized into the following procedure:

- Describe *intents* in GUI / Collect TOSCA as IaC into *Order*;
- Parse *Order* into TOSCA YAML definition;
- Discover resource plugin and endpoints requirements;
- Validate TOSCA YAML and dry-run deployment;
- Generate validated topology and store in registry record.

This procedure could be implemented following the “Register Operation” as illustraded in Algorithm 3.1, showcased in a Python-like manner.

Algorithm 3.1: Register Operation Algorithm

begin

$order : Order \longleftarrow cli.collectIntentFromUser()$
$archive : ArchiveTOSCA \longleftarrow translator.translateOrder(order)$
$if\ not\ (discovery.plugins(order)\ and\ discovery.endpoints(order)) :\ raise\ DiscoveryException()$

$if\ not\ (validator.validate(archive)) :\ raise\ ValidatorException()$
$topology \longleftarrow topologyGenerator(order,\ archive)$
$if\ not\ (resourceManager.dry_run(order,\ archive)) :\ raise\ ResourceManagerException()$
$if\ not\ (registry.store(topology)) :\ raise\ RegistryException()$
$return$

3.7.2 Deploy

The deploy operation allows the provisioning of a previously registered and validated cloud deployment or to register a new deployment on the fly and immediately deploy it to the specified cloud environ-

ment. In the event of an in-place registration, the deploy operation should be interrupted and a registry operation should be performed in accordance with the logical flow described in the subsection above. Following completion, the deployment should resume, skipping the initial step of determining the deployment's record in the registry, which is only required if the deployment is already registered in the system. The system should then deploy the topology, previously generated, via the Resource Manager which manages the available connectors and checks to see if they are still operational and ready to use. Finally, once the infrastructure has been verifiably deployed to the cloud, Int2IT should initiate the autonomic MAPE-K based life-cycle on said deployment, gathering data about the deployment's health and performance on a regular basis, saved in Data Store, and performing mitigation actions as needed. The steps can be formalized into the following procedure:

- Ascertain record of intended resources in registry / Perform in-place registration;
- Deploy topology through Resource Manager and Connectors;
- Start autonomic MAPE-K based life-cycle on deployed infrastructure;
- Collect data and store it in DataStore;

This procedure could be implemented following the "Deploy Operation" as illustrated in Algorithm 3.2, showcased in a Python-like manner.

Algorithm 3.2: Deploy Operation Algorithm

begin

$record_id \longleftarrow cli.collectRecordIDFromUser()$

$if\ not\ (registry.recordExists(record_id) :\ record\ \longleftarrow cli.inPlaceRegistration()$
$else :\ record \longleftarrow registry.getRecordFromID(record_id)$

$deployment \longleftarrow resourceManager.deploy(record)$
$resourceManager.startMAPEK(deployment)$

$return$

3.7.3 Destroy

The destroy operation leads to the complete removal of a specific deployment from the system regardless of the current state, whether it is currently online or offline. After determining the record of the specified deployment in the Registry, the Resource Manager will shutdown all the components which are currently online in a live state. Afterwards, the destroy operation will commence with the Resource Manager dispatching commands through the connectors in order to delete all resources in the cloud and remove them. Lastly, if the destroy command is issued along with the registry removal flag set to true,

then Int2IT will delete from Registry the record and files corresponding to the deployment. The steps can be formalized into the following procedure:

- Ascertain record of intended resources in Registry;
- Locate all resources described in *intent*;
- If deployment is online then Shutdown;
- Terminate and remove all resources;
- If option to remove deployment then remove from Registry.

This procedure could be implemented following the "Destroy Operation" as illustrated in Algorithm 3.3, showcased in a Python-like manner.

Algorithm 3.3: Destroy Operation Algorithm

begin

$record_id \longleftarrow cli.collectRecordIDFromUser()$

$if\ not\ (registry.recordExists(record_id):\ raiseDestroyException()$
$record \longleftarrow registry.getRecordFromID(record_id)$

$if\ resourceManager.isDeploymentOnline(record):\ resourceManager.shutdown(record.id)$

$resourceManager.destroy(record)$

$registry.destroy(record)$

$return$

3.7.4 Update

The update operation can be invoked either manually or autonomically, if any non-compliance is detected, and can be subdivided into two different categories: server configuration and system policy updates. The former consists of server-level changes such as increasing or decreasing the number of instances, replicas, or even hardware configurations. The latter, as the name implies, entails changing/removing existing policies or enacting new ones, such as raising the maximum CPU utilization threshold from 80% to 90%. The procedure steps remain the same for either category of update. To begin, and similarly to the deploy operation, the record of the intended resources in the Registry must be determined. If the deployment is currently offline, then the update operation performs changes merely in the Registry and Data Store databases. Otherwise, the Resource Manager will, through its connectors, perform the necessary changes on the cloud infrastructure after verifying the Registry and locating all the resources affected by the change. It is important to note that, whenever possible in live deployments,

a rolling upgrade should be performed to avoid major service downtime. The steps can be formalized into the following procedure:

- Ascertain record of intended resources in Registry;
- Locate all resources that require changes;
- If needed and possible perform rolling upgrade on live deployment;
- Perform changes in registry.

This procedure could be implemented following the "Update Operation" as illustrated in Algorithm 3.4, showcased in a Python-like manner.

Algorithm 3.4: Update Operation Algorithm

begin

$record_id \longleftarrow cli.collectRecordIDFromUser()$
$update \longleftarrow cli.collectUpdateFromUser()$

$if\ not\ (registry.recordExists(record_id) :\ raiseUpdateException()$
$record \longleftarrow registry.getRecordFromID(record_id)$

$if\ resourceManager.canRollingUpgrade(record) :\ resourceManager.rollingUpgrade(record, update)$

$registry.update(record, update)$

$return$

3.7.5 Shutdown

The shutdown operation works as the inverse procedure of the deploy operation, in other words, it terminates all infrastructure components of a specified deployment which is currently in a live state, since shutting down non-deployed resources is not possible. Similarly to the update operation, the shutdown can be performed manually or automatically, though the latter is only possible in the case of a non-repairable deployment, such as when cloud instances in a deployment keep "crashing" due to an unknown error, and human intervention is required to determine the source of the error and take action to correct it. To begin, and similarly to other operations, the record of the intended resources in the Registry must be determined as well as guaranteeing that they are in a deployed state. After locating all components, the Resource Manager will be responsible for shutting down each one by utilizing the connectors available and necessary to "bring down" the entirety of the deployment. The steps can be formalized into the following procedure:

- Ascertain record of intended resources in Registry;

- Locate all resources that require changes;
- Shutdown every deployed resource;
- Perform changes in Registry.

This procedure could be implemented following the “Shutdown Operation” as illustrated in Algorithm 3.5, showcased in a Python-like manner.

Algorithm 3.5: Shutdown Operation Algorithm

begin

$record_id \longleftarrow cli.collectRecordIDFromUser()$

$if\ not\ (registry.recordExists(record_id) :\ raiseShutdownException()$
$record \longleftarrow registry.getRecordFromID(record_id)$

$if\ resourceManager.isDeploymentOnline(record) :\ resourceManager.shutdown(record)$

$registry.shutdown(record)$

$return$

3.7.6 Rollback

The rollback feature is an automatic operation that occurs whenever any other operation fails or is unable to complete the expected workflow, leaving traces of unfinished changes in Int2IT, for example, a multi-component update to a deployment that is able to update components A and B (having the changes registered in the Registry database), but is unable to complete update to component C for some unknown or known reason. In this case, a rollback operation is initiated, which reverts all changes made during the failed update operation, restoring Int2IT (and the deployment) to its previous healthy state.

4

Implementation

Contents

In this chapter, the architecture of the realized proof-of-concept is also presented (Int2IT-Lite), reflecting the intricacies of the obstacles faced, while showing the capabilities of the proof-of-concept which should motivate further work on a fully realized and more advanced framework.

4.1 Int2IT-Lite: Proof-of-Concept

In this section, an overview of the architecture of the proof-of-concept Int2IT-Lite solution is presented. The solution represents the bare-bones requirements for a possible implementation of a framework that materializes the concept of the Int2IT system.

4.1.1 Configuration and Resources

Int2IT-Lite was developed entirely with Python 3.8.6, utilizing some key packages, namely: (a) cloudify-rest-client/ cloudify-common [1] [2]; (b) pymongo [3]; (c) pandas [4]; (d) pydantic [5]; (e) pyaml/oyaml [6] [7]. In addition, some containerized applications were utilized for the correct functioning of the prototype, namely: (a) Cloudify-Community:6.4 [8] and (b) MongoDB:4.2.21-rc0 [9].

MongoDB [10] was chosen as the implementation of the Registry and DataStore components for three main reasons: (a) the python library for MongoDB has been used quite in many tools as it allows for easy access to the databases main capabilities of operating over documents in a schema-less fashion as key-value pair collections; (b) the operations over the data model records did not demand complex relational operations, consisting mostly of retrieving documents by key; and (c) for the case of the "store" collection of metrics, on which some query-like filtering is done, sharding could be later applied for each deployment [11].

The Cloudify Community manager was selected as it is the only one open to the general public and, therefore, would make Int2IT-Lite available to anyone. Despite starting to develop Int2IT in version 6.3, the prototype was adapted so as to be able to upgrade to the latest version (6.4) which brought forth various quality-of-life improvements. However, an acknowledgement has to be made to Cloudify's Slack community which were able to help in various forms, including providing a solution to the docker image of Cloudify Community Manager, which got a broken release update midway through Int2IT-Lite's

[1] https://github.com/cloudify-cosmo/cloudify-rest-client
[2] https://github.com/cloudify-cosmo/cloudify-common
[3] https://pypi.org/project/pymongo/
[4] https://pandas.pydata.org/
[5] https://pydantic-docs.helpmanual.io/
[6] https://pyyaml.org/wiki/PyYAML
[7] https://pypi.org/project/oyaml/
[8] https://hub.docker.com/r/cloudifyplatform/community-cloudify-manager-aio
[9] https://hub.docker.com/_/mongo
[10] https://www.mongodb.com/
[11] https://www.mongodb.com/docs/manual/sharding/

development.

Lastly, the prototype includes pre-configured resources for the creation and provisioning of the nodes generated for the use case example. The resources are based on publicly available example blueprints [12], installation websites as well as fixes to known errors and problems. Such is the case for the *CentOS7* GCP image which, as part of the known problems guide [13], cannot use "*yum install commands*", due to a configuration typo. This is addressed by having a "hotfix" TOSCA node prior to provisioning the GCP instance with the required applications.

4.1.2 Int2IT-Lite Architectural Design

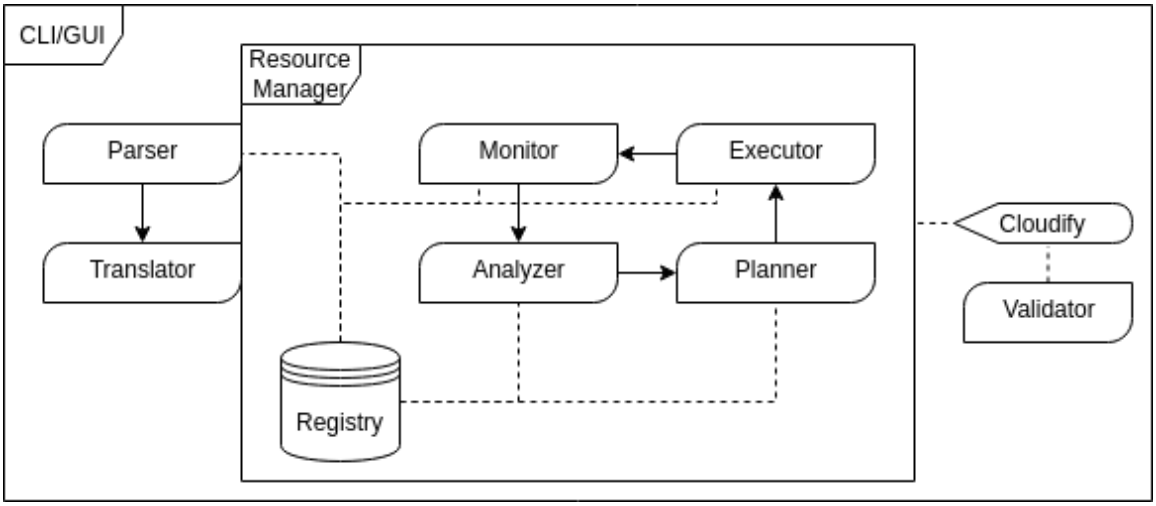

Figure 4.1: Int2IT-Lite Architecture

As it is possible to observe in Figure 4.1, the architecture of the prototype closely follows the the ideal Int2IT solution (as presented in Chapter 3). However, some components have been simplified or have been integrated together with others.

The interactive CLI component has been configured to be capable of handling the five required operations, from Section 4.2, for the correct functioning of the prototype namely: (a) Deploy; (b) Destroy; (c) Register; (d) Shutdown; and (e) Update; In addition, it is complemented by additional operations that facilitate the usage of the tool as well as other operations that act over elements in the data model, from Section 4.1.3. Therefore, the developers utilizing Int2IT-Lite are able to have finer control over the handling of their deployment and *Order* namely: (a) Backup; (b) List; (c) Help; (d) Reset.

The database components, Registry and DataStore, where merged into a single MongoDB component. The data is separated into three collections in the same database named "Int2IT": (a) *orders*; (b) *deployments*; and (c) *store*;

[12] https://github.com/cloudify-community/blueprint-examples
[13] https://cloud.google.com/compute/docs/troubleshooting/known-issues

The first collection records documents regarding the various *Orders* registered in the system, having all the necessary information to be able to create a deployment from it. The second collects documents of deployments created during the process of deploying an *Order* for the very first time. It keeps track of the most relevant information necessary for the correct functioning of the system during the autonomic management life cycle. Lastly, the latter stores information collected by the Monitor component for all deployments, which have an *UP* deployment status, so that the Analyzer may afterwards process.

The *Topology Generator* and the *Discovery* components have been omitted since, in case of the former, it has been simplified and incorporated into the *Translator*, which generated the same topology for all intents and deployments. This was chosen so for the same reason that the *Translator* was implemented as a table of translations for the Cloudify connector. The complexity and capabilities of these components could be subject of entirely separate research works, for example, other master's thesis dissertations, therefore a more simplified approach was chosen so as to meet the main requirements for the use case selected, which will be presented in the next chapter.

For the case of the *Discovery* component, given that Cloudify was the only major third-party software utilized, it was not developed for this prototype and all additional resources were assumed to have been ensured any developers using Int2IT-Lite.

Lastly, the *Validator* component has been connected to the Cloudify connector due to the fact that the Cloudify Manager already includes and enforces TOSCA archive and blueprint validation, based on Cloudify's Domain Specific Language (DSL), therefore making the *validator* component redundant.

4.1.3 Data Model

In Int2IT, the data model, as illustrated in Figure 4.2, is realized with a combination of dataclass capabilities and key-value dictionaries, both native to Python's. Every model is able to be converted from dataclass representation to a dictionary record and vice-versa.

Order: An order is composed of two models: Metadata and Services. The first holds information regarding the properties of the *intent* whereas the latter contains contents of the *intent* regarding the desired deployment.

Metadata: The metadata contains the information uniquely identifying the *intent*, as well as the desired orchestration methodology and user identity.

Services: The services consists of a key-value pair dictionary that associates a user-defined name to a specific component.

Component: A component represents a node application which can be either a Docker container or a Database (either PostgreSQL or Redis). Additionally, contains a set of Metrics which specify how to track the performance of the component.

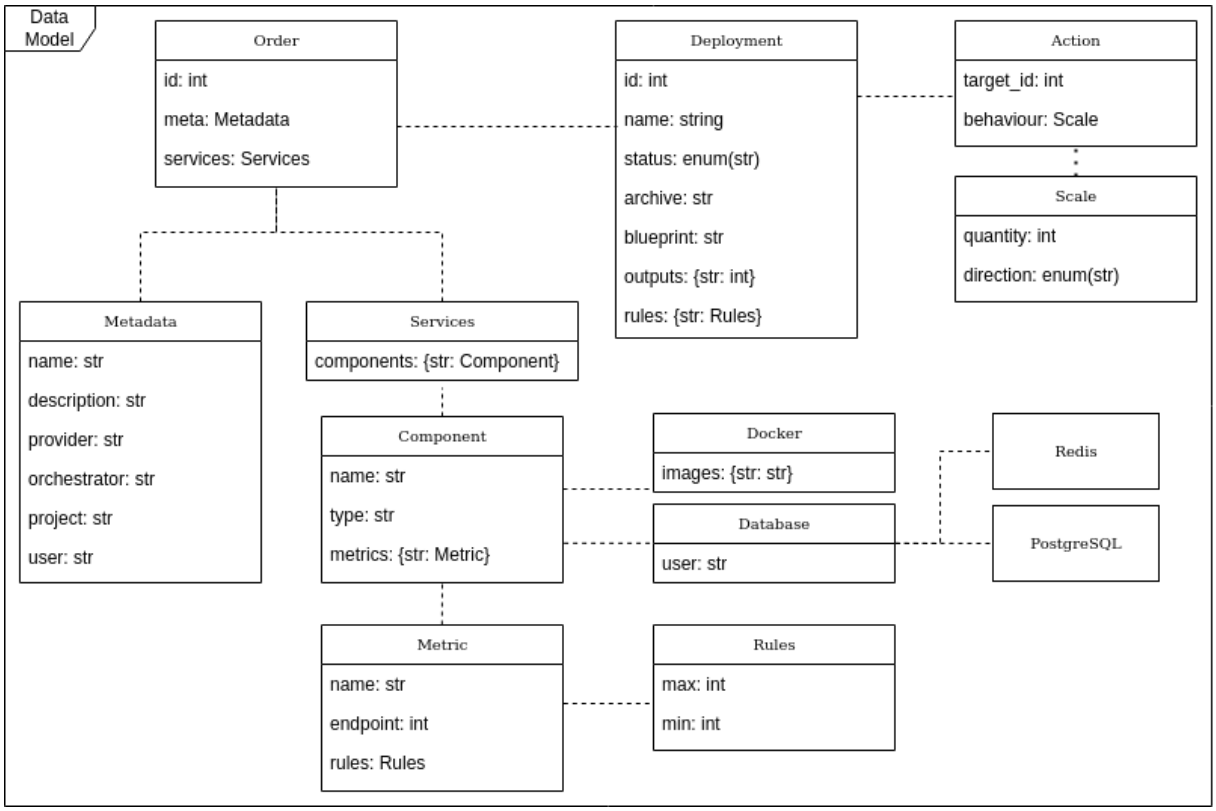

Figure 4.2: Data Model

Metric: A metric specifies an endpoint, developed by the owner of the order, from which Int2IT-Lite is able to retrieve values as part of the monitoring deployment process.

Rules: Rules specify optional maximum and minimum values associated to a metric.

Deployment: A deployment is uniquely identified by an identifier inherited from the order that originated said deployment. It has a status with a fixed set of possible values (UP, DOWN, UNDEPLOYED, TERMINATING). It stores the path to the deployment archive, which consists of a compressed directory with every file needed to deploy to Cloudify, as well as the main blueprint name. Lastly, it holds a set of outputs, consisting of key-value pairs of names of metrics and their respective endpoints. In addition, another set exists specifying key-value pairs of metric names and their rules.

Action: An action is targeted to a deployment through its deployment id. It also contains a behaviour of scaling.

Scale: Contains a quantity by which to scale a certain deployment either in a horizontal or vertical direction.

4.2 Required Operations

This section provides a description of the implementation of the required operations as well as a short comparison with the full fledged operations described in Chapter 3.

4.2.1 Register

The register operation in Int2IT-Lite starts by collecting from the CLI an *intent*, described in a YAML file. Next, the *parser* is called to firstly, using the *pyaml* Python library, parse the YAML file into an unprocessed *intent* i.e a key-value pair dictionary.

If the parsing is successful, then the *Parser* is once again called to parse the unprocessed *intent* into the *Order* representation. This process is done by recursively passing the data within the unprocessed *intent* into the construction of the various objects in the data model. For example, the *Parser* would try and create an *Order* object from the data available, which would then try to create *Metadata* and *Services* objects, passing into them the data relative to those components. After an *Order* representation is achieved, it is then passed into the *Registry*, which stores it into the "orders" collection in the database.

This procedure could be implemented following the "Register Operation" as illustrated in Algorithm 4.1, showcased in a Python-like manner.

Algorithm 4.1: Int2IT-Lite Register Operation Algorithm

begin

$location \longleftarrow cli.getLocation()$
$filename \longleftarrow cli.getFilename()$

$if\ not\ fileExists(filename, location):$
$\ raise\ FileNotFoundException()$
$file \longleftarrow getFile(filename,\ location)$

$unprocessed_intent:\ dict \longleftarrow parser.processYamlIntoIntent(file)$
$order:\ Order \longleftarrow parser.processIntentIntoOrder(unprocessed_intent)$

$if\ not\ registry.registerOrder(order):$
$\ raise\ RegisterOrderException()$
$else: print('Orderwassuccessfullyregisterwithidorder.id')$

$return$

4.2.2 Deploy

The deploy operation in Int2IT-Lite starts by finding the *Order* associated with a user given identifier and, if possible, register and create the deployment for the *Order*. To create the deployment it is needed to create an archive. This is done by creating a directory for the *Order* where all the necessary scripts, from the pre-installed Int2IT-Lite scripts, as well as the TOSCA blueprint will be gathered into. The blueprint

is generated by the Translator, prior to creating the archive, by iterating through each component in the *services* of the *order* and mapping it to a Cloudify TOSCA definition. An example of this can be found in Figure 4.3 where a Cloudify volume can be obtained through the following translation.

```python
def volume_tosca(
    cloud_config: dict,
    volume_name: str = "disk",
    image: str = DEFAULT_CLOUD_IMAGE,
    orchestrator: str = "cloudify",
    cloud: str = "gcp",
) -> dict:
    return {
        volume_name: {
            "properties": {
                "boot": True,
                f"{cloud}_config": cloud_config,
                "image": image,
                "size": 20,
            },
            "type": f"{orchestrator}.{cloud}.nodes.Volume",
        }
    }
```

Figure 4.3: Translation function for a volume in the Translator component

After having generated the archive, the deployment is registered into the *Registry*, which stores it in the "deployments" collection in the database. Having completed the registration of the deployment, the *Resource Manager* is utilized to deploy the deployment into the desired cloud with the appropriate connector. In the case of Int2IT-Lite, the only supported CP and *Orchestrator* are GCP and Cloudify, respectively. The *Resource Manager* then uploads, registers and initiates an "install" execution, on the deployment, through the Cloudify connector. If the deployment had been previously deployed and is in a "down" state, then a "start" executions is started instead. Lastly, it awaits the conclusion of the "install" (or "start") execution on the Cloudify connector and, if successful, updates the deployments status to a live deployment through the *Registry*.

This procedure could be implemented following the "Deploy Operation" as illustrated in Algorithm 4.2, showcased in a Python-like manner.

Algorithm 4.2: Int2IT-Lite Deploy Operation Algorithm

begin

$orderID \longleftarrow cli.getOrderID()$
$order \longleftarrow registry.getOrder(orderID)$

$if\ not\ translator.canBlueprint(order):$
$\quad raise\ TranslationBlueprintException()$

$blueprint \longleftarrow translator.generateBlueprint(order)$
$if\ not\ translator.generateArchive(order,\ blueprint):$
$\quad raise\ TranslationArchiveException()$

$if\ not\ registry.registerDeployment(order,\ archive,\ blueprint):$
$\quad raise\ RegistryDeploymentException()$
$deployment \longleftarrow registry.getDeployment(order.id)$

$if\ not\ resourceManager.uploadDeployment(deployment):$
$\quad raise\ ResourceManagerUploadException()$

$if\ not\ resourceManager.startDeployment(deployment):$
$\quad raise\ ResourceManagerCompletionException()$

$if\ not\ resourceManager.waitCompletion(deployment):$
$\quad raise\ ResourceManagerInstallException()$

$if\ not\ registry.updateDeployment(deployment):$
$\quad raise\ RegistryUpdateException()$

$return$

4.2.3 Update

The update operation in Int2IT-Lite starts by finding the *Order* associated with a user given identifier and, if possible, update the *Order* and respective deployment if in live state. From the CLI, the user provides both the identifier of the *Order* they want to update and the collection of values, as key-value pairs, that have to be updated. Since it could be possible that multiple components have identical keys, for some attributes at least, a process of dictionary flattening was utilized so as o uniquely identify each attribute. For example, to update the maximum value of the rules of a metric “MetricName”, then the format of the selector in the CLI should be $\{"Services.ComponentName.MetricName.Rules.Max" : newValue\}$. Lastly, the gathered selector is utilized for a “deep update” with the *Order*, in a dictionary format.

This procedure could be implemented following the “Update Operation” as illustrated in Algorithm 4.3, showcased in a Python-like manner.

Its is important to note that only “soft” updates are immediately reflected in the *order* whereas “hard” updates will only be applied when possible, which could be only performed after terminating current live deployments. An example of a “hard” update could be changing the type of a database component in the deployment from Redis to PostgreSQL, which would require a remaking of the *Order*’s deployment archive.

Algorithm 4.3: Int2IT-Lite Update Operation Algorithm

begin

$orderID \longleftarrow cli.getOrderID()$
$selector \longleftarrow cli.getUpdateSelector()$
$record: \ dict \longleftarrow registry.getOrderAsRecord(orderID)$

$if \ not \ selectorIsValid(order, \ selector):$
$\quad raise \ UpdateSelectorException()$

$updated_record \longleftarrow updateDeep(record, \ selector)$
$if \ not \ registry.updateRecord(updated_record):$
$\quad raiseRegistryUpdateRecordException()$

$return$

4.2.4 Shutdown

The shutdown operation in Int2IT-Lite starts by finding the *Order* associated with a user given identifier and, if possible, shutdown all resources in the cloud provider. From the CLI, the user provides the identifier for the deployment that they want to shutdown. If the deployment is in a live state, then the shutdown can start by utilizing the *Resource Manager* to commence a "stop" execution through the Cloudify connector. If the shutdown operation was started through a destroy operation then a "uninstall" executions is started instead. Lastly, the deployments is updated with a "down" status after the execution finishes.

This procedure could be implemented following the "Shutdown Operation" as illustrated in Algorithm 4.4, showcased in a Python-like manner.

Algorithm 4.4: Int2IT-Lite Shutdown Operation Algorithm

begin

$orderID \longleftarrow cli.getOrderID()$
$order: \ Order \longleftarrow registry.getOrder(orderID)$
$is_destroy \longleftarrow cli.isFromDestroy()$

$if \ not \ registry.deploymentIsLive(order):$
$\quad raise \ RegistryShutdownException()$

$if is_destroy:$
$\quad resourceManager.terminateDeployment(order)$

$else: \ resourceManager.shutdownDeployment(order)$

$if \ not \ resourceManager.waitForCompletion():$
$\quad raise \ ResourceManagerShutdownException()$

$return$

4.2.5 Destroy

The destroy operation in Int2IT-Lite starts by finding the *Order* associated with a user given identifier and, if possible, shutdown and eliminate all resources in the cloud provider, as well as from the *Registry*. From the CLI, the user provides the identifier for the *order* and deployment that they want to destroy. The shutdown operation is then initiated if there is a current live deployment with the provided identifier. When the shutdown is complete and no components are still deployed in the cloud, a process of resource deletion begins. This deletion effectively deletes all records associated with the order and deployment from the *Registry*'s database collections.

This procedure could be implemented following the "Destroy Operation" as illustrated in Algorithm 4.5, showcased in a Python-like manner.

Algorithm 4.5: Int2IT-Lite Destroy Operation Algorithm

begin

$orderID \longleftarrow cli.getOrderID()$
$order : Order \longleftarrow registry.getOrder(orderID)$

$if\ registry.deploymentIsLive(order) :$
$\quad resourceManager.shutdown(order.id)$

$if\ not\ registry.deleteOrder(order) :$
$\quad raiseRegistryDeleteRecordException()$

$return$

4.3 Autonomic Life-Cycle

In this section an overview will be given over the implementation of the MAPE-K model as part of the autonomic life-cycle, whose main purpose is guaranteeing the compliance of a deployments state and performance with the *Order*'s business-level metrics.

In Int2IT-Lite, the autonomic life-cycle is implemented through "Thread" processes. At the start of the program, the *Resource Manager* component, after its creation, initiates a thread on a function that cycles through the different stages of the MAPE-K model every thirty seconds. As mentioned previously, the knowledge component was simplified to two collections in the *Registry*'s database, respectively "deployments" and "store".

Each stage is implemented as Python modules that implement two functions: A main general function that iterates through all the deployments available and eligible to be autonomicaly managed i.e deployments whose status are live; and an additional one that applies the stages' purpose to a single deployment. In the following subsections, a depiction of each stages' more specific function will be provided, since the general one is mostly identical to the others in different stages.

4.3.1 Monitor

The implementation of the *Monitor* component was achieved by iterating through each deployment and acquiring their metrics' endpoint values, in other words, creating a web Representational State Transfer (REST) request to their endpoints and retrieving the values for the respective metrics.

For each metric value, a data point record is created in a format of a key-value pair dictionary, containing the following attributes: (a) ***id***: randomly generated unique id according to RFC 4122 [14] [15]; (b) ***deploymentID***: the deployments' id; (c) ***metricName***: name of the data points' metric; (d) ***value***: value of the data point; (e) ***timestamp***: a timestamp of the current time in $YY-MM-DD\ hh:mm:ss$ format.

Lastly, the data points collected are stored in the "store" collection of the knowledge component, which is incorporated into the *Registry*.

4.3.2 Analyzer

The implementation of the *Analyzer* component was achieved by iterating through each deployment and retrieving the data points, for all of the deployments' metrics, that have been collected in the last five minutes.

For the metrics that have more than five data points their rules are verified to be in compliance, where the others are skipped so as to allow for the collection of more data points and having a more stable and precise set.

Compliance is verified, for each rule of every metric, if the set of data points is at least 50% compliant with the rule. Concretely, this means that if in the last five minutes more than 50% of the data points collected by the *Monitor* do not comply with the rule, then an action needs to be taken.

After concluding that a specific deployment needs to be acted upon, the *Analyzer* will inform the planner that a target deployment is either under or over the desired values.

4.3.3 Planner

In order to scale a deployment there are two options: either scale horizontally or scale vertically. The former consists of modifying the number of computational resources, also called "scaling out/in", while the latter consists of changing the configuration of the resource itself, also called "scaling up/down".

An example of the former would be to increase the number of virtual machines by one instance, whereas the latter could be upgrading the type of virtual machine to a more computationally powerfull one.

[14] https://docs.python.org/3.8/library/uuid.html#uuid.uuid4
[15] https://www.rfc-editor.org/rfc/rfc4122.html

For this prototype, only scaling in or scaling out actions are available, because they are directly supported by Cloudify and since scaling up or down actions would require changing the blueprint of the deployments archive.

Since the only available rules are maximum and minimum values, therefore if a data point set is determined to be above the max, then a scale-in action is need. In the same manner, if the set is determined to be below the minimum permitted values, then a scale-out action is required.

4.3.4 Executor

The implementation of the *Executor*, for the Int2IT-Lite prototype, was achieved by starting “scale” executions through the Cloudify connector. The *Executor* iterates through the collection of actions received, determined by the planner component, and applies the respective scale execution.

For example, since an action consists of a target deployment identifier and a scaling behaviour (horizontal either in or out), then a ”scale” execution is started with a delta node change of either 1 or -1, depending on the direction of the behaviour. The executor then awaits the completion of each scale execution and, afterwards, the autonomic life-cycle is restarted.

5

Methodology

Contents

In this chapter, an overview is given over the use case utilized for demonstration purposes of the capabilities of the developed prototype. First, a detailed description of the use case is provided as well as a justification as to the reason for having chosen it. Afterwards, several sections will showcase the various stages through which the application be submitted to, starting from the showcase of the *intent* which correctly describes the app. Then, the TOSCA model derived from the *Order* assigned to the *intent* is analyzed and, lastly, concluding with an overview of the deployments adaptability when confronted with a monitoring process which indicates an under performance and necessity to execute scaling actions.

5.1 Docker Voting-App Use Case

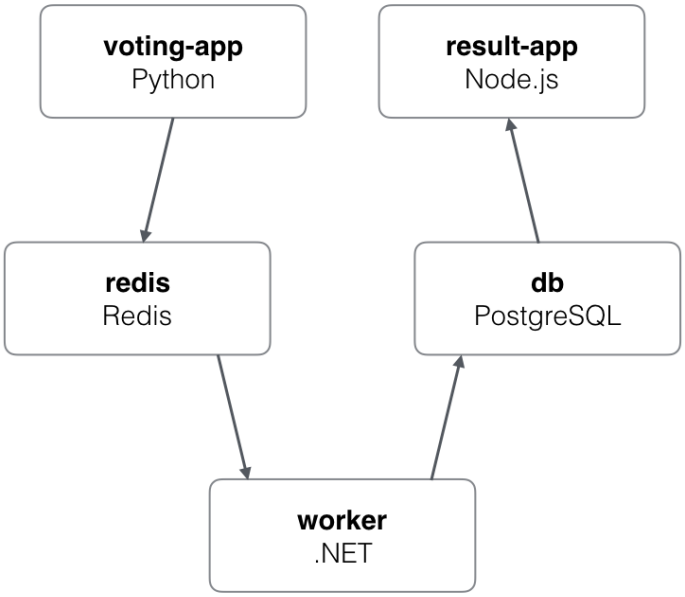

Figure 5.1: Docker Voting-App Architecture Overview

As stated in the original GitHub's repository [1], the use case represents "A simple distributed application running across multiple Docker containers.". More complex examples were considered, especially ones that emulate large applications through a microservices perspective.

For example, one could have considered the "Sockshop" microservices [2] architecture which several databases, caching and various frontend and backend applications. However, since the focus of Int2IT

[1] https://github.com/dockersamples/example-voting-app
[2] https://github.com/microservices-demo/microservices-demo

and the prototype Int2IT-Lite lies upon the infrastructure layer, since they are infrastructure management platforms, instead of the application layer, then it seemed to be excessive to deploy a system of that scale. Especially, since many of the nodes would end up serving the same role of just storing information in some form of database or functioning as an application node where developed software would run on.

Therefore, it seemed more practical and justifiable to choose the Docker voting app example for the purpose of a demonstration of the prototype's capabilities.

The Docker voting app example is composed of five services: (a) "*voting-app*" a frontend app where users vote between cats and dogs; (b) "*result-app*" a web app where the current results of the votes; (c) "*worker*" a backend vote consumer; (d) "*db*" a permanent storage database; and (e) "*redis*" as a collection queue for the votes;

5.2 Test-Bed Environment

Regarding the environment test-bed required for the creation, development and evaluation of the proof-of-concept proposed, a simple local machine or a private cluster would be perfectly suitable, since the success criteria of the prototype Int2IT-Lite depends on three principal conditions:

- The prototype is able to create a TOSCA-compliant archive which can be successfully deployed in a TOSCA-compliant orchestrator to the cloud provider specified;
- The components orchestrated are provisioned according to the *intent* that described at an application level, in other words, the software correctly installed and functioning;
- The deployment can be monitored through the specified endpoint and, from values obtained through said endpoint, scaling actions can be derived and applied, thus changing the configuration of the deployment in a predictable manner.

Given that the criteria aim to evaluate the efficacy of the solution, not the performance or efficiency of said solution, then it was justifiably sufficient to have a personal laptop as the test-bed environment for the prototype. As for the application nodes themselves, they were installed on a default instance based on the "virtual machines" archive supplied by the Cloudify team. [3] This archive is able to be deployed to various cloud providers and utilizing different orchestration technologies and it deploys a basic virtual machine with a disk all within a firewalled network.

Lastly, for the monitoring part of the deployment a Flask [4] Python server was created and configured to provide an endpoint on *localhost:5000*, where the *Monitor* component of the prototype will be able to retrieve values for the "Throughput". Specifically, a value of "90" was hard-coded into the endpoint.

[3] https://github.com/cloudify-community/blueprint-examples/releases/download/6.3.0-10/virtual-machine.zip
[4] https://flask.palletsprojects.com/en/2.2.x/

This was chosen so as to mock an under-performing deployment which, if the prototype is successfully achieving the third criteria, then it will trigger a scaling action.

5.3 Voting-App Intent

```
Metadata:
    Name: "Voting-App"
    Description: "Voting app in GCP"
    Provider: "GCP"
    Orchestrator: "Cloudify"
    User: "Admin-GCP"
    Project: "VotingAppGCP"
Services:
    app:
        Type: "Docker"
        Images:
            voting-app: "docker/example-voting-app-vote:latest"
            result-app: "docker/example-voting-app-result:latest"
            worker: "docker/example-voting-app-worker:latest"
        Metrics:
            Throughput:
                Endpoint: 5000
                Rules:
                    Min: 100
    redis:
        Type: "Redis"
        User: "Admin"
    db:
        Type: "PSQL"
        User: "Admin"
```

Figure 5.2: Voting-App Intent

In this section, it is presented a possible implementation of a GCP deployment of the Docker Voting app use case through the use of an *intent*, described in a declarative YAML format, utilizing the Cloudify connector, as can be observed in Figure 5.2 present in a YAML file named "*webvote*". Naturally, the values are not representative and are used only for demonstration purposes. It is also assumed that the author behind this *intent* has configured everything correctly when it comes to the values and any endpoint connections specified.

In this *intent*, it is can be verified that the metadata regarding the deployment concretely specifies not only the cloud provider to be used, but the specific orchestrator as well. Next, the services block describes three main components of the deployment: (a) an "app" component of type "Docker", which will run containers using the images provided, that has business metric to comply with. The "Throughput"

metric specifies to the system that, in order to be able to monitor the component correctly, it has to connect to the endpoint at port 5000. Finally, the *intent* indicates that this *Throughput* has a minimum value acceptable of 100; (b) a “redis” component of type “Redis” with a general user. This node does not have associated metrics, therefore will not be considered in the *Monitor* process; and, lastly, (c) a “db” component of type “PSQL” configured in an similar manner to the *redis* node described before.

In Figure 5.3, it is demonstrated the TOSCA topology that successfully performs the deployment, following the *Services* mentioned previously, as generated by Cloudify. As mentioned previously, the translation table includes a “hotfix” node for the deployment, since the “*CentOS*” image in GCP has a known issue that makes it incapable of executing “*yum commands*”. Beyond this “hotfix” the TOSCA model describes a *vm* instance on GCP, storage accessible with an “*agent-key*”, with a *disk*. Additionally, one application server node, which represents the Docker software, and two database nodes, representing PostgreSQL and Redis. All utilize Fabric, a python library for “ streamlining the use of SSH for application deployment or systems administration tasks”, so as to run the commands to install each software component inside the *vm* instance. [5]

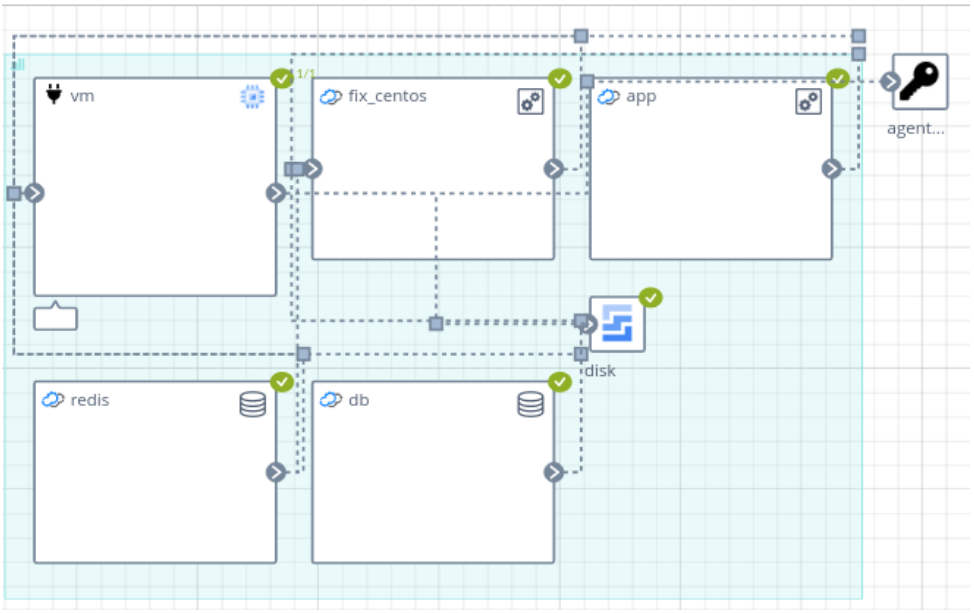

Figure 5.3: Voting-App TOSCA Model

5.4 Deployment Results

In this final section, a step-by-step analysis of the outcomes of the processing and upholstering of the use case *intent* will be performed. Firstly, in Figure 5.4, the *intent* was provided to Int2IT-Lite through

[5] https://advanced-python.readthedocs.io/en/latest/fabric.html

the CLI by attempting a *register* operation on the *webvote.yaml* file.

```
register webvote.yaml default
Successfully registered order with id 1
list orders
----------------------------ORDERS---------------------------------
```

Figure 5.4: Voting-App registered successfully

This operation completed successfully, as indicated by the system, meaning that an *Order* representation of the intent was recorded and stored in the registry with *id=1*. Following, a *list* command was able to show, in a human readable manner in Figure 5.5, the contents of the *Order* where it is possible to verify that the contents of the *intent* are accurately represented.

```
--  Order:
     id: 1
     Metadata: Metadata:
     Name: Voting-App
     Description: Voting app in GCP
     Provider: GCP
     Orchestrator: Cloudify
     User: Admin-GCP
     Project: VotingAppGCP

     Services: Services:
Docker:
     Name:
     Type: Docker
Metrics:
     Name: Throughput
     Endpoint: 5000
     Rules: Rules:
     Max: None
     Min: 100
voting-app:docker/example-voting-app-vote:latest
result-app:docker/example-voting-app-result:latest
worker:docker/example-voting-app-worker:latest
Database:
     Name:
     Type: Redis
 User: Admin
Database:
     Name:
     Type: PSQL
 User: Admin  --
-------------------------------------------------------------
```

Figure 5.5: Voting-App Order registered successfully listing

Secondly, a *deploy* operation was performed on the registered *Order* so as to, effectively, deploy the application to GCP, utilizing the Cloudify orchestrator, after a translation process has been applied. This is necessary in order to generate the appropriate TOSCA-based archive, which includes a main TOSCA blueprint that will be used to create the components in the CP. After the generation, the archive was

then uploaded, had its blueprint registered and had a deployment created out of it in Cloudify, which was then prompted to “install” the deployment, in other words, to effectively *deploy* it to the cloud.

This process if clearly depicted in the Cloudify Manager’s GUI which showcases not only a TOSCA model of the deployment, that can be verified in Figure 5.3, but the complete configuration, outputs and status progress of each node in it. After the “install” execution has concluded, as shown in Figure 5.10, a new virtual machine can be found on the Google project specified, where one can verify that the software desired is installed. The second stage has completed successfully, therefore the first two criteria for the prototype’s success have been achieved and, currently, only the autonomic life-cycle is left to be proven and verified.

Throughout every stage that has been analyzed so far, one could observe logging information, in the CLI, regarding the MAPE-K cycle. Since the current deployment was the only one present in the system (and is the only in a live status from this point on) all the information stated that no deployments had to be acted upon with any scaling action.

```
Starting Monitoring
Finished Monitoring
Starting Analyzing
Finished Analyzing with results [(1, 'INC')]
Starting Planning
Plans created with results [Action(target_id=1, behaviour=Scale(quantity=1, direction=<ScaleDirection.HORIZONTAL: 'HORIZONTAL'>))]
Executing Actions
Starting scaling
Waiting for deployment execution to be completed
```

Figure 5.6: Voting-App scaling action is required.

ID							
ID	Voting-App.GCP.gcp	scale	18-10-2022 23:32	18-10-2022 23:35	admin	✓ completed	☰
ID	Voting-App.GCP.gcp	install	18-10-2022 23:25	18-10-2022 23:29	admin	✓ completed	☰

Figure 5.7: Voting-App scale successful.

However, after some time elapsed, the *Monitor* component had enough opportunities to record several metric data points. This, in turn allows the *Analyzer* to provide an analysis over a relevant set of data points. This leads to the conclusion that the deployment with *id=1* has been under-performing, therefore a goal to increase the computational resources is given to the *Planner*. The *Planner*, after receiving the analysis report, creates a plan to apply a positive horizontal scaling action to the deployment and deliver is it to the *executor*, which then starts a “scaling” execution on Cloudify. The system now awaits the conclusion of the *scaling* after which the the life-cycle will continue for the specific deployment, which can be observed in Figure 5.6, since during the execution the autonomic stage only considers deployments who are “live” and ignores all other deployments that are scaling or “down”.

Eventually, the scale operation concludes and another process of deploying the TOSCA blueprint is

successful, as shown in Figure 5.7.

In addition, a new entire virtual machine can be verified to have been created in the GCP project identical to the original one, similarly having the necessary software components installed, as seen in Figure 5.8. With this, the stage of the autonomic life-cycle showcase is concluded and the third and final criteria for success has been achieved.

	Status	Name	Zone	Recommendations	In use by	Internal IP	Exte	Connect	
☐	✓	vm-ac1g56	europe-west1-b			10.132.0.16 (nic0)	35.2 (nic	SSH	⋮
☐	✓	vm-ihsfzo	europe-west1-b			10.132.0.17 (nic0)	35.2 (nic	SSH	⋮

Figure 5.8: State of the VM instances in GCP project after scale execution in Cloudify.

Lastly, all that is missing is the verification that Int2IT-Lite allows for a successful clean-up of the resources which were deployed. This is done by executing an “uninstall” execution for the deployment, triggered either with a *shutdown* or *destroy* operation from the CLI, which concludes successfully, as seen in Figure 5.9, leaving an empty project on GCP.

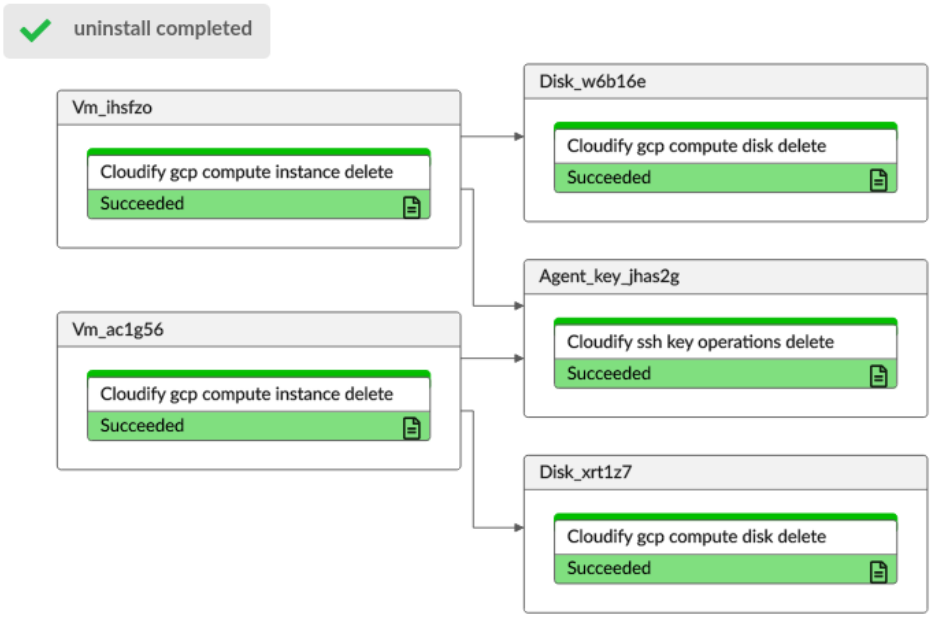

Figure 5.9: Voting-App clean up resources successful.

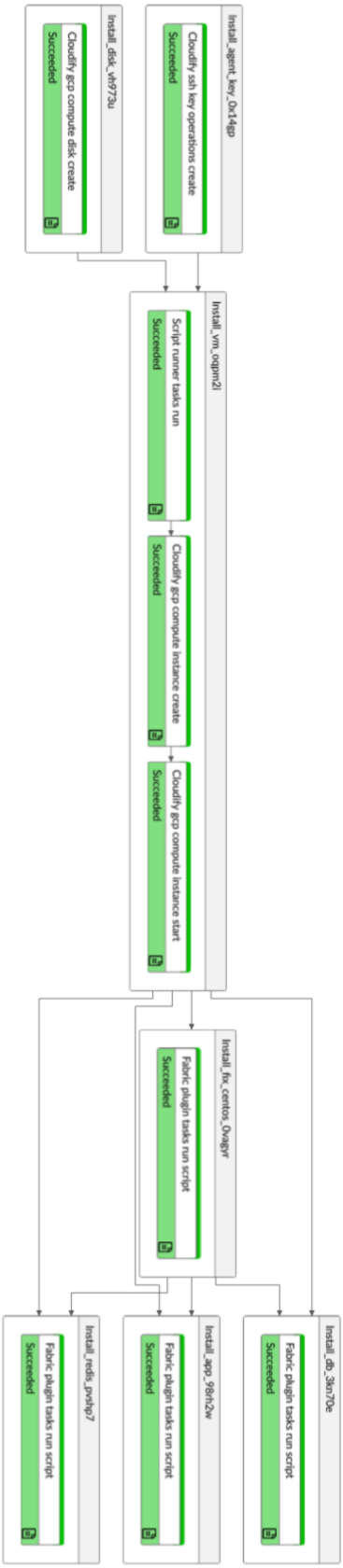

Figure 5.10: Voting-App install is successful.

6

Conclusion

Contents

6.1 Final Remarks

For the purposes of a master's dissertation, this document proposes a proof-of-concept solution to autonomous cloud orchestration via descriptive high-level abstractions referred to as "intents", which is inspired by the very recent trend of Intent-Based Networking, and focuses on achieving business level goals while concealing a significant amount of low-level implementation and maintenance details.

Furthermore, it aims to supplement popular DevOps methodologies and practices, specifically the use of IaC via the OASIS TOSCA standard, which has been extensively researched in academia but has not yet been widely adopted by businesses, but with promising results and favorable opinions from cloud experts who have adopted it at some point.

Before presenting the proposed solution, an academic literature review is conducted to provide an understanding of the already established current state of the art in the topics of IBN, DevOps and IaC, autonomic computing, and concluding with an in-depth review of the entire OASIS TOSCA previous works timeline.

The solution consists of an autonomic infrastructure management platform called Int2IT, which is divided into four stages: *Intent* processing, TOSCA processing, an autonomic deployment cycle, and a final orchestration layer. This solution is then verified experimentally by the development of a proof-of-concept tool named Int2IT-Lite, through the implementation of the Docker voting app use case in the GCP cloud provider.

To ensure the viability of this proof-of-concept, three main criteria had to be satisfied: (a) creation and deployment of a generated TOSCA archive, based on an *intent* description of the Docker voting app use case, into a GCP project through the Cloudify orchestrator; (b) the software components of the deployment are provisioned correctly and functional; (c) the deployment is able to be monitored, in an autonomic manner, and self-adapts according to the high-level metrics defined in the *intent*. It was shown the achievement of all three of these success criteria, therefore making Int2IT-Lite a viable proof-of-concept tool for the intent-based infrastructure problem.

6.2 System Limitations and Future Work

Before moving on to the next section, which depicts the architecture for the proposed proof-of-concept, it is necessary to clarify some aspects and topics that, due to time and work scope constraints, will not be addressed in Int2IT. Nonetheless, they may pique the interest of future researchers in this field, even possibly presenting as natural expansions of the proposed proof-of-concept.

6.2.1 Natural Language Intent Intelligent Processing

As stated previously, for the purpose of this paper *intents* are processed statically which means, only a finite number of instructions are supported, evidenced by the table of translations presented in Chapter 3. However, it would be interesting to further develop the translation component of Int2IT by adopting a Natural Language intelligent processing which would enable a more high-level abstraction, discarding the need for business-oriented *intents* to specify some elements of the infrastructure, and policies, in a specific formal manner by using keywords and expressions needed for the current Int2IT Translator. Moreover, a system as such would provide an enhanced scalability as it would be able to process varied definitions of *intents* which would ultimately be mapped to a certain specification.

6.2.2 Workload Predictive Elasticity

For this proof-of-concept, neither the Resource Manager nor the Optimizer components generate any predictions about future workloads and capacity in order to provision the current deployed infrastructure with additional resources (or discard potentially underutilized ones), anticipating new system requirements to accommodate the new demand. As a result, any expansion of this system should include elasticity capabilities in order to meet ever-changing demands without requiring reactive actions based on policies defined by the system *intent* specifications. Nevertheless, it is important to note that providing elasticity does not render policy specification and assurance obsolete, as it still provides the capability to adapt the system to a desired optimization when predicted workloads remain similar to current ones, implying that elasticity would primarily allow for more advanced scalability.

6.2.3 Infrastructure Drift Detection

Int2IT assumes that the current state of the deployed infrastructure is exactly as it has stored in its DataStore, which is done for simplicity as it would be out of scope for this project to consider a fully fledged operation and development business environment. However, it would be interesting and beneficial to provide the capability to detect specification changes caused by manual intervention in the cloud providers through GUI, byzantine faults during any of the operations, or even faults occurred in the CP-side. Tools such as *driftctl*[1] already provide this service for IaC tools, such as Terraform, and could be considered the basis of work for any further research on this topic. Additionally, instead of requiring manual intervention to fix any drifts in infrastructure, one possible approach or research topic could be to create some automation tool that not only detects drift, but also provides automatic correction, and even, possibly, prevention.

[1] https://driftctl.com

6.2.4 Security and Privacy

Many security and privacy aspects will be considered out of scope for this proof-of-concept and will not be given much focus other than the minimum acceptable security such as authentication in cloud environments with the use of a protocol such as *OAuth2* [24]. However, this topic is of utmost importance and, therefore, should be addressed in future works. From Guerriero et al. [3] findings, it is possible to observe that experts consider tools that support more automated security to be very important, ranking them as the fourth most important activity, right behind "languages standardization", which is, in fact, one of the goals of the OASIS TOSCA standard and a critical aspect in Int2IT. As a result, it may be worthwhile to investigate the feasibility of integrating security tools such as Hashicorp's Vault[2] into Int2IT or any TOSCA cloud deployment tool with the goal of extending the current state of the art.

6.2.5 IaC Converters Into OASIS TOSCA Standard

As pointed out by Guerriero et al. [3], "language standardization" in IaC is another critical aspect and a direct cause of the complexity in infrastructure management. As a result, it would be interesting for future academic works to create and propose implementations of IaC converters, the primary responsibility of which would be to receive as input the various IaC languages in use and convert them into valid TOSCA, allowing the conversion of previous deployments into a more standardized form of infrastructure description.

[2] https://www.vaultproject.io

Bibliography

[1] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero, and D. A. Tamburri, "DevOps: Introducing Infrastructure-as-Code," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. Buenos Aires: IEEE, May 2017, pp. 497–498.

[2] J. Bellendorf and Z. Á. Mann, "Specification of cloud topologies and orchestration using TOSCA: A survey," *Computing*, vol. 102, no. 8, pp. 1793–1815, Aug. 2020.

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

[4] A. Brogi, J. Soldani, and P. Wang, "TOSCA in a Nutshell: Promises and Perspectives," in *Advanced Information Systems Engineering*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, C. Salinesi, M. C. Norrie, and Ó. Pastor, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, vol. 7908, pp. 171–186.

[5] A. Vetter, "Detecting Operator Errors in Cloud Maintenance Operations," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. Luxembourg, Luxembourg: IEEE, Dec. 2016, pp. 639–644.

[6] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[7] M. D. Mascarenhas and R. S. Cruz, "Int2it: An intent-based tosca it infrastructure management platform," in *2022 17th Iberian Conference on Information Systems and Technologies (CISTI)*, 2022, pp. 1–7.

[8] G. Davoli, W. Cerroni, S. Tomovic, C. Buratti, C. Contoli, and F. Callegati, "Intent-based service management for heterogeneous software-defined infrastructure domains: Intent-based service management for heterogeneous software-defined infrastructure domains," *International Journal of Network Management*, vol. 29, no. 1, p. e2051, Jan. 2019.

[9] P.-J. Nefkens, *Transforming Campus Networks to Intent-Based Networking*, 2020.

[10] S. Hariri, B. Khargharia, H. Chen, J. Yang, Y. Zhang, M. Parashar, and H. Liu, "The Autonomic Computing Paradigm," *Cluster Computing*, vol. 9, no. 1, pp. 5–17, Jan. 2006.

[11] P. Lipton, D. Palma, M. F. Rutkowski, M. F. Rutkowski, and D. A. Tamburri, "TOSCA solves big problems in the cloud and beyond," *IEEE Cloud Computing*, 2018.

[12] M. Cankar, A. Luzar, and D. A. Tamburri, "Auto-scaling Using TOSCA Infrastructure as Code," in *Software Architecture*, H. Muccini, P. Avgeriou, B. Buhnova, J. Camara, M. Caporuscio, M. Franzago, A. Koziolek, P. Scandurra, C. Trubiani, D. Weyns, and U. Zdun, Eds. Cham: Springer International Publishing, 2020, vol. 1269, pp. 260–268.

[13] T. Waizenegger, M. Wieland, T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, B. Mitschang, A. Nowak, and S. Wagner, "Policy4TOSCA: A Policy-Aware Cloud Service Provisioning Approach to Enable Secure Cloud Computing," in *On the Move to Meaningful Internet Systems: OTM 2013 Conferences*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, R. Meersman, H. Panetto, T. Dillon, J. Eder, Z. Bellahsene, N. Ritter, P. De Leenheer, and D. Dou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 8185, pp. 360–376.

[14] OASIS, "Topology and Orchestration Specification for Cloud Applications Version 1.0," OASIS Standard, 2013.

[15] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "OpenTOSCA – A Runtime for TOSCA-Based Cloud Applications," in *Service-Oriented Computing*, P. P. Maglio, M. Weske, J. Yang, and M. Fantinato, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 6470, pp. 692–695.

[16] A. Brogi, D. Neri, L. Rinaldi, and J. Soldani, "Orchestrating incomplete TOSCA applications with Docker," *Science of Computer Programming*, vol. 166, pp. 194–213, Nov. 2018.

[17] A. Brogi, A. Di Tommaso, and J. Soldani, "Sommelier: A Tool for Validating TOSCA Application Topologies," in *Model-Driven Engineering and Software Development*, L. F. Pires, S. Hammoudi, and B. Selic, Eds. Cham: Springer International Publishing, 2018, vol. 880, pp. 1–22.

[18] A. Sampaio, T. Rolim, N. C. Mendonça, and M. Cunha, "An Approach for Evaluating Cloud Application Topologies Based on TOSCA," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, Jun. 2016, pp. 407–414.

[19] D. A. Tamburri, W.-J. Van den Heuvel, C. Lauwers, P. Lipton, D. Palma, and M. Rutkowski, “TOSCA-based Intent modelling: Goal-modelling for infrastructure-as-code,” *SICS Software-Intensive Cyber-Physical Systems*, vol. 34, no. 2-3, pp. 163–172, Jun. 2019.

[20] 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: SCITEPRESS - Science and Technology Publications, 2020, pp. 216–226.

[21] M. Bogo, J. Soldani, D. Neri, and A. Brogi, “Component-aware orchestration of cloud-based enterprise applications, from TOSCA to Docker and Kubernetes,” *Software: Practice and Experience*, vol. 50, no. 9, pp. 1793–1821, Sep. 2020.

[22] O. Tomarchio, D. Calcaterra, G. Di Modica, and P. Mazzaglia, “TORCH: A TOSCA-Based Orchestrator of Multi-Cloud Containerised Applications,” *Journal of Grid Computing*, vol. 19, no. 1, p. 5, Mar. 2021.

[23] J. DesLauriers, T. Kiss, R. C. Ariyattu, H.-V. Dang, A. Ullah, J. Bowden, D. Krefting, G. Pierantoni, and G. Terstyanszky, “Cloud apps to-go: Cloud portability with TOSCA and MiCADO,” *Concurrency and Computation: Practice and Experience*, vol. 33, no. 19, Oct. 2021.

[24] D. Hardt, “The OAuth 2.0 authorization framework,” RFC 6749, Oct. 2012.