# Assessing Architecture Conformance to Coupling-Related Infrastructure-as-Code Best Practices: Metrics and Case Studies

Evangelos Ntentos[1], Uwe Zdun[1], Jacopo Soldani[2], and Antonio Brogi[2]

[1] University of Vienna, Faculty of Computer Science, Research Group Software Architecture, Austria
firstname.lastname@univie.ac.at
[2] University of Pisa, Faculty of Computer Science, Italy
firstname.lastname@unipi.it

**Abstract.** Infrastructure as Code (IaC) is an IT practice that facilitates the management of the underlying infrastructure as software. It enables developers or operations teams to automatically manage, monitor, and provision resources rather than organize them manually. In many industries, this practice is widespread and has already been fully adopted. However, few studies provide techniques for evaluating architectural conformance in IaC deployments and, in particular, aspects such as loose coupling. This paper focuses on coupling-related patterns and practices such as deployment strategies and the structuring of IaC elements. Many best practices are documented in gray literature sources, such as practitioner books, blogs, and public repositories. Still, there are no approaches yet to automatically check conformance with such best practices. We propose an approach based on generic, technology-independent metrics tied to typical architectural design decisions for IaC-based practices in microservice deployments to support architecting in the context of continuous delivery practices. We present three case studies based on open-source microservice architectures to validate our approach.

**Keywords:** Infrastructure as Code, metrics, software architecture, architecture conformance, IaC best practices

## 1 Introduction

Today, many microservice-based systems are being rapidly released, resulting in frequent changes not only in the system implementation but also in its infrastructure and deployment [7,13]. Furthermore, the number of infrastructure nodes that a system requires is increasing significantly [13] and the managing and structuring of these elements can have a significant impact on the development and deployment processes. *Infrastructure as Code* enables automating the provisioning and management of the infrastructure nodes through reusable scripts, rather than through manual processes [11]. IaC can ensure that a provisioned environment remains the same every time it is deployed in the same configuration, and configuration files contain infrastructure specifications making the process of editing and distributing configurations easier [11,1].

IaC can also contribute to improving consistency and ensuring loose coupling by separating the deployment artifacts according to the services' and teams' responsibilities. The deployment infrastructure can be structured using infrastructure stacks. An infrastructure stack is a collection of infrastructure elements/resources that are defined, provisioned, and updated as a unit [11]. A wrong structure can result in severe issues if coupling-related aspects are not considered. For instance, defining all the system deployment artifacts as only one unit in one infrastructure stack can significantly impact the dependencies of system parts and teams as well as the independent deployability of system services. Most of the established practices in the industry are mainly reported in the so-called "grey literature," consisting of practitioner blogs, system documentation, etc. The architectural knowledge is scattered across many knowledge sources that are usually based on personal experiences, inconsistent, and incomplete. This creates considerable uncertainty and risk in architecting microservice deployments.

In this work, we investigate such IaC-based best practices in microservice deployments. In this context, we formulate a number of coupling-related *Architectural Design Decisions (ADDs)* with corresponding decision options. In particular, the ADDs focus on *System Coupling through Deployment Strategy* and *System Coupling through Infrastructure Stack Grouping*. For each of these, we define a number of generic, technology-independent metrics to measure the conformance of a given deployment model to the (chosen) ADD options. Based on this architectural knowledge, our goal is to provide an automatic assessment of architecture conformance to these practices in IaC deployment models. We also aim for a continuous assessment, i.e., we envision an impact on continuous delivery practices, in which the metrics are assessed with each delivery pipeline run, indicating improvement, stability, or deterioration in microservice deployments. In order to validate the applicability of our approach and the performance of the metrics, we conducted three case studies on open source microservice-based systems that also include the IaC-related scripts. The results show that our set of metrics is able to measure the support of patterns and practices.

This paper aims to answer the following research questions:

- **RQ1** How can we measure conformance to coupling-related IaC best practices in the context of IaC architecture decision options?
- **RQ2** What is a set of minimal elements needed in an IaC-based deployment and microservice architecture model to compute such measures?

This paper is structured as follows: Section 2 discusses related work. Next, we describe the research methods and the tools we have applied in our study in Section 3. In Section 4 we explain the ADDs considered in this paper and the related patterns and practices. Section 5 introduces our metrics in a formal model. Then, three case studies are explained in Section 6. Section 7 discusses the RQs regarding the evaluation results and analyses the threats to validity of our study. Finally, in Section 8 we draw conclusions and discuss future work.

## 2 Related Work

Several existing works target collecting IaC bad and best practices. For instance, Sharma et al. [17] present a catalog of design and implementation language-specific

smells for Puppet. A broad catalog of language-agnostic and language-specific best and bad practices related to implementation issues, design issues, and violations of essential IaC principles is presented by Kumara et al. [9]. Schwarz et al. [16] offer a catalog of smells for Chef. Morris [11] presents a collection of guidance on managing IaC. In his book, there is a detailed description of technologies related to IaC-based practices and a broad catalog of patterns and practices. Our work also follows IaC-specific recommendations given by Morris [11], as well as those more microservice-oriented given by Richardson [15]. We indeed build on their guidelines and catalogs of bad/best practices to support architecting the deployment of microservices, while also enabling us to assess and improve the quality of obtained IaC deployment models.

In this perspective, it is worth relating our proposal with existing tools and metrics for assessing and improving the quality of IaC deployment models. Dalla Palma et al. [4,5] suggest a catalog of 46 quality metrics focusing on Ansible scripts to identify IaC-related properties and show how to use them in analyzing IaC scripts. A tool-based approach for detecting smells in TOSCA models is proposed by Kumara et al. [10]. Sotiropoulos et al. [18] provide a tool to identify dependency-related issues by analyzing Puppet manifests and their system call trace. Van der Bent et al. [2] define metrics reflecting IaC best practices to assess Puppet code quality. All such approaches focus on the use of different metrics to assess and improve the quality of IaC deployment models, showing the potential and effectiveness of metrics in doing so. We hence follow a similar, metrics-based approach but targeting a different aspect than those of the above mentioned approaches, namely *system coupling*. To the best of our knowledge, ours is the first solution considering and tackling such aspects.

Other approaches worth mentioning are those by Fischer et al. [6] and Krieger et al. [8], who both allow automatically checking the conformance of declarative deployment models during design time. They both model conformance rules in the form of a pair of deployment model fragments. One of the fragments represents a detector subgraph that determines whether the rule applies to a particular deployment model or not. The comparison of the model fragments with a given deployment model is done by subgraph isomorphism. Unlike our study, this approach is generic and does not introduce specific conformance rules, such as checking coupling-related ADDs in IaC models.

Finally, it is worth mentioning that architecture conformance can also be checked with other techniques such as dependency-structure matrices, source code query languages, and reflexion models as shown by Passos et al. [14]. So far, methods based on various interrelated IaC-based metrics to check pattern/best practice conformance like ours do not yet exist. Also, none are able to produce assessments that combine different assessment parameters (i.e., metrics). Such metrics, if automatically computed, can be used as a part of larger assessment models during development and deployment time.

## 3 Research and Modeling Methods

Figure 1 shows the research steps followed in this study. We first studied knowledge sources related to IaC-specific best practices from practitioner books and blogs, and the scientific literature (such as [9,11,15,16,17]) as well as open-source repositories (such as the case studies discussed in Section 6). We then analyzed the data collected using

qualitative methods based on Grounded Theory [3] coding methods, such as open and axial coding, and extracted the two core IaC decisions described in Section 4 along with their corresponding decision drivers and impacts. We followed the widely used pattern catalogs by Morris [11] and Richardson [15] closely to obtain the necessary information since both are well documented, detailed, and address many relevant concerns in the IaC domain. Among the many design decisions, covered in these catalogs, we selected those that are directly connected to IaC practices, operate at a high abstraction level (i.e., they are "architectural" design decisions), and are related to architectural coupling issues. We then defined a set of metrics for automatically computing conformance to each coupling-related pattern or practice per decision described in Section 4. We studied and modeled three case studies following the *Model Generation* process. Finally, we evaluated our set of metrics using the case studies. Furthermore, in our work [12], we have introduced a set of detectors to automatically reconstruct an architecture component model from the source code. Combining the automatic reconstruction with the automatic computation of metrics, the evaluation process can be fully automated.

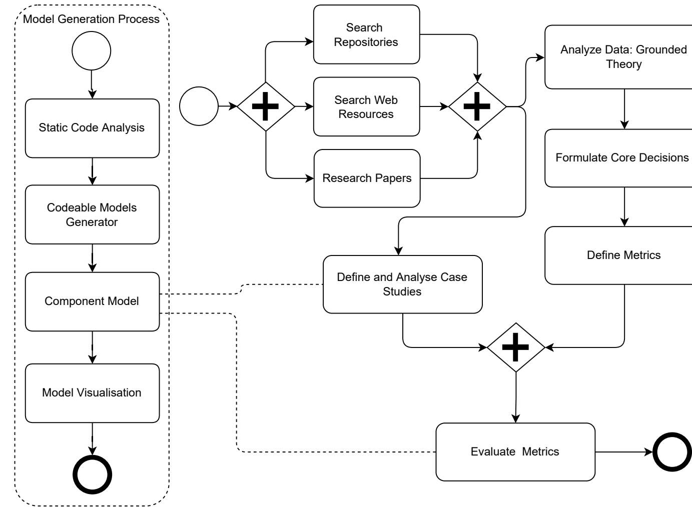


**Fig. 1.** Overview of the research method followed in this study

The systems we use as case studies were developed by practitioners with relevant experience and are supported by the companies Microsoft, Instana, and Weaveworks as microservice reference applications, which justifies the assumption that they provide a good representation of recent microservice and IaC practices. We performed a fully manual static code analysis for the IaC models that are in the repositories together with the application source code. To create the models, we used our existing modeling tool Codeable Models. The result is a set of decomposition models of the software

systems along with their deployments (see Section 5.1). The code and models used in and produced as part of this study have been made available online for reproducibility [3].

Figure 2 shows an excerpt of the resulting model of *Case Study 1* in Section 6. The model contains elements from both application (e.g., *Service, Database*) and infrastructure (e.g., *Container, Infrastructure Stack, Storage Resources*). Furthermore, we have specified all the deployment-related relationships between these elements. In particular, a *Service* and a *Web Server* are *deployed on* a *Container*. A *Database* is *deployed on Storage Resources* and also on a separate *Container*. An *Infrastructure Stack defines deployment of* a *Container* as well as a *Web Server*. All the containers *run on* a *Cloud Server* (e.g., ELK, AWS, etc.).

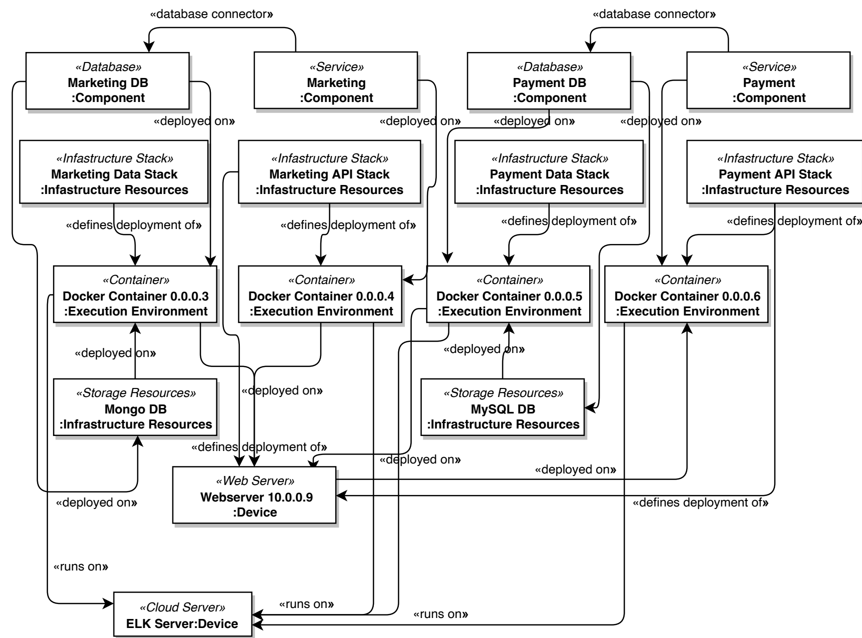


**Fig. 2.** Excerpt of the reconstructed model CS1 from Table 1

## 4 Decisions on Coupling-related, IaC-Specific Practices

In this section, we briefly introduce the two coupling-related ADDs along with their decision options which we study in this paper. In one decision, we investigate the deployment strategy between services and execution environments, and in the second, we focus on the structure of all deployment artifacts.

[3] https://doi.org/10.5281/zenodo.6696130

**System Coupling through Deployment Strategy Decision.** An essential aspect of deploying a microservice-based system is to keep the services independently deployable and scalable and ensure loose coupling in the deployment strategy. Services should be isolated from one another, and the corresponding development teams should be able to build and deploy a service as quickly as possible. Furthermore, resource consumption per service is another factor that should be considered, since some services might have constraints on CPU or memory consumption [15]. Availability and behavior monitoring are additional factors for each independent service that should be ensured in a deployment. One option, which hurts the *loose coupling* of the deployment, is *Multiple Services per Execution Environment* [4] In this pattern, services are all deployed in the same execution environment making it problematic to change, build, and deploy the services independently. A second option for service deployment is *Single Service per Execution Environment* [15]. This option ensures loose coupling in deployment since each service is independently deployable and scalable, and resource consumption can be constrained for each service and monitoring services separately. Development team dependencies are also reduced. Although *Single Service per Execution Environment* reduces coupling significantly, an incorrect structure of the system artifacts can introduce additional coupling in deployment even if all services are deployed on separate execution environments. The following decision describes in detail the structuring practices.

**System Coupling through Infrastructure Stack Grouping Decision.** Managing the infrastructure resources can impact significant architectural qualities of a microservice-based system, such as loose coupling between services and independent development in different teams. Grouping of different resources into infrastructure stacks should reflect the development teams' responsibilities to ensure independent deployability and scalability. An infrastructure stack may include different resources such as *Compute Resources* (e.g., VMs, physical servers, containers, etc.) and *Storage Resources* (e.g., block storage (virtual disk volumes), structured data storage, object storage, etc.) [11]. An important decision in infrastructure design is to set the size and structure of a stack. There are several patterns and practices on how to group the infrastructure resources into one or multiple stacks. A pattern that is useful when a system is small and simple is the *Monolith Stack* [11]. This pattern facilitates the process of adding new elements to a system as well as stack management. However, there are some risks to using this pattern. The process of changing a larger monolith stack is riskier than changing a smaller one, resulting in more frequent changes. Also, services cannot be deployed and changed independently and different development teams may be dependent on each other [11]. A similar pattern is the *Application Group Stack* [11]. This kind of stack includes all the services of multiple related applications. This pattern is appropriate when there is a single development team for the infrastructure and deployment of these services and has similar consequences as the *Monolith Stack* pattern.

[4] The term *Execution Environment* is used here to denote the environment in which a service runs such as a VM, a Container, or a Host. Please note that execution environments can be nested. For instance, a VM can be part of a Production Environment which in turn runs on a Public Cloud Environment. Execution environments run on Devices (e.g., Cloud Server).

A structuring that can work better with microservice-based systems is the *Service Stack* [11]. In this pattern, each service has its own stack which makes it easier to manage. Stacks boundaries are aligned to the system and team boundaries. Thus, teams and services are more autonomous and can work independently. Furthermore, services and their infrastructure are loosely coupled since independent deployability and scalability for each service are supported. The pattern *Micro Stack* [11] goes one step further by breaking the *Service Stack* into even smaller pieces and creates stacks for each infrastructure element in a service (e.g., router, server, database, etc.). This is beneficial when different parts of a service's infrastructure need to change at different rates. For instance, servers have different life cycles than data and it might work better to have them in separate stacks. However, having many small stacks to manage can add complexity and make it difficult to handle the integration between multiple stacks [11].

## 5 Metrics Definition

In this section, we describe metrics for checking conformance to each of the decision options described in Section 4.

### 5.1 Model Elements Definition

In this paper, we use and extend a formal model for metrics definition based on our prior work [19]. We extend it here to model the integration of component and deployment nodes. A microservice decomposition and deployment architecture model $M$ is a tuple $(N_M, C_M, NT_M, CT_M, c_source, c_target, nm_connectors, n_type, c_type)$ where:

- $N_M$ is a finite set of component and infrastructure **nodes** in Model $M$.
- $C_M \subseteq N_M \times N_M$ is an ordered finite set of **connector edges**.
- $NT_M$ is a set of **component types**.
- $CT_M$ is a set of **connector types**.
- $c_source : C_M \rightarrow N_M$ is a function returning the component that is the **source** of a link between two nodes.
- $c_target : C_M \rightarrow N_M$ is a function returning the component that is the **target** of a link between two nodes.
- $nm_connectors : \mathbb{P}(N_M) \rightarrow \mathbb{P}(C_M)$ is a function returning the set of connectors for a set of nodes: $nm_connectors(nm) = \{c \in C_M : (\exists n \in nm : (c_source(c) = n \land c_target(c) \in C_M) \lor (c_target(c) = n \land c_source(c) \in C_M))\}$.
- $n_type : N_M \rightarrow \mathbb{P}(NT_M)$ is a function that maps each node to its set of **direct and transitive node types**. (for a formal definition of node types see [19]).
- $c_type : C_M \rightarrow \mathbb{P}(CT_M)$ is a function that maps each connector to its set of **direct and transitive connector types**. (for a formal definition of connector types see [19]).

All deployment nodes are of type *Deployment_Node*, which has the subtypes *Execution_Environment* and *Device*. These have further subtypes, such as *VM* and *Container*

for *Execution_Environment*, and *Server*, *IoT Device*, *Cloud*, etc. for *Device*. Environments can also be used to distinguish logical environments on the same kind of infrastructure, such as a *Test_Environment* and a *Production_Environment*. All types can be combined, e.g. a combination of *Production_Environment* and *VM* is possible.

The microservice decomposition is modeled as nodes of type *Component* with component types such as *Service* and connector types such as *RESTful HTTP*.

The connector type *deployed_on* is used to denote a deployment relation of a *Component* (as a connector source) on an *Execution_Environment* (as a connector target). It is also used to denote the transitive deployment relation of *Execution_Environments* on other ones, such as a *Container* is deployed on a *VM* or a *Test_Environment*. The connector type *runs_on* is used to model the relations between execution environments and the devices they run on.

The type *Stack* is used to define deployments of *Devices* using the *defines_deployment_of* relation. Stacks include environments with their deployed components using the *includes_deployment_node* relation.

### 5.2 Metrics for System Coupling through Deployment Strategy Decision

The *System Coupling through Deployment Strategy* related metrics, introduced here, each have a continuous value with range from 0 to 1, with 0 representing the optimal case where the coupling is minimized by applying the recommended IaC best practices.

**Shared Execution Environment Connectors Metric (SEEC)**. This metric $SEEC : \mathbb{P}(C_M) \rightarrow [0,1]$ returns the number of the *shared* direct connectors from deployed service components to execution environments (e.g., containers or VMs) in relation to the total number of such service to environment connectors. For instance, the connectors of two services that are deployed on the same container are considered as shared. This gives us the proportion of the shared execution environment connectors in the system. In this context, let the function $service_env_connectors : \mathbb{P}(C_M) \rightarrow \mathbb{P}(C_M)$ return the set of all connectors between deployed services and their execution environments: $service_env_connectors(cm) = \{c \in cm : Service \in n_type(c_source(c)) \wedge Execution_Environment \in n_type(c_target(c)) \wedge deployed_on \in c_type(c)\}$. Further, let the function $shared_service_env_connectors : \mathbb{P}(C_M) \rightarrow \mathbb{P}(C_M)$ return the set of connectors from multiple components to the same execution environment: $shared_service_env_connectors(cm) = \{c1 \in service_env_connectors(cm) : \exists\, c2 \in C_{EE} : c_source(c1) \neq c_source(c2) \wedge c_target(c1) = c_target(c2)\}$. Then SEEC can be defined as:

$$SEEC(cm) = \frac{|shared_service_env_connectors(cm)|}{|service_env_connectors(cm)|}$$

**Shared Execution Environment Metric (SEE)**. The metric $SEE : \mathbb{P}(N_M) \rightarrow [0,1]$ measures the shared execution environments that have service components deployed on them (e.g., a container/VM that two or more services are deployed on) in relation to all executions environments with deployed services:

$$SEE(nm) = \frac{|\{n \in nm : (\exists\, c \in nm_connectors(nm) : c \in shared_service_env_connectors(cm) \wedge c_target(c) = n)\}|}{|\{n \in nm : (\exists\, c \in nm_connectors(nm) : c \in service_env_connectors(cm) \wedge c_target(c) = n)\}|}$$

### 5.3 Metrics for System Coupling through Infrastructure Stack Grouping Decision

The metrics for *System Coupling through Infrastructure Stack Grouping* decision are return boolean values as they detect the presence of a decision option. Please note that the boolean metrics are defined for arbitrary node sets, i.e. they can be applied to any subset of a model to determine sub-models in which a particular practice is applied.

For the metrics below, let the function $services : \mathbb{P}(N_M) \rightarrow \mathbb{P}(N_M)$ return the set of services in a node set: $services(nm) = \{n \in nm : Service \in n_type(n)\}$. Further, let the function $stack_deployed_envs : N_M \times \mathbb{P}(N_M) \rightarrow \mathbb{P}(N_M)$ return environments included in a Stack $s$ with $stack_included_envs(s, nm) = \{e \in nm : (\exists\, c \in nm_connectors(cm) : Stack \in n_type(s) \wedge c_source(c) = s \wedge c_target(c) = e \wedge includes_deployment_node \in c_type(c))\}$. Let the function $stack_deployed_components : N_M \times \mathbb{P}(N_M) \rightarrow \mathbb{P}(N_M)$ return the components deployed via an environment by a Stack $s$ with $stack_deployed_components(s, nm) = \{n \in nm : (\exists\, c \in nm_connectors(cm) : Component \in n_type(c_source(c)) \wedge c_target(c) \in stack_included_envs(s) \wedge deployed_on \in c_type(c))\}$. With this, $stacks_deploying_services : \mathbb{P}(N_M) \rightarrow \mathbb{P}(N_M)$ can be defined, which returns all stacks that deploy at least one service: $stacks_deploying_services(nm) = \{s \in nm : Stack \in n_type(s) \wedge (n \in services(nm) : n \in stack_deployed_components(s))\}$ Finally, we can define $stacks_deploying_non_service_components(nm) : \mathbb{P}(N_M) \rightarrow \mathbb{P}(N_M)$ as $stacks_deploying_non_service_components(nm) = \{s \in nm : Stack \in n_type(s) \wedge (n \in nm : n \in stack_deployed_components(s) \wedge n \notin services(nm))\}$.

**Monolithic Stack Detection Metric (MSD).** The metric $MSD : \mathbb{P}(N_M) \rightarrow Boolean$ returns $True$ if only one stack is used in a node set that deploys more than one service via stacks (e.g., a number of/all system services are deployed by the only defined stack in the infrastructure) and $False$ otherwise.

$$MSD(nm) = \begin{cases} True : & \text{if } |stacks_deploying_services(nm)| = 1 \\ False : & otherwise \end{cases}$$

**Application Group Stack Detection Metric (AGSD).** The metric $AGSD : \mathbb{P}(C_M) \rightarrow Boolean$ returns $True$ if multiple stacks are used in a node set to deploy services and more services are deployed via stacks than there are stacks (e.g., system services are deployed by one stack and other elements such as routes are deployed by different stack(s)). That is, multiple services are clustered in groups on at least one of the stacks.

$$AGSD(nm) = \begin{cases} True: & \text{if } |stacks_deploying_services(nm)| > 1 \wedge \\ & \quad |stacks_deploying_services(nm)| < |services(nm)| \\ False: & otherwise \end{cases}$$

**Service-Stack Detection Metric (SES).** The metric $SES : \mathbb{P}(N_M) \rightarrow Boolean$ returns $True$ is the number of services deployed by stacks equals the number of stacks (e.g., each system service is deployed by its own stack).

$$SES(nm) = \begin{cases} True: & \text{if } |stacks_deploying_services(nm)| = |services(nm)| \\ False: & otherwise \end{cases}$$

**Micro-Stack Detection Metric (MST).** The metric $MST : \mathbb{P}(C_M) \rightarrow Boolean$ returns $True$ if SES is $True$ and also there is one or more stacks that deploy non-service components (e.g., databases, monitoring components, etc.). For instance, a service is deployed by one stack and its database is deployed by another stack as well as a router is deployed by its own stack, etc.:

$$MST(nm) = \begin{cases} True: & \text{if } SES(nm) = True \wedge \\ & \quad |stacks_deploying_non_service_components(nm)| > 0 \\ False: & otherwise \end{cases}$$

**Services per Stack Metric (SPS).** To measure how many services on average are deployed by a service-deploying stack, we define the metrics $SPS : \mathbb{P}(C_M) \rightarrow \mathbb{R}$ as:

$$SPS(nm) = \frac{\begin{array}{r} |\{n \in services(nm) : (\exists s \in nm : Stack \in n_type(s) \wedge \\ n \in stack_deployed_components(s))\}| \end{array}}{|stacks_deploying_services(nm)|}$$

**Components per Stack Metric (CPS).** To measure how many components on average are deployed by a component-deploying stack, we define the metrics $CPS : \mathbb{P}(C_M) \rightarrow \mathbb{R}$ as:

$$CPS(nm) = \frac{\begin{array}{r} |\{n \in nm : (\exists s \in nm : Stack \in n_type(s) \wedge \\ n \in stack_deployed_components(s))\}| \end{array}}{\begin{array}{l} |stacks_deploying_services(nm) \cup \\ \quad stacks_deploying_non_service_components(nm)| \end{array}}$$

## 6 Case Studies

In this section, we describe the case studies used to evaluate our approach and test the performance of the metrics. We studied three open-source microservice-based systems. We also created variants that introduce typical violations of the ADDs described in the literature or refactorings to improve ADD realization to test how well our metrics help to spot these issues and improvements. The cases are summarized in Table 1 and metrics results are presented in Table 2.

| *Case Study ID* | *Model Size* | *Description / Source* |
| --- | --- | --- |
| CS1 | 68 components 167 connectors | E-shop application using pub/sub communication for event-based interaction as well as files for deployment on a Kubernetes cluster. All services are deployed in their own infrastructure stack (from `https://github.com/dotnet-architecture/eShopOnContainers`). |
| CS1.V1 | 67 components 163 connectors | Variant of Case Study 1 in which half of the services are deployed on the same execution environment and some infrastructure stacks deploy more than one service. |
| CS1.V2 | 60 components 150 connectors | Variant of Case Study 1 in which some services are deployed on the same execution environment and half of the non-services components are deployed by a component-deploying stack. |
| CS2 | 38 components 95 connectors | An online shop that demonstrate and test microservice and cloud-native technologies and uses a single infrastructure stack to deploy all the elements (from `https://github.com/microservices-demo/microservices-demo`). |
| CS2.V1 | 40 components 101 connectors | Variant of Case Study 2 where multiple infrastructure stacks are used to deploy the system elements as well as some services are deployed on the same execution environment. |
| CS2.V2 | 36 components 88 connectors | Variant of Case Study 2 where two infrastructure stacks are used to deploy the system elements (one for the services and one for the rest elements) as well as some services are deployed on the same execution environment. |
| CS3 | 32 components 118 connectors | Robot shop application with various kinds of service interconnections, data stores, and Instana tracing on most services as well as an infrastructure stack that deploys the services and their related elements (from `https://github.com/instana/robot-shop`). |
| CS3.V1 | 56 components 147 connectors | Variant of Case Study 3 where some services are deployed in their own infrastructure stack as well as some services are deployed on the same execution environment. |
| CS3.V2 | 56 components 147 connectors | Variant of Case Study 3 where all services are deployed in their own infrastructure stack as well as all services are deployed on their own execution environment. |

**Table 1.** Overview of modeled case studies and the variants (size, details, and sources)

**Case Study 1: eShopOnContainers Application.** The *eShopOnContainers* case study is a sample reference application realized by Microsoft, based on a microservices architecture and Docker containers, to be run on Azure and Azure cloud services. It contains multiple autonomous microservices, supports different communication styles (e.g., synchronous, asynchronous via a message broker). Furthermore, the application contains the files required for deployment on a Kubernetes cluster and provides the necessary IaC scripts to work with ELK for logging (Elasticsearch, Logstash, Kibana).

To investigate further, we performed a full manual reconstruction of an architecture component model and an IaC-based deployment model of the application as ground truth for the case study. Figure 2 shows the excerpt component model specifying the component types (e.g., *Services, Facades, and Databases*), and connector types (e.g., *database connectors*, etc.) as well as all the IaC-based deployment component types (e.g., *Web Server, Cloud Server, Container, Infrastructure Stack, Storage Resources*, etc.) and IaC-based deployment connector types (e.g., *defines deployment of, deployed on*, etc.) using types as introduced in Section 5 shown here as stereotypes.

The component model, consists of in total 235 elements such as component types, connector types, IaC-based deployment component types and IaC-based deployment connector types. More specifically, *19 Infrastructure Stacks*, *19 Execution Environments*, *6 Storage Resources*, *7 Services*, *6 Databases*, *19 Stack-to-Execution Environment connectors*, *7 Stack-to-Service connectors* and *6 Storage Resources-to-Database connectors*. There are also other 146 elements in the application (e.g., *Web Server*, *Cloud Server*, *Stack-to-Web Server connectors*, etc.).

The values of metrics *SEEC* and *SEE* show that *Single Service per Execution Environment* pattern is fully supported. We treat both components and connectors as equally essential elements; thus, we use two metrics to assess coupling in our models. Given that *SEE* returns the shared execution environments, it is crucial to measure how strongly these environments are shared. The *SEEC* value indicates this specific aspect by returning the proportion of the shared connectors that these environments have.

The application uses multiple stacks to deploy the services and the other elements, and this is shown by the outcomes of the metrics for the *System Coupling through Infrastructure Stack Grouping* decision. Since multiple stacks have been detected the metrics *MSD* and *AGSD* return *False*. The *SES* metric returns *True* meaning that the *Service Stack* pattern is used. The *MST* returns *True* which means the *Micro-Stack* pattern for the node sub-set *Storage Resources* is also used. The *SPS* value shows that every service is deployed by a service-deploying stack. Furthermore, *CPS* also shows that components that belong to node sub-set *Storage Resources* are deployed by a component-deploying stack. Overall the metrics results in this case study show no coupling issue in deployment, and all best practices in our ADDs have been followed.

For further evaluation, we created two variants to test our metrics' performance in more problematic cases. Our analysis in *CS1.V1* shows that half of the execution environments are shared, and around two-thirds of the connectors between services and execution environments are also considered as shared, meaning these execution environments are strongly coupled with the system services. Using both values, we have a more complete picture of the coupling for all essential elements in this model. Furthermore, the *SPS* value indicates that the *Service Stack* pattern is partially supported, meaning some services are grouped in the same stacks. The analysis in *CS1.V2* shows that our metrics can measure all the additional violations that have been introduced.

**Case Study 2: Sock Shop Application.** The *Sock Shop* is a reference application for microservices by the company Weaveworks to illustrate microservices architectures and the company's technologies. The application demonstrates microservice and cloud-native technologies. The system uses *Kubernetes* for container-orchestration and services are deployed on *Docker* containers. *Terraform* infrastructure scripts are provided to deploy the system on *Amazon Web Services (AWS)*. We believe it to be a good representative example of the current industry practices in microservice-based architectures and IaC-based deployments.

The reconstructed model of this application contains in total 133 elements. In particular, *1 Infrastructure Stack*, *13 Execution Environments*, *3 Storage Resources*, *7 Services*, *4 Databases*, *13 Stack-to-Execution Environment connectors*, *7 Stack-to-Service connectors* and *4 Storage Resources-to-Database connectors*. There are also another 74 elements in the application such as *Web Server*, *Cloud Server*, *Stack-to-Web Server connectors* and *Execution Environment-to-Cloud Server connectors*.

We have tested our metrics to assess the conformance to best patterns and practices in IaC-based deployment. The outcome of the metrics related to *System Coupling through Deployment Strategy* decision shows that also this application fully supports the *Single Service per Execution Environment* pattern. That is, all services are deployed in separate execution environments. Regarding the *System Coupling through Infrastructure Stack Grouping* decision, we detected the *Monolith Stack* pattern, which means one

stack defines the deployment of all system elements, resulting in a highly coupled deployment. The metrics *AGSD*, *SES*, and *MST* are all *False*, and *SPS* and *CPS* return 0, since a monolith stack has been detected. These values can guide architects to improve the application by restructuring the infrastructure to achieve the desired design.

In our variants, we introduced gradual but not perfect improvements. The metrics results for *CS2.V1* show an improvement compared to the initial version. That is, *Monolith Stack* pattern is not used since three infrastructure stacks have been detected, and some services are deployed by service-deploying stacks. In *CS2.V2* there is a slight improvement since *Application Group Stacks* has been detected. However, *SEEC* and *SEE* metrics indicate that there is a strong coupling between services and execution environments. In both variants, the metrics have well detected the improvements made.

**Case Study 3: Robot-Shop Application.** *Robot-Shop* is a reference application by the company Instana provides to demonstrate polyglot microservice architectures and Instana monitoring. It includes the necessary IaC scripts for deployment. All system services are deployed on *Docker* containers and use *Kubernetes* for container-orchestration. Moreover, *Helm* is also supported for automating the creation, packaging, configuration, and deployment to *Kubernetes* clusters. End-to-end monitoring is provided, and some services support *Prometheus* metrics.

The reconstructed model of this application contains in total 150 elements. In particular, *2 Infrastructure Stacks*, *18 Execution Environments*, *2 Storage Resources*, *10 Services*, *3 Databases*, *13 Stack-to-Execution Environment connectors*, *10 Stack-to-Service connectors* and *3 Storage Resources-to-Database connectors*. There are also 89 additional elements in the application.

The metrics results for the *System Coupling through Deployment Strategy* decision are both optimal, showing that in this application all services are deployed in separate execution environments. For the *System Coupling through Infrastructure Stack Structuring* decision, the *AGSD* metric return *True* which means that the *Application Group Stack* pattern is used, resulting in highly coupled services' deployment. Thus, the metrics *SES* and *MST* are *False* and *SPS* and *CPS* return 0. According to these values, architects can be supported to address the detected violations (e.g., as done in *CS3.V2*).

In the variants, we introduced one gradual improvement first and then a variant that addresses all issues. Our analysis in *CS3.V1* shows significant improvement in infrastructure stack grouping. Most of the services are deployed on their own stack, and components that belong to node sub-set *Storage Resources* are completely deployed by a separate stack. However, coupling between services and execution environments has also been detected. Variant *CS3.V2* is even more improved since, in this case, all services are deployed by service-deploying stacks, and no coupling has been detected. In both variants, the metrics have faithfully identified the changes made.

## 7 Discussion

**Discussion of Research Questions.** To answer **RQ1**, we proposed a set of generic, technology-independent metrics for each IaC decision, and each decision option corresponds to at least one metric. We defined a set of generic, technology-independent

**Table 2.** Metrics Calculation Results

| Metrics | CS1 | CS1.V1 | CS1.V2 | CS2 | CS2.V1 | CS2.V2 | CS3 | CS3.V1 | CS3.V2 |
|---|---|---|---|---|---|---|---|---|---|
| **System Coupling through Deployment Strategy** | | | | | | | | | |
| **SEEC** | 0.00 | 0.71 | 0.42 | 0.00 | 0.25 | 0.62 | 0.00 | 0.37 | 0.00 |
| **SEE** | 0.00 | 0.50 | 0.20 | 0.00 | 0.14 | 0.40 | 0.00 | 0.16 | 0.00 |
| **System Coupling through Infrastructure Stack Grouping** | | | | | | | | | |
| **MSD** | False | False | False | True | False | False | False | False | False |
| **AGSD** | False | False | False | False | False | True | True | False | False |
| **SES** | True | False | False | False | False | False | False | False | True |
| **MST** | True | True | False | False | False | False | False | False | True |
| **SPS** | 1.00 | 0.20 | 0.57 | 0.00 | 0.12 | 0.00 | 0.00 | 0.62 | 1.00 |
| **CPS** | 1.00 | 1.00 | 0.50 | 0.00 | 0.00 | 0.00 | 0.00 | 1.00 | 1.00 |

metrics to assess each pattern's implementation in each model automatically and conducted three case studies to test the performance of these metrics. For assessing pattern conformance, we use both numerical and boolean values. In particular, *SEEC* and *SEE* measures return a range from 0 to 1, with 0 representing the optimal case where a set of patterns is fully supported. Having this proportion, we can assess not only the existence of coupling but also how severe the problem is. However, this is not the case for the *MSD*, *AGSD*, *SES*, and *MST* metrics that return *True* or *False*. Using these metrics, we intend to detect the presence of the corresponding patterns. For *Service Stack* and *Micro Stack* decision options, we introduce two additional metrics with numerical values that can be applied on node subsets to assess the level of the pattern support when patterns are not fully supported. However, applying them on node subsets has the limitation that many runs need to be made, leading to ambiguous results. Our case studies' analysis shows that every set of decision-related metrics can detect and assess the presence and the proportion of pattern utilization.

Regarding **RQ2**, we can assess that our deployment meta-model has no need for significant extensions and is easy to map to existing modeling practices. More specifically, to fully model the case studies and the additional variants, we needed to introduce 13 device type nodes and 11 execution environment nodes types such as *Cloud Server* and *Virtual Machine* respectively, and 9 deployment relation types and 7 deployment node relations. Furthermore, we also introduced a deployment node meta-model to cover all the additional nodes of our decisions, such as *Storage Resources*. The decisions in *System Coupling through Deployment Strategy* require modeling several elements such as the *Web Server*, *Container*, and *Cloud Server* nodes types and technology-related connector types (e.g. *deployed on*) as well as deployment-related connector types (e.g. *Runs on, Deployed in Container*). For the *Coupling through Infrastructure Stack Grouping* decision, we have introduced attributes in the system nodes (e.g., in *Infrastructure Stack, Storage Resources*) and connector types (e.g., *defines deployment of, includes*).

**Threats to Validity.** We mainly relied on third-party systems as the basis for our study to increase internal validity and thus avoid bias in system composition and structure. It is possible that our search procedures resulted in some unconscious exclusion of specific sources; we mitigated this by assembling a team of authors with many years of

experience in the field and conducting a very general and broad search. Because our search was not exhaustive and the systems we found were created for demonstration purposes, i.e., were relatively modest in size, some potential architectural elements were not included in our metamodel. Furthermore, this poses a potential threat to the external validity of generalization to other, more complex systems. However, we considered widely accepted benchmarks of microservice-based application as reference applications, in a way to reduce this possibility. Another potential risk is that the system variants were developed by the author team itself. However, this was done following best practices documented in the literature. We were careful to change only certain aspects in a variant and keep all other aspects stable. Another possible source of internal validity impairment is the modeling process. The author team has considerable experience with similar methods, and the systems' models have been repeatedly and independently cross-checked, but the possibility of some interpretive bias remains. Other researchers may have coded or modeled differently, resulting in different models. Because our goal was only to find a model that could describe all observed phenomena, and this was achieved, we do not consider this risk to be particularly problematic for our study. The metrics used to assess the presence of each pattern were deliberately kept as simple as possible to avoid false positives and allow for a technology-independent assessment.

## 8 Conclusions and Future Work

We have investigated the extent to which it is possible to develop a method to automatically evaluate coupling-related practices of ADDs in an IaC deployment model. Our approach models the critical aspects of the decision options with a minimal set of model elements, which means it is possible to extract them automatically from the IaC scripts. We then defined a set of metrics to cover all decision options described in Section 4 and used the case studies to test the performance of the generated metrics. Before, for the coupling aspects of IaC deployment models, no general, technology-independent metrics have been studied in depth. Our approach treats deployment architectures as a set of nodes and links, considering the technologies used, which were not supported in prior studies. The goal of our approach is a continuous evaluation, taking into account the impact of continuous delivery practices, in which metrics are evaluated continuously, indicating improvements and loose coupling of deployment architecture compliance.

In future work, we plan to study more decisions and related metrics, test further in larger systems, and integrate our approach in a systematic guidance tool.

**Acknowledgments.** This work was supported by: FWF (Austrian Science Fund) project IAC$^2$: I 4731-N.

## References

1. Artac, M., Borovssak, T., Di Nitto, E., Guerriero, M., Tamburri, D.A.: Devops: Introducing infrastructure-as-code. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). pp. 497–498 (2017)

2. van der Bent, E., Hage, J., Visser, J., Gousios, G.: How good is your puppet? an empirically defined and validated quality model for puppet. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 164–174 (2018). https://doi.org/10.1109/SANER.2018.8330206
3. Corbin, J., Strauss, A.L.: Grounded theory research: Procedures, canons, and evaluative criteria. Qualitative Sociology **13**, 3–20 (1990)
4. Dalla Palma, S., Di Nucci, D., Palomba, F., Tamburri, D.A.: Toward a catalog of software quality metrics for infrastructure code. Journal of Systems and Software **170**, 110726 (2020)
5. Dalla Palma, S., Di Nucci, D., Tamburri, D.A.: Ansiblemetrics: A python library for measuring infrastructure-as-code blueprints in ansible. SoftwareX **12**, 100633 (2020)
6. Fischer, M.P., Breitenbücher, U., Képes, K., Leymann, F.: Towards an approach for automatically checking compliance rules in deployment models. In: Proceedings of The Eleventh International Conference on Emerging Security Information, Systems and Technologies (SECURWARE). pp. 150–153. Xpert Publishing Services (XPS) (2017)
7. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Professional (2010)
8. Krieger, C., Breitenbücher, U., Képes, K., Leymann, F.: An Approach to Automatically Check the Compliance of Declarative Deployment Models. In: Papers from the 12th Advanced Summer School on Service-Oriented Computing (SummerSoC 2018). pp. 76–89. IBM Research Division (Oktober 2018)
9. Kumara, I., Garriga, M., Romeu, A.U., Di Nucci, D., Palomba, F., Tamburri, D.A., van den Heuvel, W.J.: The do's and don'ts of infrastructure code: A systematic gray literature review. Information and Software Technology **137**, 106593 (2021)
10. Kumara, I., Vasileiou, Z., Meditskos, G., Tamburri, D.A., Van Den Heuvel, W.J., Karakostas, A., Vrochidis, S., Kompatsiaris, I.: Towards semantic detection of smells in cloud infrastructure code. In: Proceedings of the 10th International Conference on Web Intelligence, Mining and Semantics. p. 63–67. WIMS 2020, Association for Computing Machinery, New York, NY, USA (2020)
11. Morris, K.: Infrastructure as Code: Dynamic Systems for the Cloud, vol. 2. O'Reilly (2020)
12. Ntentos, E., Zdun, U., Plakidas, K., Genfer, P., Geiger, S., Meixner, S., Hasselbring, W.: Detector-based component model abstraction for microservice-based systems. Computing **103**, 2521–2551 (August 2021)
13. Nygard, M.: Release It! Design and Deploy Production-Ready Software. Pragmatic Bookshelf (2007)
14. Passos, L., Terra, R., Valente, M.T., Diniz, R., das Chagas Mendonca, N.: Static architecture-conformance checking: An illustrative overview. IEEE software **27**(5), 82–89 (2010)
15. Richardson, C.: A pattern language for microservices. `http://microservices.io/patterns/index.html` (2017)
16. Schwarz, J., Steffens, A., Lichter, H.: Code smells in infrastructure as code. In: 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC). pp. 220–228 (2018). https://doi.org/10.1109/QUATIC.2018.00040
17. Sharma, T., Fragkoulis, M., Spinellis, D.: Does your configuration code smell? In: Proceedings of the 13th International Conference on Mining Software Repositories. p. 189–200. MSR '16, Association for Computing Machinery, New York, NY, USA (2016)
18. Sotiropoulos, T., Mitropoulos, D., Spinellis, D.: Practical fault detection in puppet programs. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. p. 26–37. ICSE '20, Association for Computing Machinery, New York, NY, USA (2020)
19. Zdun, U., Navarro, E., Leymann, F.: Ensuring and assessing architecture conformance to microservice decomposition patterns. In: Maximilien, M., Vallecillo, A., Wang, J., Oriol, M. (eds.) Service-Oriented Computing. pp. 411–429. Springer International Publishing, Cham (2017)