Int2IT: An Intent-based TOSCA IT Infrastructure Management Platform

Manuel Duarte Mascarenhas

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. 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.

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

1 Introduction

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 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.

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]:

(a) “Automated application deployment and management”; (b) “Portability of application descriptions and

• *Manuel Duarte Mascarenhas, nr. 90751,*
E-mail: manuel.d.mascarenhas@tecnico.ulisboa.pt,
Instituto Superior Técnico, Universidade de Lisboa.

Manuscript received October 31, 2022.

their management"; (c) "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 work attempts to contribute to solve, with the development of a proof-of-concept for an autonomic intent-based OASIS TOSCA infrastructure management tool.

The developed tool, named Int2IT, 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 [7].

1.2 Goals

Int2IT is an Intent-Based autonomic infrastructure management platform that uses OASIS TOSCA for IaC standardization and infrastructure provisioning. Therefore, the goals of this paper can be summarized as:

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

2 Background

In this section some background concepts and clarifications are provided for the most relevant topics that this work 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: (a) Taking trash from bin in kitchen and wrapping it; (b) Carry it outside and dump it in the containers; (c) 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.

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. The model is made up of five components: Monitor, Analyze, Plan, Execute, and Knowledge. In Section 3.3, 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].

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 subsection 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.

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).

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

1. https://projects.eclipse.org/projects/soa.winery

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 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.

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 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.

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

2. https://www.docker.com
3. https://cloudify.co
4. https://www.terraform.io
5. https://kubernetes.io

OpenStack cloud of the University of Westminster and, finally, to Microsoft Azure.

3 Int2IT

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.

3.1 Intent Processing

The primary purpose of this stage is to process the captured *intents* via the Graphical User Interface (GUI) or Command Line Interface (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.

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.

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 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.

3.2 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.

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*.

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.

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.

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.

3.3 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.

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.

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.

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.

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.

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.

3.4 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.

4 INT2IT-LITE

Int2IT-Lite was developed entirely with Python 3.8.6, utilizing some key packages, namely: (a) cloudify-rest-client/ cloudify-common [6] [7]; (b) pymongo [8]; (c) pandas [9]; (d) pydantic [10]; (e) pyaml/oyaml [11] [12]. In addition, some containerized applications were utilized for the correct functioning of the prototype, namely: (a) Cloudify-Community:6.4 [13] and (b) MongoDB:4.2.21-rc0 [14].

MongoDB [15] 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 [16].

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 [17], installation websites as well as fixes to known errors and problems.

4.1 Architecture Design

The prototype has an architecture closely following the one of the ideal Int2IT solution. However, some components have been simplified or have been integrated together with into 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. 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*.

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

6. https://github.com/cloudify-cosmo/cloudify-rest-client
7. https://github.com/cloudify-cosmo/cloudify-common
8. https://pypi.org/project/pymongo/
9. https://pandas.pydata.org/
10. https://pydantic-docs.helpmanual.io/
11. https://pyyaml.org/wiki/PyYAML
12. https://pypi.org/project/oyaml/
13. https://hub.docker.com/r/cloudifyplatform/community-cloudify-manager-aio
14. https://hub.docker.com/_/mongo
15. https://www.mongodb.com/
16. https://www.mongodb.com/docs/manual/sharding/
17. https://github.com/cloudify-community/blueprint-examples

Monitor component for all deployments, which have an *UP* deployment status, so that the Analyzer may afterwards process.

The components *Topology Generator* and *Discovery* described in Int2IT are not featured prominently on Int2IT-Lite, because they have either been simplified or incorporated into other components. 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.

4.2 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.

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. After an *Order* representation is achieved, it is then passed into the *Registry*, which stores it into the "orders" collection in the database.

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.

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*.

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. Lastly, the gathered selector is utilized for a "deep update" with the *Order*, in a dictionary format.

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.

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.

4.3 Autonomic Life-Cycle

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.

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 [18] [19]; (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*.

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.

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".

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 and vice-versa.

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. The executor then awaits the completion of each scale execution and, afterwards, the autonomic life-cycle is restarted.

18. https://docs.python.org/3.8/library/uuid.html#uuid.uuid4
19. https://www.rfc-editor.org/rfc/rfc4122.html

5 Results

In this section, an overview is given over the use case utilized for demonstration purposes of the capabilities of the developed prototype.

As stated in the original GitHub's repository [20], 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.

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.

5.1 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 [21].

Lastly, for the monitoring part of the deployment a Flask [22] 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. 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.

20. https://github.com/dockersamples/example-voting-app
21. https://github.com/cloudify-community/blueprint-examples/releases/download/6.3.0-10/virtual-machine.zip
22. https://flask.palletsprojects.com/en/2.2.x/

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, present in a YAML file named "*webvote*". It is assumed that the author behind the *intent* has configured everything correctly when it comes to the values and any endpoint connections specified. In this *intent*, the metadata regarding the deployment concretely specifies not only the cloud provider to be used, but the specific orchestrator as well. The services block in the intent 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. A "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.

5.3 Deployment Results

A step-by-step analysis of the outcomes of the processing and upholstering of the use case *intent* provided the following results:

Firstly, the *intent* was provided to Int2IT-Lite through the CLI by attempting a *register* operation on the *webvote.yaml* file. 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 the contents of the *Order*, where it was possible to verify that the contents of the *intent* are accurately represented.

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 was clearly depicted in the Cloudify Manager's GUI which showcased not only a TOSCA model of the deployment, but the complete configuration, outputs and status progress of each node in it. After the "install" execution has concluded, a new virtual machine could be found on the Google project specified, where it was possible to verify that the software desired was installed.

As the second stage had completed successfully, therefore the first two criteria for the prototype's success have been achieved and, currently, only the autonomic life-cycle was left to be proven and verified.

Throughout every stage that has been analyzed so far, it was possible to observe logging information, in the CLI, regarding the MAPE-K cycle. Since the current deployment was the only one present in the system (and was 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.

However, after some time elapsed, the *Monitor* component had enough opportunities to record several metric data points. This, in turn allowed the *Analyzer* to provide an analysis over a relevant set of data points. This leads to the conclusion that the deployment with *id=1* had been under-performing, therefore a goal to increase the computational resources was given to the *Planner*. The *Planner*, after receiving the analysis report, created a plan to apply a positive horizontal scaling action to the deployment and deliver it to the *executor*, which then started a "scaling" execution on Cloudify. The system then awaited for the conclusion of the *scaling* after which the life-cycle continued for the specific deployment, since during the execution the autonomic stage only considers deployments which are "live" and ignores all other deployments that are scaling or "down". Eventually, the scale operation concluded and another process of deploying the TOSCA blueprint was successful.

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

Lastly, all that was missing was the verification that Int2IT-Lite allows for a successful clean-up of the resources that were deployed. This was done by executing an "uninstall" execution for the deployment, triggered either with a *shutdown* or *destroy* operation from the CLI, which concluded successfully, leaving an empty project on GCP.

6 Conclusion

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 was conducted to provide an understanding of the already established current state of the art in the topics of Intent Based Networking (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 layers: *Intent* processing, TOSCA processing, an autonomic deployment cycle, and a final orchestration layer. This solution was 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.

References

[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.