# Detecting and Resolving Coupling-Related Infrastructure as Code Based Architecture Smells in Microservice Deployments

Evangelos Ntentos, Uwe Zdun
*Faculty of Computer Science,*
*Research Group Software Architecture*
*University of Vienna*
Vienna, Austria
firstname.lastname@univie.ac.at

Ghareeb Falazi, Uwe Breitenbücher, Frank Leymann
*Institute of Architecture of Application Systems*
*University of Stuttgart*
Stuttgart, Germany
firstname.lastname@iaas.uni-stuttgart.de

***Abstract*—The Infrastructure as Code (IaC) concept enables IT infrastructure to be managed as software: resources can be managed, monitored, and provisioned automatically instead of manually by developers or operations teams. Many industries have already embraced this concept widely. However, research on IaC-based deployments, particularly research focusing on loose coupling, often does not offer methods for evaluating architectural conformance, spotting architecture smells, and support for correcting the found smells. Our work strives to provide an automatic method for continuously developing microservice-based systems and the associated infrastructure. We aim to offer an automated architectural refactoring method that checks if IaC-based deployments adhere to patterns and best practices and do not contain potential architectural smells. We provide architects with viable options for enhancing architectural conformance during microservice development. In short, by continuously detecting architectural smells and suggesting possible fixes, we aim to support architecture evolution within the framework of continuous delivery practices. We evaluate our approach using three case studies and variants based on open-source microservice architectures.**

***Index Terms*—*Infrastructure as code, metrics, architecture smells, modeling, best practices***

## I. Introduction

Microservice-based systems often support rapid release techniques, resulting in frequent infrastructure and deployment modifications. Additionally, the infrastructure components that a system needs are growing rapidly [1]. Managing and organizing these pieces often impacts the development and deployment processes. Infrastructure as Code (IaC) facilitates automated management and provisioning of infrastructure components [2]. By dividing deployment artifacts according to the duties of services and teams, IaC can also ensure that a deployed environment stays the same each time it is deployed in the same configuration [2], [3]. Furthermore, it can help with coherence and maintain loose coupling by separating deployment artifacts according to the responsibilities of services and teams. It helps keep the architecture diagrams and the actual deployment consistent.

There have been several architectural patterns and other "best practices" for microservice-based systems [4], [5], [6] as well as microservice deployments [2]. However, providing practical mechanisms to enforce such patterns and practices specifically for IaC-based deployments has received very little attention up to this point. This is troublesome because managing architecture compliance manually can be challenging, particularly in big complex architectures. Moreover, enhancing one best practice may cause problems with another since best practices are often interdependent. Thus, numerous additional system architectural and implementation constraints impact the architectures in ways that might result in unintentional or deliberate breaches of best practices for IaC-based deployments. In the context of DevOps and continuous delivery, it is anticipated that the architecture changes rapidly and often without central coordination. *Architecture smell* is a term used in software engineering to describe specific characteristics or traits of a software architecture that indicate a potential problem or suboptimal design [7]. These smells can arise due to various factors, such as poor modularization, lack of cohesion, high coupling, or inefficient use of design patterns.

We have observed that there are multiple factors contributing to the increasing complexity of architecture. If infrastructure as code technologies are utilized to deploy these architectures, there is a significant possibility that architectural issues may be embedded in the IaC models without the developers' immediate knowledge.

This study aims to provide actionable solutions to fix architectural smells of loose coupling-related IaC best practices. We focus on two major Architectural Design Decisions (ADD) in this scope, *System Coupling through Deployment Strategy* and *System Coupling through Infrastructure Stack Grouping*, that have been modeled based on an empirical study of existing best practices and patterns used by practitioners in our previous work [8].

We provide automated architecture refactoring tailored for architectural design in the context of IaC-related ADDs. We also employ the experimentally proven metrics suggested in our previous work [8]. These metrics allow us to analyze the degree to which an IaC deployment model adheres to preferred or less preferred design alternatives for each of the ADDs previously mentioned. We systematically specify each potential smell for every design option in the ADDs, and propose automated smell detection algorithms based on those specifications. Using the combination of available ADD options, the chosen option, potential smells, and detected smells, we can determine all possible next decision options

by applying solutions to the smells. This results in a search tree of models for the next architecture iteration, which we individually evaluate using our metrics. Based on this, we can assess the conformance of IaC-based deployment models regarding architectural patterns and potential refactorings and provide an architect with all potential improvements. The purpose of smell detection is to discover the precise locations in the models where the smells occur.

This method is also intended to be continually applicable across each run of a continuous delivery pipeline.

This paper aims to study the following research questions:

- **RQ1** What are the potential coupling-related architectural smells in IaC-based deployment models related to System Coupling through Deployment Strategy and System Coupling through Infrastructure Stack Grouping, and how can they be automatically detected?
- **RQ2** What are the possible fixes for the architectural smells in IaC-based deployment models related to System Coupling through Deployment Strategy and System Coupling through Infrastructure Stack Grouping, and how can architects be supported in correcting them?

In total, 12 IaC-based deployment models (three case studies and nine variations) based on microservice-based systems that practitioners developed are used to evaluate our approach (see Table I). We implemented automated smell detection and refactoring algorithms to detect potential smells and develop every solution that may be used to solve each smell. The improvements over the initial version are then measured using our metrics [8] on coupling aspects in IaC-based deployments. The results show that every smell can be addressed in no more than three refactoring steps, producing ideal metric values.

This paper is structured as follows: In Section II, we explain the decisions in the focus of this paper. We also explain related patterns and practices, as well as the corresponding metrics, as the background of our work. Section III discusses and compares to related work. Next, we describe the research methods and the tools we have applied in our study in Section IV. Then, three case studies are explained in Section V. We then describe the approach details in Section VI. In Section VII, we explain the evaluation process of our work. Section VIII discusses the RQs regarding the evaluation results. In Section IX, we then analyze the threats to validity. Finally, in Section X, we conclude and discuss future work.

## II. Background

This section will briefly discuss two coupling-related ADDs and their associated options. This information is based on our previous research [8], in which we conducted an empirical study to identify the IaC-related best practices and patterns currently used by practitioners. We also examined the potential decision drivers, or the factors that influence the decision-making process, and developed metrics to evaluate how well a given system model adheres to our decision model's recommended patterns and practices. By analyzing the reported outcomes of these decisions, we can determine which options are more or less popular among microservice practitioners. We

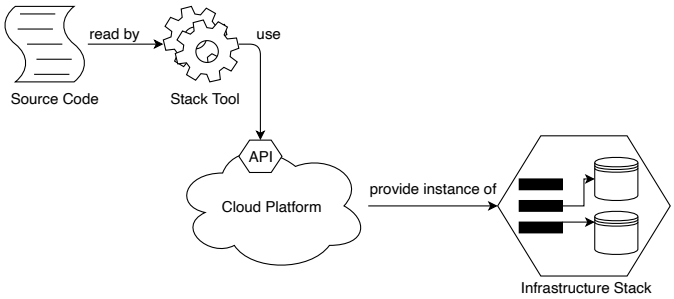


Fig. 1: The lifecycle of an infrastructure stack. this figure is adopted from Morris book [2]

employed 9 IaC-based component and deployment architecture models for evaluation, which are used and extended in this work, listed in Table I and explained in Section IV.

### A. Infrastructure Stack

Infrastructure stacks can be used to organize the deployment infrastructure, which refers to the set of hardware, software, and networking resources required to deploy and run applications, services, and systems in a production environment. According to [2], an infrastructure stack is *a group of infrastructure resources defined, provisioned, and updated collectively*. A non-optimal structure can harm the system if coupling-related factors are not considered. For instance, the dependencies of system parts and teams and the independent deployability of system services might be impacted by grouping all declarations of the system's infrastructure resources in only one infrastructure stack.

Figure 1 shows the lifecycle of an infrastructure stack. The resources and services that an infrastructure platform offers are the elements of a stack, and they are described by source code. For instance, a stack might consist of computing resources (e.g., a virtual machine), storage resources (e.g., disk volume), and network resources (e.g., a subnet) [2]. A stack management tool reads the source code for the stack and assembles the defined elements in the code to provision an instance of the infrastructure stack using a cloud platform's API [2].

### B. Architectural Design Decisions (ADDs)

*1) ADD 1: System Coupling through Deployment Strategy:* Maintaining the services' independence, scalability, and loose coupling is crucial when implementing a microservice-based system. The corresponding development teams should be able to construct and deploy a service swiftly, and services should be segregated. Another aspect to consider is resource use per service since certain services may restrict CPU or memory usage [4]. Extra criteria should be guaranteed for each autonomous service, such as availability or behavior monitoring.

The System Coupling through Deployment Strategy decision concerns how services are deployed in execution environments. The following decision options can be chosen: (i) *Multiple Services per Execution Environment*, where services

are all deployed in the same execution environment making it problematic to change, build, and deploy the services independently. 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 runs on a Public Cloud Environment. Execution environments run on Devices (e.g., Cloud Server). The most recommended option is the (ii) *Single Service per Execution Environment* pattern [4], in which each service is deployed in its execution environment and can be managed independently. In our previous work [8], we empirically identified two metrics that can be used to differentiate and assess the decision options' conformance:

- *Shared Execution Environment Connectors Metric (SEEC)* to measure the proportion of the shared connectors between services and execution environments.
- *Shared Execution Environment Metric (SEE)* to measure the proportion of the shared execution environments.

*2) ADD 2: System Coupling through Infrastructure Stack Grouping:* Another essential aspect of microservices deployment is the grouping of the infrastructure elements. The System Coupling through Infrastructure Stack Grouping decision concerns how grouping different resources into infrastructure stacks should reflect the development teams' responsibilities to ensure independent deployability and scalability. The following decision options can be chosen: *Monolith Stack* [2], where all resources are grouped in a single stack. Another option is *Application Group Stack*, in which multiple services are deployed by one stack. A structuring that can work better with microservice-based systems is the *Service Stack*, in which one stack deploys one service and all related infrastructure resources. The *Micro Stack* pattern [2] goes one step further by breaking the *Service Stack* into even smaller pieces and creating stacks for each infrastructure resource in a service (e.g., router, server, database, etc.). For this decision, we have empirically defined six metrics that can be used to assess conformance to each of the decision options:

- *Monolithic Stack Detection Metric (MSD)* to detect if a single stack is used to deploy all the infrastructure elements.
- *Application Group Stack Detection Metric (AGSD)* to detect if a single stack is used to deploy all system services.
- *Service-Stack Detection Metric (SES)* to detect if every service is deployed by its own stack.
- *Micro-Stack Detection Metric (MST)* to detect if every infrastructure element is deployed by its own stack.
- *Services per Stack Metric (SPS)* to measure how many services are deployed by a service-deploying stack on average.
- *Components per Stack Metric (CPS)* to measure how many components, on average, are deployed by a component-deploying stack.

## III. Related Work

In this section, we provide details on and compare related works. We first discuss related studies for IaC-based best practices and patterns, then tool-based approaches for smell detection, and finally, approaches for evaluating the conformance of architectures.

### A. Related Works on IaC-Based Best Practices and Patterns

As the industry adopts and popularizes IaC practices, many scientific studies are compiling or organizing IaC-related patterns, practices, smells, and anti-patterns. For example, a list of design and implementation language-specific smells for Puppet is presented by Sharma et al. [9]. Kumara et al. [10] offer a comprehensive list of best and worst practices relating to implementation problems, design problems and smells of fundamental IaC concepts. Schwarz et al. [11] provide a list of smells for Chef. Morris [2] provides management recommendations for infrastructure as code. This book includes an extensive list of patterns and practices that fall under several categories and a complete discussion of technologies relevant to IaC-based practices. Our work follows the IaC-specific principles outlined in this book and those in [4]. Many of these publications are less concerned with architecture decisions in the deployment architecture than our work is. In contrast to our research, they do not provide architecture conformance assessment or detect and resolve architecture smells.

### B. Tool-based and Network Smell Detection Approaches

A tool-based approach for detecting smells in TOSCA models is proposed by Kumara et al. [12]. Sotiropoulos et al. [13] develop a tool-based approach that identifies dependency-related issues by analyzing Puppet manifests and their system call trace. Van der Bent et al. [14] define metrics that also reflect best practices to assess Puppet code quality. Saatkamp et al. [15] utilize architectural and design patterns to reorganize topology-driven deployment models to identify any issues obstructing a successful deployment. Their approach covers two aspects: (1) identifying problems in reorganized deployment models through architecture and design patterns and (2) automating problem detection by formalizing the issue and its context through implementing patterns. This work presents a method for identifying and implementing suitable solutions for issues in declarative deployment models in an automated manner. Saatkamp et al. [16] also present an approach that uses first-order logic to evaluate the applicability of solutions to a specific deployment model by expressing the required deployment context as a logical formula. Adaptation algorithms are also defined to operate on topological elements indicated by the deployment context to realize the solution in the deployment model. In [17] Saatkamp et al. demonstrate using formalized patterns to detect problems in two application scenarios. The Message Mover and Integration Provider patterns, relevant to restructured topology-based deployment models in distributed applications, show the approach's applicability in message-based systems. Reusable conditions to express pattern rules have also been defined. Although some

of these works concentrate on the quality assurance of IaC systems, none of them, unlike our work, address and focus primarily on coupling-related issues in IaC deployment models and on architecture smell detection and fixes.

### C. Related works on Frameworks and Metrics

An approach for automatically verifying declarative deployment models' conformity throughout design time is presented in [18], [19]. The method enables modeling compliance rules as two fragments of a deployment model. One of the parts is a detector subgraph that decides whether the rule applies to a particular deployment model. Subgraph isomorphism compares the model fragments to the deployment model in question. In contrast to our study, this technique generally does not incorporate any particular compliance rules, like checking coupling-related ADDs in IaC models. It presupposes that the rule modeler can convert best practices into compliance rules with the desired format. Additionally, it doesn't indicate the severity of a rule violation; instead, it merely offers a Boolean result showing whether or not the rule is being broken. Weller et al. [20] present the Deployment Model Abstraction Framework (DeMAF), a tool that allows the transformation of technology-specific deployment models into technology-independent deployment models modeled based on the Essential Deployment Metamodel (EDMM). This framework demonstrates the capability of abstracting deployment models in a technology-agnostic manner.

Numerous studies concentrate on methods for spotting design or architectural smells, but most do not specifically target the IaC domain. Garcia et al.'s approach [21], [22] provides a format for a collection of offensive smells in architecture. Additionally, these findings provide potential methods for detecting these architectural smells. The relationship between smells and project problems was investigated by Le et al. [23]. Marinescu [24] has proposed several detection techniques that use metrics-based heuristics to find design flaws. To detect architecture erosion or drift, Garcia et al. [25] describe a machine learning-based method for reconstructing an architectural perspective that includes a system's parts and connectors.

Although several of these publications examine features of the microservice domain and other aspects of architecture smell detection, none address detecting and refactoring coupling-related smells in an IaC domain. This leads to our expectation that, in the context of loose coupling, our work yields more precise detections of decision-specific smells and more focused suggestions for fixes than this other research possibly could.

## IV. Research and Modeling Methods

This section summarizes the main research and modeling methods applied in our study. For reproducibility, all the code and models produced in this study are available online as an open-access data set in a long-term archive [1].

### A. Research Method

The steps used in this study are shown in Figure 3. We have already provided a detailed explanation of the architectural decisions and model-based metrics that served as the foundation for this study in Section II. We offer explicit definitions and algorithms for detecting potential smells for each decision option and detailed definitions and methods for the possible fixes for each smell in Section VI.

Every smell associated with the ADDs listed in the previous section will be found using our method. Any suggested architecture refactorings will be applied to each model in our data set. For each smell fix, we ran all possible smell detection algorithms and refactorings on the resulting refactored models until no more smells were found or the refactored model matched a previous version. In the latter case, this shows that it is impossible to eliminate all smells because doing so would require creating new smells. To assess each final model's improvement over the initial model, we examined pattern conformance using metrics on IaC coupling.

### B. Modeling Method

We used a dataset of 12 IaC-based deployment models (3 case studies and 9 variations) listed in Table I to evaluate our approach. This dataset consists of three sources of microservice-based systems and deployment artifacts. The fact that professionals with relevant expertise created the systems we discovered supports the notion that they serve as a solid example of the IaC coupling-related best practices enumerated in Section II. Figure 2 shows the steps we followed for reconstructing the model from the source code. We conducted a complete manual static code analysis for the IaC models included in the repositories and the source code for the applications. Our modeling tool Codeable models, [2] allows for the precise specification of meta-models, models, and code instances used to create the models. The result is a collection of meticulously designed software systems and IaC-based deployment models. Figure 4 shows an excerpt of an IaC-based deployment model.

## V. Case Studies

This section briefly describes the case studies used to evaluate our approach. We studied three open-source microservice-based systems and created nine variants that introduce typical ADD smells of the ADDs described in Section II. Table I summarizes the case studies and corresponding variants.

*a) Case Study 1: eShopOnContainers Application:* The *eShopOnContainers* case study is a prototype reference application, realized by Microsoft, built on a microservices architecture and Docker containers that can be used with Azure and Azure cloud services. It features several independent microservices and accommodates various communication modes (e.g., synchronous and asynchronous via a message broker). The code repository also offers the necessary IaC scripts to work with ELK for logging and deployment on a Kubernetes cluster (Elasticsearch, Logstash, Kibana).

[1] https://doi.org/10.5281/zenodo.7692017

[2] https://github.com/uzdun/CodeableModels

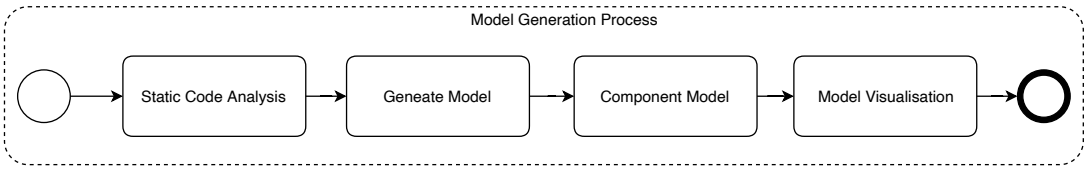


Fig. 2: Overview diagram of the model generation process

*b) Case Study 2: Sock Shop Application:* The *Sock Shop* is a microservices reference application built by the company Weaveworks to show several microservice architectures and the company's technologies. The application showcases cloud-native and microservices technology. Services are deployed on *Docker* containers, and the system employs *Kubernetes* to manage containers. The system may be deployed on Amazon Web Services (AWS) using infrastructure scripts for Terraform. In our opinion, this is a common practice in the industry regarding microservice-based designs and IaC-based deployments.

*c) Case Study 3: Robot-Shop Application: Robot-Shop* is a reference application by the company Instana that showcases polyglot microservice architectures with Instana monitoring. The IaC scripts are included. Kubernetes is used for container orchestration, and all system services are deployed on Docker containers. *Helm* is also supported for automated cluster *Kubernetes* construction, packaging, setup, and deployment. In addition, some services support *Prometheus* metrics and offer end-to-end monitoring.

## VI. Architecture Smells and Fix Options Definition

This section presents an overview of the various smells, possible solutions, and algorithms we have designed for detecting smells and implementing fixes. To further clarify our approach, we also include examples from the decision of "System Coupling via Infrastructure Stack Grouping."

Our previous work [8] presents a microservice-based architecture model description/modeling of a directed graph of components and connectors. This model serves as the foundation for our definitions of smells and fixes.

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 components 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: $nm\_connectors(nm) = \{c \in C_M : (\exists n \in nm : (c\_source(c) = n \wedge c\_target(c) \in C_M) \vee (c\_target(c) = n \wedge c\_source(c) \in C_M))\}$.
- $n\_type : N_M \rightarrow \mathbb{P}(NT_M)$ is a function that maps each node to its set of **direct and transitive node types**. (For a formal definition of node types, see [26].)
- $c\_type : C_M \rightarrow \mathbb{P}(CT_M)$ is a function that maps each connector to its set of **direct and transitive connector types**. (For a formal definition of connector types, see [26].)

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 distinguish logical environments on the same infrastructure, such as a *Test_Environment* and a *Production_Environment*. Combining all types, e.g., *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, e.g., a *Container* that is deployed on a *VM* or a *Test_Environment*. The connector type *runs_on* models 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.

### A. Smell Detection

Table II summarizes the possible smells we have detected for each ADD. It also describes how the algorithms we use to detect smells in models are based on meta-model definition introduced in Section VI. For example, Algorithm 1 describes the steps required for detecting the Smell *Services are Deployed on a Single Execution Environment* of ADD 1. It returns a list of smells, each represented by a set of service environment connectors in which two services $s_m$ and $s_j$ share an execution environment $e_i$.

Algorithm 1: Detect System Services are Deployed on a Single Execution Environment Smell

| *Case Study ID* | *Model Size* | *Description / Source* |
|---|---|---|
| CS1 | 68 components<br>167 connectors | E-shop application using pub/sub communication for event-based interaction and files for deployment on a Kubernetes cluster. All services are deployed in their infrastructure stack (from https://github.com/dotnet-architecture/eShopOnContainers). |
| CS1.V1 | 67 components<br>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<br>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. |
| CS1.V3 | 60 components<br>150 connectors | Variant of Case Study 1 in which some services are deployed on the same execution environment and the non-services components are deployed by a component-deploying stack. |
| CS2 | 38 components<br>95 connectors | An online shop that demonstrates and tests 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<br>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 | 40 components<br>101 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. |
| CS2.V3 | 60 components<br>150 connectors | Variant of Case Study 2 in which some services are deployed on the same execution environment and the non-services components are deployed by a component-deploying stack. |
| CS3 | 32 components<br>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<br>147 connectors | Variant of Case Study 3 where some services are deployed in their infrastructure stack and some services are deployed on the same execution environment. |
| CS3.V2 | 56 components<br>147 connectors | Variant of Case Study 3 where all services are deployed in their infrastructure stack and all services are deployed on their execution environment. |
| CS3.V3 | 54 components<br>148 connectors | Variant of Case Study 3 where some services are deployed in their infrastructure stack and some services are deployed on their execution environment. |

TABLE I: Overview of modeled case studies and the variants (size, details, and sources), adapted from our previous work [8]

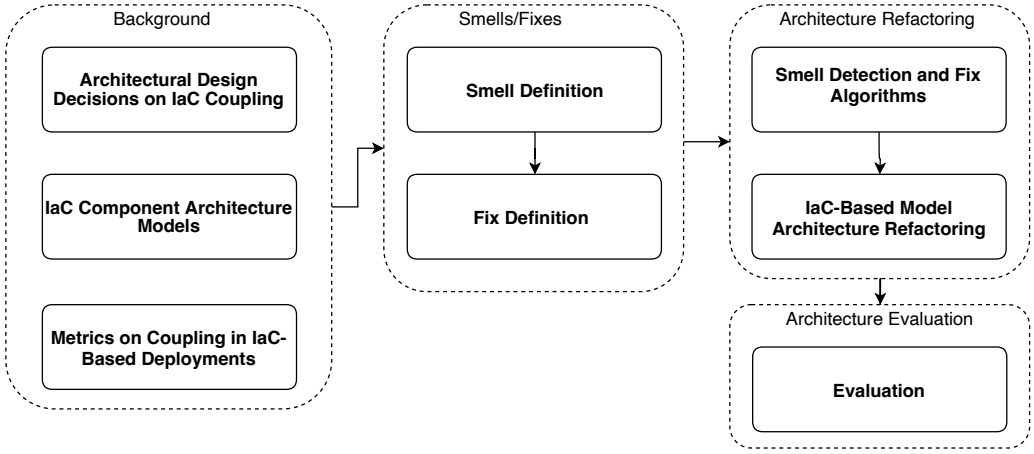


Fig. 3: Overview diagram of the research method followed in this study (the diagram is adapted from our previous work [8])

```
input: Model M
output: Set<Tuple>
begin
smells ← ∅
for s_m ∈ services(M):
  for s_j ∈ services(M):
    for e_i ∈ execution_environment(M):
      if ((s_m, e_i) ∈ service_env_connectors(M) ∧
          (s_j, e_i) ∈ service_env_connectors(M)):
        smells ← smells ∪ {(s_m, e_i), (s_j, e_i)}
  return smells
end
```

### B. Fixes

Table III summarizes all possible fixes for each detected smell and the fix algorithm. Many of the fixes require human review and sometimes a human decision to be applicable. For instance, the architect may be faced with a decision of which infrastructure stack is better suitable to the application requirements. For example, Algorithm 2 shows one of the fix algorithms, integrating services deployed in the same execution environments needed to realize D1.S1.F3.

Algorithm 2: Integrate Services Deployed in the Same Execution Environment (D1.S1.F3)

```
input: Model M, Execution_Environment env
output: –
begin
  new_service ← Null
  first ← True
  integration_annotations ← ∅

  for s ∈ get_services(M, env):
    if first:
      new_service = create_service(M,
```

| *Smells* | *Smell Detection Algorithm Summary* |
|---|---|
| **D1: System Coupling through Deployment Strategy** | |
| *D1.S1: System services are running/deployed on a single execution environment [Host/VM/Container].* | All service connectors in the model are traversed. If at least two services are deployed on the same execution environment, an instance of the smell is found. The detector operation returns each such service-execution connector that is found. |
| **D2: System Coupling through Infrastructure Stack Grouping** | |
| *D2.S1: All infrastructure elements and services are part of a single infrastructure stack.* | All infrastructure elements connectors in the model are traversed. If only one infrastructure stack is found to be used by them, an instance of this smell is found. The detector operation returns the list of all relevant model elements. |
| *D2.S2: Two or more services are part of a single infrastructure stack.* | All service and stack connectors in the model are traversed. The smell is found if multiple services are clustered in groups on at least one of the stacks. The detector operation returns the list of all relevant model elements. |
| *D2.S3: If* Service Stack *is* True*: Infrastructure elements (e.g., databases, routers, etc.) that services depend on are not part of their service stack.* | All infrastructure elements connectors in the model are traversed. Suppose non-service components (e.g., databases) are connected to a different stack than their services. In that case, the smell is found. The detector operation returns the list of all such model elements. |
| *D2.S4: If* Service Stack *is* True*: Infrastructure elements (e.g., databases, routers, etc.) that services are not dependent on are part of their service stack.* | All infrastructure elements connectors in the model are traversed. Suppose non-service components (e.g., databases) are connected to a stack they are not dependent on. In that case, the smell is found, and the detector operation returns the list of all relevant model elements. |

TABLE II: Detected Smells and Smell Detection Algorithms

| *Smell* | *Fix* | *Fix Summary* |
|---|---|---|
| **D1: System Coupling through Deployment Strategy** | | |
| *D1.S1* | *D1.S1.F1: Do not fix the smell* | The architect has the option to *not* fix the smell, e.g. because it has no significant impact on system deployment. |
| | *D1.S1.F2: Deploy each service in a separate execution environment* | Disconnect services from the execution environment and introduce a new execution environment for each service. Connect the services to the execution environments. |
| | *D1.S1.F3: Integrate the services deployed on the same execution environment into one service* | Disconnect services from the execution environment. Merge the services using the same execution environment into a single service using that execution environment. Connect the new service to the execution environment. Here we add annotations that functionality has been added to one service so that implementers, later on, can realize this functionality. The architect must check whether such integration is possible and can provide developers with annotations about the envisaged service integration details. |
| | *D1.S1.F4: Introduce VMs/Containers in a Host/VM to separate services in different execution environments* | Disconnect services from the execution environment and introduce new VMs/containers for each service in the same host. Connect the services to the VMs/containers. |
| **D2: System Coupling through Infrastructure Stack Grouping** | | |
| *D2.S1* | *D2.S1.F1: Do not fix the smell* | The architect has the option to *not* fix the smell, e.g. because it has no significant impact on system deployment. |
| | *D2.S1.F2: Create separate infrastructure stacks for each service and additional infrastructure elements* | Disconnect services from the infrastructure stack. Introduce new infrastructure stacks for each service and additional infrastructure elements. Connect the services and elements to their stack. |
| *D2.S2* | *D2.S2.F1: Do not fix the smell* | The architect has the option to *not* fix the smell, e.g. because it has no significant impact on system deployment. |
| | *D2.S2.F2: Create separate infrastructure stacks for each service* | Disconnect services from the infrastructure stack. Introduce new infrastructure stacks for each service and connect the services to their stack. |
| *D2.S3* | *D2.S3.F1: Do not fix the smell* | The architect has the option to *not* fix the smell, e.g. because it has no significant impact on system deployment. |
| | *D2.S3.F2: Move the service-dependent infrastructure elements in the same service stack* | Disconnect service-dependent infrastructure elements from the infrastructure stack. Connect service-dependent infrastructure elements to the infrastructure stacks they are dependent on. |
| *D2.S4* | *D2.S4.F1: Do not fix the smell* | The architect has the option to *not* fix the smell, e.g. because it has no significant impact on system deployment. |
| | *D2.S4.F2: Move the service-independent infrastructure elements in the dependent service stack* | Disconnect service-independent infrastructure elements from the infrastructure stack. Connect service-independent infrastructure elements to the infrastructure stacks they are dependent on. |

TABLE III: Detected Fixes And Fix Algorithms

```
      get_service_name(M, s),
      get_applicable_stereotypes(M, s))
    first ← False
else:
    set_service_name(M, new_service,
      get_service_name(M, new_service) + " + " +
      get_service_name(M, s))
    add_applicable_stereotypes(M, new_service, s)

integration_annotations ← integration_annotations
    ∪ {"integrated functionality from: " +
       get_service_name(M, s)}
    add_connector(new_service, env,
      get_applicable_stereotypes(M, (s, env)))
    delete_service(s)

  add_annotations(M, new_service, integration_annotations)
end
```

### C. Smell Detection and Fixes Example

Figure 4 shows an excerpt model of CS1.V2 from Table I. As an illustrative example, we use it here to demonstrate the *System services are running/deployed on a single execution environment [Host/VM/Container](D1.S1)* smell. All services are deployed on a single *Container* in this model. Here, the *Docker Container 0.0.0.3* is considered as *shared execution environment*, causing the corresponding smell. It would be triggered in our approach by providing a bad metric value, which would trigger the detailed detection, which would return the $\{$(*Catalog, Docker Container 0.0.0.3*), (*Basket, Docker Container 0.0.0.3*), (*Order, Docker Container 0.0.0.3*)$\}$ set of tuples. If we run our fix algorithms, the resulting model fix suggestions are:

- *Applying Fix D1.S1.F2: Catalog*, *Order* and *Basket* services will be disconnected from the execution environment. The fix will introduce different execution environments for each service, which will be connected to its own execution environment.
- *Applying Fix D1.S1.F3:* The *Catalog*, *Order* and *Basket* services can be integrated into one new service.
- *Applying Fix D1.S1.F4:* The fix will introduce new execution environments for each service as part of the same host. *Catalog*, *Order*, and *Basket* services will be disconnected from the execution environment, and each service will be connected to its execution environment.

## VII. Evaluation

To evaluate our work, we have fully implemented our algorithms for detecting smells and performing fixes, and generating the metrics described in Section II to measure the improvements and presence of remaining smells in our model set. If multiple smells are present in a model, then the algorithms can be employed iteratively until all smells have been fully resolved.

For example, let us illustrate the exhaustive, iterative refactoring for the CS1.V2 Model (see Figure 4). CS1.V2 has the following smells—"System services are running/deployed on a single execution environment" (D1.S1), "Two or more services are part of the same infrastructure stack" (D2.S2), and "Infrastructure elements (e.g., databases, routers, etc.) that services depend on are not part of their service stack" (D2.S3) as indicated by the respective measures in Table II. There are two branches at the first iteration step of the refactoring process in Table IV. The first iteration step results in 4 possible model variants, one for each fix option from Table III. In these models, the corresponding smells have been fixed. However, all the new models still contain a smell. M1–M3 still has the D2.S2 smell since this is not resolved in this branch. M4 still has the D1.S3 smell.

The second iteration step results in four further models. In turn, the resulting models M1.A, M2.A, and M3.A, now contain the additional smells D2.S3. At the end of the third step, we have four suggested model variants, all optimally resolving the smells. The architect can choose the refactoring sequence from among these final optimal model variants but can also choose not to apply specific fixes, e.g., due to other constraints outside our study's scope.

We followed this technique for *all* 12 system models in Table I to evaluate them. In Table V, along with the starting smells and architecture evaluation values for each model, are the number of intermediate models and smell cases at every step, as well as the number of final suggested models with an optimal metric assessment. Please note that the metrics below correspond to each of the smells; D1.S1 has two related metrics, and D2.S2 has three metrics, one for detecting the *Application Group Stack* pattern, one for detecting the *Service Stack* pattern, and one to measure the proportion of this.

The number of smells in the starting model and the potential emergence of additional smells throughout the refactoring process determines the number of steps necessary to attain ideal models. All models are fully resolved, or all assessment metrics have optimal values after a maximum of three phases, as shown in Table V.

## VIII. Discussion of Research Questions

For each potential alternative option, we methodically detected several decision-based smells, which are included in Table II to address **RQ1**. The purpose of the smell detectors is to discover the precise locations in the models where the smells occur because we have empirically demonstrated in our prior work [8] that the metrics described in Section II can reliably distinguish preferred or less preferred design options. For each system model in our evaluation dataset, proposing corrections to improve the architecture was possible. This indicates that the algorithms had correctly detected the resolution's proper location or locations.

We built a variety of methods for **RQ2** that addressed every conceivable smell and provided several correction alternatives (see Table III). A search tree of potential architecture models is produced if every option is tested (such as the one shown in Table IV). This search tree may then be evaluated using metrics to gauge how much the basic architecture has improved and detect unresolved issues. We have demonstrated that an iterative method of employing our algorithms successively yields, within a few steps, a variety of potential architectural models that eliminate all smells detected and guarantee pattern conformity of the system architecture (see Table V). The numerous ideal model versions produced by our method offer architects a great deal of design flexibility. The approach is suited to be used in a continuous delivery environment, which was one of our study's aims. This is because detection is automated, and human expertise is only used in the fix process.

## IX. Threats to Validity

The information and solutions presented in our study are based on published literature and best practices in the field. Our evaluation dataset consists of a representative collection of systems drawn from three different sources and specifically selected to demonstrate various features of IaC architectures (see

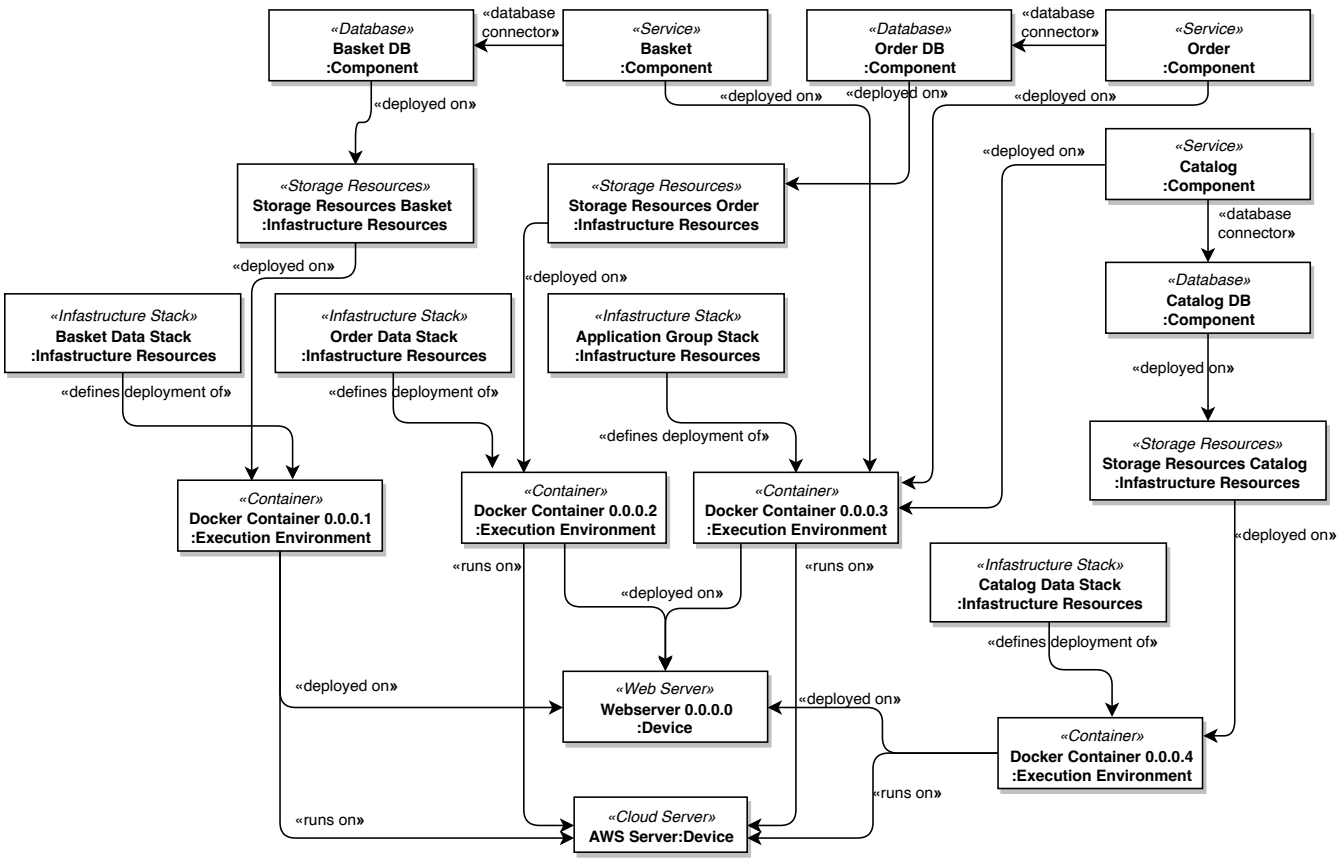


Fig. 4: Excerpt of an Architecture Component Model of Case CS1.V2 in Table I.

| | CS1.V2 | | |
|---|---|---|---|
| *Step 1* | Smells | D1.S1 | D2.S2 |
| | Produced Component Models (Fixes) | M1, M2, M3 | M4 |
| *Step 2* | Smells | D2.S2 | D2.S3 |
| | Produced Component Models (Fixes) | M1.A, M2.A, M3.A | **M4.A** |
| *Step 3* | Smells | D2.S3 | No additional smell |
| | Produced Component Models (Fixes) | **M1.A-1, M2.A-2, M3.A-3** | **–** |
| **Total** | **4 Optimal Component Models** | | |

TABLE IV: Example of an exhaustive iterative application of our approach in the CS1.V2 model. Final (i.e. optimally resolved) resulting models are rendered in boldface font.

Table I). While our method is based on traditional component-and-connector models, widely used in the literature, and modified to include deployment aspects, it is designed to be abstract and general.

To ensure our results' accuracy and reliability, the authors' team carried out the modeling process, and all models were independently cross-checked. The authors have extensive expertise in modeling methodologies and are confident that alternative interpretations of the models would still be generally similar and compatible with our findings. However, it should be noted that our method depends on a specific modeling strategy and may not apply to all architectures.

One of the main limitations of our study is that it only considers two specific ADDs and the associated trends, metrics, and issues. In real-world architectures, it would be necessary to consider a broader range of ADDs to evaluate the architecture fully. Additionally, our measurements and tools were applied at a relatively high level of abstraction to accommodate different IaC technologies, such as Ansible, Terraform, and Puppet.

Another potential limitation of our technique is its ability to effectively address larger, more complex systems commonly found in industry but which we could not include in our research. While our method is automated to some extent, it still requires input and guidance from the architect, which may make it challenging to implement in practice. Additionally, our method cannot match the expertise and ability of a skilled architect to design a more optimal solution. This is a common

| Model ID | Initial Model Assessments | | | | Models Generated / Remaining smell Instances per Refactoring Step | | | Resulting Suggested Models |
|---|---|---|---|---|---|---|---|---|
| | *D1.S1* | *D2.S1* | *D2.S2* | *D2.S3, D2.S4* | *Step 1* | *Step 2* | *Step 3* | |
| CS1 | 0.00, 0.00 | False | True, False, 1.00 | True, 1.00 | 1 / 1 | 1 / 0 | – | 1 |
| CS1.V1 | 0.71, 0.50 | False | False, False, 0.20 | True, 1.00 | 4 / 3 | 3 / 0 | – | 4 |
| CS1.V2 | 0.42, 0.20 | False | False, False, 0.57 | False, 0.50 | 4 / 4 | 4 / 3 | 3 / 0 | 4 |
| CS1.V3 | 0.57, 0.25 | False | False, False, 0.42 | False, 0.33 | 4 / 4 | 4 / 3 | 3 / 0 | 4 |
| CS2 | 0.00, 0.00 | True | False,False, 0.00 | False, 0.00 | 1 / 1 | 1 / 0 | – | 1 |
| CS2.V1 | 0.25, 0.14 | False | False,False, 0.12 | False, 1.00 | 4 / 3 | 3 / 0 | – | 4 |
| CS2.V2 | 0.62, 0.40 | False | False,True, 0.00 | False, 0.00 | 4 / 4 | 6 / 0 | – | 6 |
| CS2.V3 | 0.25, 0.14 | False | False,False, 0.12 | False, 0.33 | 4 / 4 | 4 / 3 | 3 / 0 | 4 |
| CS3 | 0.00, 0.00 | False | False,False, 0.00 | False, 0.00 | 1 / 1 | 1 / 0 | – | 1 |
| CS3.V1 | 0.37, 0.16 | False | False,False, 0.62 | False, 1.00 | 4 / 4 | 6 / 0 | – | 6 |
| CS3.V2 | 0.00, 0.00 | False | True,False, 1.00 | True, 1.00 | 1 / 1 | 1 / 0 | – | 1 |
| CS3.V3 | 0.25, 0.14 | False | False,False, 0.75 | False, 0.33 | 4 / 4 | 4 / 3 | 3 / 0 | 4 |

TABLE V: This table shows the results of evaluating the initial models used in our study. It includes the number of models created at each step of applying our algorithms in an iterative process, the number of smell instances (calculated by multiplying the number of generated models by the number of smells per model) that remained or were introduced in each iteration, and the final count of recommended (optimal) models.

limitation of generic architecture assistance techniques that we aim to address in future research.

We want to emphasize that our current approach is just a starting point for examining the issue of evaluating and improving IaC architectures. The models we have produced do not yet consider factors such as the amount of code or rewriting needed to implement them. Despite these limitations, it is still valuable to have a semi-automatic approach that can detect and analyze violations of architectural best practices, even if some of these issues may be inevitable in practice. Practitioners may not always adhere to best practices, and systems may be developed without a deliberate effort to follow them or may drift from their original specifications over time.

## X. Conclusion and Future Work

In this work, we investigate the use of coupling-related architectural design decisions in infrastructure as code architectures and their impact on the system's overall design. We identify specific "smells" that may indicate issues and have developed automated detectors to locate the source of these smells within the model. We have created a set of potential solutions for each detected smell to address these smells and ensure adherence to best practices in microservice-based architectures.

We conducted three case studies on open-source microservice-based systems to evaluate our approach. We introduced smells or refactorings to 9 variants of these systems to test the performance of our smell detection algorithms in more complex scenarios. Our analysis of these models, which ranged in architectural complexity and the presence of patterns and smells, showed that our technique could eliminate smells in just three refactoring steps, most of which could be automated.

One of the main advantages of our method is its fully automated metric computation and smell detection, which makes it suitable for incorporation into a continuous delivery pipeline as an additional "architecture evaluation" step. While the proposed fixes on the IaC-based models are automated, architects still have the flexibility to make design choices and provide feedback as needed.

We plan to expand the range of ADDs and smells supported by our method and improve it by including runtime metrics and other design components. We also plan to increase the size and complexity of our model dataset and empirically validate our approach through its use in real delivery pipelines as part of a feedback loop.

## XI. Acknowledgments.

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

## References

[1] M. Nygard, *Release It! Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2007.

[2] K. Morris, *Infrastructure as Code: Dynamic Systems for the Cloud*. O'Reilly, 2020, vol. 2.

[3] 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)*, 2017, pp. 497–498.

[4] C. Richardson, "A pattern language for microservices," http://microservices.io/patterns/index.html, 2017.

[5] O. Zimmermann, M. Stocker, U. Zdun, D. Luebke, and C. Pautasso, "Microservice API patterns," https://microservice-api-patterns.org, 2019.

[6] J. Skowronski, "Best practices for event-driven microservice architecture," https://hackernoon.com/best-practices-for-event-driven-microservice-architecture-e034p21lk, 2019.

[7] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *2009 13th European Conference on Software Maintenance and Reengineering*, 2009, pp. 255–258.

[8] E. Ntentos, U. Zdun, J. Soldani, and A. Brogi, "Assessing architecture conformance to coupling-related infrastructure-as-code best practices: Metrics and case studies," in *16th European Conference on Software Architecture*, ser. Software Architecture - 16th European Conference, ECSA 2022, Prague, Czech Republic, September 19?23, 2022, Proceedings, September 2022, pp. 101–116. [Online]. Available: http://eprints.cs.univie.ac.at/7338/

[9] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 189–200.

[10] I. Kumara, M. Garriga, A. U. Romeu, D. Di Nucci, F. Palomba, D. A. Tamburri, and W.-J. van den Heuvel, "The do's and don'ts of infrastructure code: A systematic gray literature review," *Information and Software Technology*, vol. 137, p. 106593, 2021.

[11] J. Schwarz, A. Steffens, and H. Lichter, "Code smells in infrastructure as code," in *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, 2018, pp. 220–228.

[12] I. Kumara, Z. Vasileiou, G. Meditskos, D. A. Tamburri, W.-J. Van Den Heuvel, A. Karakostas, S. Vrochidis, and I. Kompatsiaris, "Towards semantic detection of smells in cloud infrastructure code," in *Proceedings of the 10th International Conference on Web Intelligence, Mining and Semantics*, ser. WIMS 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 63–67.

[13] T. Sotiropoulos, D. Mitropoulos, and D. Spinellis, "Practical fault detection in puppet programs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 26–37.

[14] E. van der Bent, J. Hage, J. Visser, and G. Gousios, "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)*, 2018, pp. 164–174.

[15] K. Saatkamp, U. Breitenbücher, O. Kopp, and F. Leymann, "An approach to automatically detect problems in restructured deployment models based on formalizing architecture and design patterns," *SICS Softw.-Inensiv. Cyber-Phys. Syst. 34*, p. 85–97, 2019.

[16] K. Saatkamp, U. Breitenbücher, M. Falkenthal, L. Harzenetter, and F. Leymann, "An approach to determine & apply solutions to solve detected problems in restructured deployment models using first-order logic," in *International Conference on Cloud Computing and Services Science*, 2019.

[17] K. Saatkamp, U. Breitenbücher, O. Kopp, and F. Leymann, "Application scenarios for automated problem detection in tosca topologies by formalized patterns," in *Papers From the 12th Advanced Summer School on Service-Oriented Computing (SummerSOC'18)*. IBM Research Division, 2018, pp. 43–53.

[18] M. P. Fischer, U. Breitenbücher, K. Képes, and F. Leymann, "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)*. Xpert Publishing Services (XPS), 2017, pp. 150–153.

[19] C. Krieger, U. Breitenbücher, K. Képes, and F. Leymann, "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)*. IBM Research Division, Oktober 2018, Konferenz-Beitrag, pp. 76–89.

[20] M. Weller, U. Breitenbücher, S. Speth, and S. Becker, "The deployment model abstraction framework."

[21] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *2009 13th European Conference on Software Maintenance and Reengineering*, 2009, pp. 255–258.

[22] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Toward a catalogue of architectural bad smells," in *Architectures for Adaptive Software Systems*, R. Mirandola, I. Gorton, and C. Hofmeister, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 146–162.

[23] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural decay in open-source software," in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 176–17 609.

[24] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, 2004, pp. 350–359.

[25] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Yuanfang Cai, "Enhancing architectural recovery using concerns," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 2011, pp. 552–555.

[26] U. Zdun, E. Navarro, and F. Leymann, "Ensuring and assessing architecture conformance to microservice decomposition patterns," in *Service-Oriented Computing*, M. Maximilien, A. Vallecillo, J. Wang, and M. Oriol, Eds. Cham: Springer International Publishing, 2017, pp. 411–429.