# Chapter 4
# Quality Assurance and Design-Time Optimization


**Indika Kumara, Alfio Lazzaro, Nina Mujkanovic, Zoe Vasileiou, and Damian A. Tamburri**

**Abstract** Heterogeneous applications are getting more and more complex, making the authoring of their deployment models an error-prone and demanding task. Heterogeneous resources also make performance optimization of applications complex. In this chapter, we will present the quality assurance and application optimization support of the SODALITE framework, which offers the capabilities for verifying deployment models, detecting bugs and smells in them, and optimizing application components for specific hardware resources. This chapter discusses how the above-mentioned capabilities of the SODALITE framework can be used to develop optimized, defect-free deployment models.

## 4.1 Introduction

The SODALITE modeling layer produces the deployment model of an application in terms of Infrastructure as Code (IaC) scripts. IaC simplifies the provision and configuration of the IT infrastructure at scale. As the size and complexity of IaC

I. Kumara (✉)
Jheronimus Academy of Data Science, Tilburg University, Tilburg, The Netherlands
e-mail: i.p.k.weerasinghadewage@tilburguniversity.edu

A. Lazzaro · N. Mujkanovic
HPE HPC/AI EMEA Research Lab, Basel, Switzerland
e-mail: alfio.lazzaro@hpe.com

N. Mujkanovic
e-mail: nina.mujkanovic@hpe.com

Z. Vasileiou
Information Technologies Institute, Centre for Research and Technology Hellas, Marousi, Greece
e-mail: zvasilei@iti.gr

D. A. Tamburri
Jheronimus Academy of Data Science, Technical University Eindhoven, Eindhoven, The Netherlands
e-mail: d.a.tamburri@tue.nl



E. Di Nitto et al. (eds.), *Deployment and Operation of Complex Software in Heterogeneous Execution Environments*, PoliMI SpringerBriefs,
https://doi.org/10.1007/978-3-031-04961-3_4

projects increase, it is critical to maintaining the code and design quality of IaC Scripts [6, 7] According to a recent report on Cloud Threat,[1] nearly 200,000 insecure IaC templates were found among IaC scripts used by a set of enterprises, and 65% of cloud incidents are due to misconfigurations. Thus, the detection and correction of defective and erroneous IaC scripts are of paramount importance. To address this problem, the SODALITE platform offers a set of tools to detect defects such as errors and code smells.

In addition to the generation of the IaC scripts, the deployment process can also create container images for application components. Software application developers and users are now targeting diverse computing platforms, such as on-premise supercomputers and clouds with heterogeneous node architectures. Compute intensive applications such as Artificial Intelligence (AI) training that use High-Performance Computing (HPC) have specific requirements for specialized execution environments, including computing accelerators, high speed interconnects, and fast memory and storage. Even if software-defined environments provide flexibility and portability, we still need applications to use and benefit from these diverse resources optimally. For example, AI training frameworks require target-specific libraries and drivers to be configured. In the context of HPC infrastructures, with various hardware and software dependencies and libraries, building or selecting an optimized container for deploying AI-based components is crucial. The same concepts apply to Message Passing Interface (MPI) applications, where the applications have to efficiently use the network to get performance and parallel scalability. To address these issues, SODALITE offers an application optimizer called MODAK that maps the optimal application parameters to the infrastructure target by building or selecting an optimized container and then encoding optimizations in a job script.

The rest of this chapter is organized as follows. Section 4.2 presents the support for validating the deployment topology of an application and verifying arbitrary constraints on the components and their properties. Section 4.3 discusses the detection of smells and bugs in IaC using rule-based and data-driven approaches. Section 4.4 presents the MODAK tool in detail, and Sect. 4.5 concludes the chapter.

## 4.2 Verifying IaC

Validation services are provided to the user during the authoring process of the deployment models. Based on the deployment models saved as interconnected Knowledge Graphs, described in Chap. 3, powerful semantic queries can run upon the Knowledge Base using strong inference for uncovering new information out of existing relations. Additional to advanced context-aware searching, matchmaking, and reuse, described in the previous chapter, pre-deployment validation is a crucial component that ensures a reliable IaC deployment model.

[1] https://www.paloaltonetworks.com/prisma/unit42-cloud-threat-research-1h21.

The validation of the AADM, during the design phase, is aiming at checking the consistency of the structures. In TOSCA, the type system supports inheritance as a type can extend another, inheriting all its concepts (e.g. properties, capabilities). Each template of the AADM is an instance of a specific type, namely an infrastructure resource or software component, and gets validated against this type definition.

### *4.2.1 Validation Cases*

Using custom reasoning logic, semantic validation errors can be inferred with regards to the TOSCA type definition. The assigned values to the component templates are validated against the corresponding type schema.

#### 4.2.1.1 Topology Validation

There are errors in the deployment model that are onerous to be manually detected as it is needed to manually check all inter-node relationships in a TOSCA application topology and their interconnection constraints. Based on the validity conditions of the Sommelier [3], an open-source validator of TOSCA application topologies, our services are validating the interconnections of the deployment model. All the TOSCA elements, that are forming a relationship, are checked, namely the source (Requirements of a node), the relationship itself, and its target (a node or a capability of a node).

In TOSCA [9], various components, such as an application, a database, are modeled as templates and are instances of types, such as node types, relationship types, and capability types. The node types contain the definitions of the requirements of a component, the capabilities that are offered for other components. The capability types express the capabilities and `valid_source_types` (valid names of Node Types that are supported as valid sources of any relationship). The relationship types denote the explicit relationships between the nodes, or alternatively implicit relationships are declared through requirements.

#### 4.2.1.2 Required Properties

In the type schema, it is optionally to be defined if a property is required to be assigned to a template by the *required* key. Therefore, if there exists a property definition in a type and *required* equals true, and there is no *default* value, then such a property should be assigned to the templates being instances of this type. In Listing 4.1, a TOSCA node type definition is depicted with the *name* mandatory property. In Listing 4.2, a SPARQL query detecting the required properties is shown.

```yaml
node_types:
    sodalite.nodes.DockerNetwork:
        derived_from: tosca.nodes.SoftwareComponent
        properties:
            name:
                description: "The name of the network"
                type: string
                required: true
```

List. 4.1: Excerpt of a TOSCA node type definition with a required property

```
select distinct ?property
where {
    ?resource soda:hasInferredContext ?context .
    ?context tosca:properties ?concept .
    ?concept DUL:classifies ?property .
    {
        ?concept DUL:hasParameter [DUL:classifies tosca:required; tosca:hasDataValue
        ↪  true].
    } UNION {
        FILTER NOT EXISTS {?concept DUL:hasParameter
                           [DUL:classifies tosca:required; tosca:hasDataValue []]}.
    }
}
```

List. 4.2: SPARQL Query detecting required properties

#### 4.2.1.3 Property Values

Each property definition of the node type includes a type of the assigned property value. There are various property types such as `string`, `integer`, `list`, and `map`. A node type that has two properties with the type string and integer defined is shown in Listing 4.3. Rule-based reasoning infers if the assigned template property values are valid according to the type, using SPARQL queries upon the Knowledge Graphs.

```yaml
sodalite.nodes.DockerizedComponent:
    derived_from: tosca.nodes.SoftwareComponent
    properties:
        client_key:
            description: "Path tot he client's TLS key file."
            type: string
            required: false
            default: ""
        sleep:
            description: "Sleep after image is deployed"
            type: integer
            required: false
            default: 0
```

List. 4.3: Part of a TOSCA node type with properties of different types

#### 4.2.1.4 Constraints

A constraint clause might be optionally present in the property definition of the type defining the allowed values that can be assigned in the corresponding template property. The constraints can be as simple as a list with valid values, shown in Listing 4.4 or a given range (e.g. greater than, less than), or as complex as an object of a custom type. In Listing 4.5, a SPARQL query is shown that retrieves the properties of a type that have constraints with a list.

```
sodalite.datatypes.modak.optimisation.opt_build:
    derived_from: tosca.datatypes.Root
    properties:
        cpu_type:
            type: string
            constraints:
                - valid_values: [ "x86", "arm", "amd", "power" ]
        acc_type:
            type: string
            constraints:
                - valid_values: [ "nvidia", "amd", "fpga" ]
```

List. 4.4: Part of a TOSCA data type with property value constraints

```
select distinct ?constraint ?constr_type ?value ?listvalue
where
{
    ?var soda:hasInferredContext ?context .
    ?context tosca:properties ?concept .
    ?concept DUL:classifies ?property .
    ?concept DUL:hasParameter [DUL:classifies tosca:CONSTRAINTS; DUL:hasParameter
        [DUL:classifies ?constraint; tosca:hasValue ?listvalue]].
    ?listvalue rdf:type tosca:List .
    ?concept DUL:hasParameter [DUL:classifies tosca:type; tosca:hasValue ?constr_type].
}
```

List. 4.5: SPARQL Query returning only the constraints of a type including a list

## 4.3 Detecting Smells and Linguistic Anti-patterns in IaC

SODALITE developed the tools that can detect such smells and linguistic anti-patterns in IaC. A software smell is any characteristic in the artifacts of the software that possibly indicates a deeper problem or quality issue [11]. Linguistic anti-patterns are recurring poor practices concerning inconsistencies among the naming, documentation, and implementation of an entity, which have shown to be a good proxy for defect prediction [1].

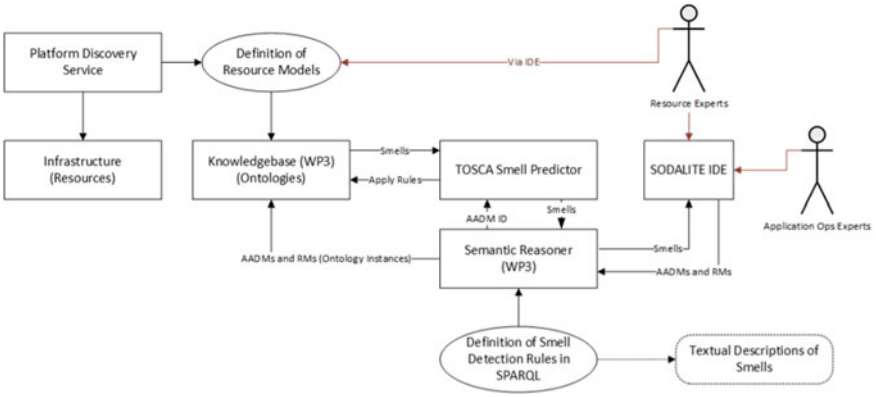


**Fig. 4.1** An Overview of our Approach to TOSCA Smell Detection

### *4.3.1 Semantic Approach to Detecting Smells*

SODALITE proposes a semantic rule-based approach to detect the smells and antipatterns in IaC, for example, smells in TOSCA blueprints [8]. Our framework facilitates the generation of knowledge graphs to capture TOSCA-based deployment models. The aim is to map IaC code constructs to self-contained, independent, and reusable knowledge components, amenable to analysis and validation using Semantic Web standards, such as SPARQL. A semantic approach helps us to deal with structure and semantic relations over various types of resources, their relationships, and properties. The semantic reasoning process is able to draw new and hidden knowledge from the existing information.

Figure 4.1 shows the high-level architecture and workflow of our approach to detect the occurrences of smells in deployment model descriptions. More specifically:

- **Population of Knowledgebase.** Resource Experts populate the knowledgebase by creating resource models (ontology instances representing resources/nodes in the infrastructure) using SODALITE IDE. Platform Discovery Service may (semi-)automatically update the knowledge base by creating resources models.
- **Definition of Smells Detection Rules.** We use the semantic rules in SPARQL to detect different smells in deployment models. SODALITE developed rules to detect common security and implementation smells. New, additional rules can be defined to detect new types of smells.
- **Detection of Smells.** Application Ops Experts create the AADM instances for representing the deployment models of the applications. The AADM is automatically translated into the corresponding ontological representation and is saved in the knowledgebase. The smell detection rules are applied over the ontologies in the knowledgebase to detect deployment model-level smells. If a smell is detected, the details of the smell are returned to the Application Ops Experts. The detected

**Table 4.1** Smells, their descriptions, and the abstract detection rules

| Smell | Smell Description | Abstract Detection Rule |
|---|---|---|
| Admin by default | Default users are administrative users | isUser (x.name) ∧ isAdmin (x.name) |
| Empty password | A password as a zero-length string | isPassword(x.name) ∧ (isEmpty(x.value) isEmpty(x.defaultValue)) |
| Hard-coded secret | Secrets such as usernames and passwords are hardcoded | (isPassword(x.name) ∨ isUser (x.name) ∨ isSecKey (x.name)) ∨ ∼(isEmpty(x.value) ∨ isEmpty(x.defaultValue)) |
| Suspicious comment | A comment includes the information indicating secrets and buggy implementations, etc. | hasComment(x) ∧ isSuspicious(x.comment) |
| Unrestricted IP address | Using "0.0.0.0" or "::" as binding IP addresses of servers | isIPAddress(x.name) ∧ (isInvalidBind (x.value) ∨ isInvalidBind (x. defaultValue)) |
| Insecure Communication | Using insecure communicate protocols, instead of secure their counterparts | (isURL(x.value) ∧ isInsecure(x.value)) ∨ (isURL(x.defaultvalue) ∧ isInsecure(x.defaultvalue)) |
| Weak Crypto. Algo. | Use of weak cryptography algorithms such as MD5 and SHA1 | hasWeakAlgo(x.value) ∨ hasWeakAlgo(x.defaultvalue) |
| Insufficient Key Size | The key used by an encryption algorithm is less than the recommended key size, e.g., 2048 bits for RSA algorithm | isCryptoKeySize(x.name) ∧ (hasInsufficientKey-Size(x.value) ∨ hasInsufficientKey-Size(x.defaultvalue)) |
| Inconsistent naming convention | The conventions used for naming nodes, properties, attributes, etc., are inconsistent | (case=='CamelCase'→ isCamelCase(x)) ∨ (case=='SnakeCase'→ isSnakeCase(x)) ∨(case=='DashCase'→ isDashCase(x)) |
| Invalid Port Ranges | TCP port values are not within the range from 0 to 65535 | isTCPPort(x) ∨ (outOfValidRange (x.value) ∧ outOfValidRange (x.defaultvalue)) |

smells are shown in the IDE as warnings. The same flow applies to Resource Ops Experts, as they also receive warnings for their resource models.

Table 4.1 shows the (abstract) rules to detect 10 TOSCA smells. The rules are implemented as SPARQL queries for specifying detection rules. Listing 4.6 shows

an excerpt from the SPARQL query for detecting Admin by default smell. Line 4 implements the function is User using a regex matching. Lines 5–9 retrieve the default value for a property of a node. Line 14 realizes the function isAdmin using the IN operator. The SPARQL queries for the other smells are available online in the SODALITE GitHub repository.

```
1  select distinct ?property ?propertyDef
2  where {
3      ?property DUL:classifies ?propertyDef.
4      FILTER(regex(str(?propertyDef),"user(.+?)|(.+?)?user","i")).
5      optional { # node type definitions - tier1
6          ?property DUL:hasParameter ?p .
7          ?p DUL:classifies tosca:default .
8          ?p tosca:hasDataValue ?value.
9        }.
10     optional { #node template definitions - tier0
11         ?property tosca:hasDataValue ?value.
12       }
13     FILTER (bound(?value)).
14     FILTER (str(?value) IN ('admin', 'root'))
15 }
```

List. 4.6: Part of AdminByDefault SPARQL Query.

### 4.3.2 *A Learning-Based Approach for Detecting Linguistic Anti-patterns*

We develop a novel approach to detect linguistic anti-patterns in IaC using deep learning and word embeddings [2]. We focus on name-body inconsistencies in IaC code units, for example, tasks in Ansible playbooks or roles. We use the Convolutional Neural Networks (CNN) [5] as the deep learning algorithm, and Word2Vec [4] as the word embedding method. CNNs are neural networks that consist of neurons with learnable weights and biases. Word2vec is a two-layer neural network that processes text by creating vector representations from words.

Figure 4.2 shows the workflow of our approach:

- **Corpus Tokenization.** Given a corpus of Ansible tasks, this phase generates token streams for both task names and bodies. To tokenize a task's body while considering its semantic properties, we build and use its abstract syntax tree.
- **Data Sets Generation.** Finding a sufficient number of real buggy task examples containing inconsistencies is challenging. Therefore, as in [10], we apply simple code transformations to generate buggy examples from likely correct examples. We perform such transformations on the tokenized data set and assume that most corpus tasks do not have inconsistencies.
- **From Datasets to Vectors.** We employ Word2Vec to convert the token sequences into distributed vector representations (code embeddings). We train a deep learning model for each Ansible module type as our experiments showed a single model does not perform well, potentially due to low token granularity. Thus, the tokenized

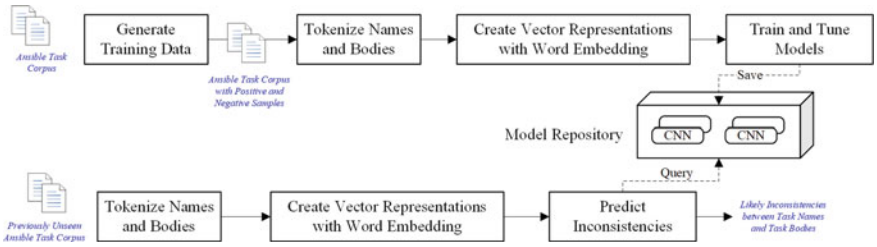


**Fig. 4.2** Linguistic anti-patterns detection pipeline

data set is divided into subsets per module, and the code embeddings for each subset are separately generated.

- **Model Training.** This phase feeds the code embeddings to a CNN model and trains the model to distinguish between the tasks having name-body inconsistencies from correct tasks. The trained model is stored in the model repository.
- **Inconsistency Detection.** The trained models (classifiers) from the model repository are employed to predict whether the name and body of a previously unseen Ansible task are consistent or not. Each task is transformed into its corresponding vector representations, which can be consumed by a classifier.

We evaluated our approach with an Ansible dataset systematically collected from open source repositories. Table 4.2 presents the inconsistency detection results for the top 10 Ansible modules in our data set. Overall, our approach yielded an accuracy ranging from 0.785 to 0.915, AUC metric from 0.779 to 0.914, and MCC metric from 0.570 to 0.830. Our approach achieved the highest performance for detecting inconsistency in the file module, where the accuracy was 0.915, the F1 score for the inconsistent class was 0.92, and the F1 score for the consistent class was 0.91.

## 4.4 Optimizing Containerized Applications

The MODAK (Model Optimized Deployment of Applications in Containers) package, a software-defined optimization framework for containerized MPI and AI applications, is the SODALITE component responsible for enabling the static optimization of applications before deployment. Application optimization is enabled using performance modeling and container technology. Containers provide an optimized runtime for application deployment based on the target hardware and along with any software dependencies and libraries. MODAK aims to manage the optimized application containers for the deployment to infrastructure in a software-defined way.

**Table 4.2** Classification results for the top 10 used Ansible modules

| Evaluation metric/module | | Shell | Command | set_fact | Template | File | gather_facts | Copy | Service | Debug | Fail |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Inconsistent | Precision | 0.880 | 0.790 | 0.770 | 0.820 | 0.900 | 0.900 | 0.860 | 0.870 | 0.870 | 0.820 |
| | Recall | 0.810 | 0.840 | 0.900 | 0.940 | 0.940 | 0.830 | 0.810 | 0.760 | 0.770 | 0.690 |
| | F1 score | 0.843 | 0.814 | 0.830 | 0.876 | 0.920 | 0.864 | 0.834 | 0.811 | 0.817 | 0.749 |
| Consistent | Precision | 0.810 | 0.820 | 0.890 | 0.930 | 0.930 | 0.905 | 0.82 | 0.800 | 0.750 | 0.760 |
| | Recall | 0.890 | 0.770 | 0.750 | 0.800 | 0.890 | 0.770 | 0.870 | 0.900 | 0.860 | 0.870 |
| | F1 score | 0.848 | 0.794 | 0.814 | 0.860 | 0.910 | 0.870 | 0.844 | 0.847 | 0.801 | 0.811 |
| Accuracy | | 0.847 | 0.805 | 0.819 | 0.868 | 0.915 | 0.817 | 0.838 | 0.833 | 0.809 | 0.785 |
| MCC | | 0.697 | 0.610 | 0.649 | 0.744 | 0.830 | 0.685 | 0.678 | 0.669 | 0.625 | 0.570 |
| AUC | | 0.848 | 0.804 | 0.822 | 0.868 | 0.914 | 0.848 | 0.838 | 0.830 | 0.814 | 0.779 |

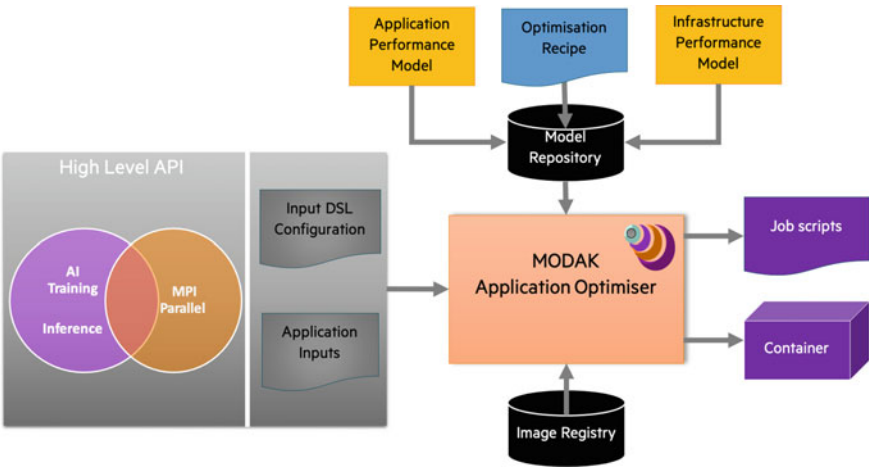


**Fig. 4.3** MODAK architecture

### *4.4.1 Architecture*

Figure 4.3 gives an overview of the MODAK components. MODAK exposes a high-level application API for the two types of applications supported: AI training and inference and MPI-parallelized applications. We pass this information to MODAK, which matches it with the performance model outputs to produce a job script for the execution submission of the optimized container. MODAK can also auto-tune and auto-scale applications based on user input. MODAK requires the following inputs:

- Job submission options for batch schedulers such as SLURM and TORQUE
- Application configuration such as application name, run and build commands
- Optimization DSL with the specification of the target hardware, software libraries, and optimizations to encode, as well as inputs for auto-tuning and auto-scaling. Examples of the DSL are provided in Sect. 6.4.4.

After providing the inputs, MODAK produces a job script (for batch submission) and retrieves a pre-built optimized container that can be used for application deployment. An image registry contains MODAK optimized containers, while performance models, optimization rules, and constraints are stored and retrieved from the Model repository. The Singularity container technology was chosen to provide a portable and reproducible runtime for the application deployment, due to better performance and native support for HPC resources than other popular container technologies. In the Sect. 4.4.2 we describe in detail each MODAK component with the related features.

### 4.4.2 Features

MODAK automates optimization using four main components, as described below:

- Mapper. The Mapper maps application deployment to an optimized container based on the user-specified input (DSL). While most AI applications are deployed in containers, this is not the default option for MPI parallel applications. Containers should provide an optimized runtime for the application deployment. With diverse hardware and software dependencies and libraries, building or selecting an optimized container for application deployment is crucial. For example, MPI libraries on the host machine and in the container should match when deploying applications on HPC systems in order for the container to use the hardware-optimized version of MPI available on the host. AI training frameworks require target-specific libraries and drivers to be configured. Even though Docker and Singularity support labeling containers, they are seldom used when developing them. To overcome this issue, containers are pre-built for different hardware and MODAK labels them with supported hardware and software information, including any optimizations. An application user uses a similar JSON format to query for an optimized container, and the mapper returns the container type, location, and file name. The user can pull the container from the hub and execute the application with that runtime. Currently, MODAK supports TensorFlow, PyTorch, MXNet, MPICH, OpenMPI, and MVAPICH2 containers for x86 and NVIDIA GPUs. This can be further extended to support specific network interconnects, and storage filesystems like Lustre.
- Enforcer. The optimization process depends not only on application and infrastructure but also on the configuration and data. MODAK allows users to define optimization rules that are enforced for deployment. The Enforcer component returns the optimization script to be used based on the rules and user-selected optimizations in the input DSL. For example, enabling graph compiler-based optimizations in an AI framework requires environment settings to be modified. For MPI-based applications, there are many environment settings that change the way message passing is optimized based on message size and communication pattern. Data-related optimizations may involve the possibility to automatically copy the data to fast disks, if available, to improve I/O bound applications. MODAK can embed the chosen optimizations in the job script submitted to a batch scheduler.
- Autotune. Applications and their dependencies have many configurable parameters which can drastically change performance when altered. Tuning all the parameters is both resource-intensive and time-consuming. Autotuning frameworks help make automated choices regarding application build and deployment, the algorithms they use, and the way the application is launched or changes code.
- Autoscale. Scaling applications to more nodes improves the performance of most MPI parallel applications. The parallel speedup and scaling efficiency is defined as follows

$$\text{Parallel Speedup} = \frac{T_{\text{ref}}}{T_{\text{parallel}}} \tag{4.1}$$

$$\text{Efficiency} = \frac{n_{\text{ref}}\, T_{\text{ref}}}{n\, T_{\text{parallel}}} \tag{4.2}$$

where $T_{\text{ref}}$ and $T_{\text{parallel}}$ correspond to the runtime on a reference number of nodes $n_{\text{ref}}$ (usually a single node), and the runtime on $n$ nodes, respectively. While we aim to achieve higher speedups as we increase nodes, poor efficiency denotes higher overheads and higher costs. Applications are usually scaled until the efficiency drops below a certain percentage. In MODAK, we can predict the efficiency and speedup of an application on $n$ nodes based on the performance prediction model. This allows MODAK to automatically scale applications to a certain number of nodes based on the model prediction. Using the parallel efficiency metric specified by the user, Autoscale aims to predict the scale at which parallel efficiency is achieved, and automatically increase the number of nodes of the deployment.

## 4.5 Conclusion and Future Work

In this chapter, we have presented the design-time quality assurance and optimization support of the SODALITE framework. To enable the deployment of defect-free IaC scripts, we offer the tools to verify IaC scripts against various constraints, and defect smells and linguistic anti-patterns in them. We use semantic rule-based techniques and deep learning-based techniques, as appropriate. Moreover, to optimize AI or MPI workloads with different configurations and data sets for heterogeneous infrastructure targets, we introduced MODAK, a novel tool that maps optimal application parameters to infrastructure using performance modeling and container technology. MODAK optimized containers were tested on the internal SODALITE HPC Testbed. The test scenarios were taken from the SODALITE use cases compute-intensive tasks. We found that the performance boost of using optimized application containers can reach up to 10x compared wit the unoptimized versions of the application.

As future work, we plan to extend our smell and defect detection support to detect more linguistic inconsistencies and misconfigurations in different IaC languages. We will also extend MODAK to support machine learning applications for the edge.

## References

1. Arnaoudova V et al (2013) A new family of software anti-patterns: linguistic anti-patterns. In: 2013 17th European conference on software maintenance and reengineering, pp 187–196. https://doi.org/10.1109/CSMR.2013.28
2. Borovits N et al (2020) DeepIaC: deep learning-based linguistic anti- pattern detection in IaC. In: MaLTeSQuE 2020. Virtual. Association for Computing Machinery, USA, pp 7–12
3. Brogi A, Tommaso AD, Soldani J (2017) Sommelier: a tool for validating TOSCA application topologies. In: 5th International conference on model- driven engineering and software development, pp 1–22

4. Kenneth Ward Church (2017) Word2Vec. Nat Lang Eng 23(1), 155–162. https://doi.org/10.1017/S1351324916000334
5. Goodfellow I, Bengio Y, Courville A (2016) Deep learning. MIT Press
6. Kumara I et al (2021) Quality Assurance of heterogeneous applications: the SODALITE approach. In: Zirpins C et al (ed) Advances in service-oriented and cloud computing. Springer International Publishing, Cham, pp 173–178. ISBN: 978-3-030-71906-7
7. Kumara I et al (2021) The do's and don'ts of infrastructure code: a systematic gray literature review. Inf Softw Technol 137:106593. ISSN: 0950-5849. https://doi.org/10.1016/j.infsof.2021.106593. https://www.sciencedirect.com/science/article/pii/S0950584921000720
8. Kumara I et al (2020) Towards semantic detection of smells in cloud infrastructure code. In: WIMS 2020. Association for Computing Machinery, Biarritz, France, pp 63–67
9. Lipton P et al (2020) Tosca simple profile in YAML version 1.3. In: OASIS committee specification 1
10. Pradel M, Sen K (2018) DeepBugs: a learning approach to name- based bug detection. In: Proceedings of ACM programing language 2. https://doi.org/10.1145/3276517
11. Sharma T, Spinellis D (2018) A survey on software smells. J Syst Softw 138:158–173. ISSN: 0164-1212