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

Evangelos Ntentos¹, Uwe Zdun¹, Ghareeb Falazi², Uwe Breitenbücher², Frank Leymann²

¹ Software Architecture Group, University of Vienna, Austria
firstname.lastname@univie.ac.at

² Institute of Architecture of Application Systems, University of Stuttgart, Germany
{falazi,breitenbuecher,leymann}@iaas.uni-stuttgart.de

BIBTEX:

@inproceedings{Ntentos2023_CouplingIaCSmells,
  Title = {{Detecting and Resolving Coupling-Related Infrastructure as Code Based Architecture Smells in Microservice Deployments}},
  Author = {Ntentos, Evangelos and Zdun, Uwe and Falazi, Ghareeb and Breitenb"{u}cher, Uwe and Leymann, Frank},
  Year = 2023,
  Month = jul,
  Booktitle = {2023 IEEE 16th International Conference on Cloud Computing (CLOUD)},
  Publisher = {IEEE},
  Pages = {201--211},
  Doi = {10.1109/CLOUD60044.2023.00031}
}

© 2023 IEEE Computer Society. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
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 refactoring 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 [3] 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 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 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 [I] 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: Monolithic 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 [3]. 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. García 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, García 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
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 used 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.
- $nm_connectors : \mathcal{P}(N_M) \rightarrow \mathcal{P}(C_M)$ is a function returning the set of connectors for a set of nodes:

\[
\begin{align*}
\text{nm_connectors}(nm) = \{c \in C_M : (\exists n \in nm : (c_{source}(c) = n \land c_{target}(c) \in C_M) \lor (c_{target}(c) = n \land c_{source}(c) \in C_M))\}
\end{align*}
\]

- $n_{type} : N_M \rightarrow \mathcal{P}(NT_M)$ is a function that maps each node to its set of direct and transitive node types.
- $c_{type} : C_M \rightarrow \mathcal{P}(CT_M)$ is a function that maps each connector to its set of direct and transitive connector types.

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

<table>
<thead>
<tr>
<th>Case Study ID</th>
<th>Model Size</th>
<th>Description / Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS1</td>
<td>68 components</td>
<td>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 <a href="https://github.com/dotnet-architecture/eShopOnContainers">https://github.com/dotnet-architecture/eShopOnContainers</a>).</td>
</tr>
<tr>
<td>CS1.V1</td>
<td>67 components</td>
<td>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.</td>
</tr>
<tr>
<td>CS1.V2</td>
<td>60 components</td>
<td>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.</td>
</tr>
<tr>
<td>CS1.V3</td>
<td>60 components</td>
<td>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.</td>
</tr>
<tr>
<td>CS2</td>
<td>38 components</td>
<td>An online shop that demonstrates and tests microservice and cloud-native technologies and uses a single infrastructure stack to deploy all the elements (from <a href="https://github.com/microservices-demo/microservices-demo">https://github.com/microservices-demo/microservices-demo</a>).</td>
</tr>
<tr>
<td>CS2.V1</td>
<td>40 components</td>
<td>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.</td>
</tr>
<tr>
<td>CS2.V2</td>
<td>40 components</td>
<td>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.</td>
</tr>
<tr>
<td>CS2.V3</td>
<td>60 components</td>
<td>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.</td>
</tr>
<tr>
<td>CS3</td>
<td>32 components</td>
<td>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 <a href="https://github.com/instana/robot-shop">https://github.com/instana/robot-shop</a>).</td>
</tr>
<tr>
<td>CS3.V1</td>
<td>56 components</td>
<td>Variant of Case Study 3 where some services are deployed in their infrastructure stack and some services are deployed on the same execution environment.</td>
</tr>
<tr>
<td>CS3.V2</td>
<td>56 components</td>
<td>Variant of Case Study 3 where all services are deployed in their infrastructure stack and all services are deployed on their execution environment.</td>
</tr>
<tr>
<td>CS3.V3</td>
<td>54 components</td>
<td>Variant of Case Study 3 where some services are deployed in their infrastructure stack and some services are deployed on their execution environment.</td>
</tr>
</tbody>
</table>

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)

```plaintext
input: Model M, Execution_Environment env
output: –
begin
new_service ← Null
first ← True
integration_annotations ← Ø
for s ∈ get_services(M, env):
    if first:
        new_service = create_service(M,
        service = s,
        env = env)
        first ← False
        integration_annotations ← Ø
    else:
        add_service(new_service, s, env)
        integration_annotations ← Ø
    integration_annotations ← add_annotations(integration_annotations, s)
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.

Fig. 3: Overview diagram of the research method followed in this study (the diagram is adapted from our previous work [8])
TABLE II: Detected Smells and Smell Detection Algorithms

<table>
<thead>
<tr>
<th>Smell</th>
<th>Fit Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2: System Coupling through Infrastructure Stack Grouping</td>
<td>Fit Summary</td>
</tr>
<tr>
<td>D2.S4: Move the service-independent infrastructure elements in the same service stack</td>
<td>Fix the smell, e.g. because it has no significant impact on system deployment. Connect the services to the VMs/containers.</td>
</tr>
<tr>
<td>D2.S3: Move the service-dependent infrastructure elements to the infrastructure stack they are dependent on</td>
<td>Fix the smell, e.g. because it has no significant impact on system deployment. Connect the services to the VMs/containers.</td>
</tr>
<tr>
<td>D2.S2: Two or more services are part of a single infrastructure stack</td>
<td>Fix the smell, e.g. because it has no significant impact on system deployment. Connect the services to the VMs/containers.</td>
</tr>
<tr>
<td>D2.S1: Additional infrastructure elements are part of their service stack.</td>
<td>Fix the smell, e.g. because it has no significant impact on system deployment. Connect the services to the VMs/containers.</td>
</tr>
</tbody>
</table>

TABLE III: Detected Fixes and Fix Algorithms

<table>
<thead>
<tr>
<th>Fix</th>
<th>Fit Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1: System Coupling through Deployment Strategy</td>
<td>Fix Summary</td>
</tr>
<tr>
<td>D1.S1: Do not fix the smell</td>
<td>Fix the smell, e.g. because it has no significant impact on system deployment. Connect the services to the VMs/containers.</td>
</tr>
<tr>
<td>D1.S2: Deploy each service in a separate execution environment</td>
<td>Fix the smell, e.g. because it has no significant impact on system deployment. Connect the services to the VMs/containers.</td>
</tr>
<tr>
<td>D1.S3: Integrate the services deployed on the same execution environment</td>
<td>Fix the smell, e.g. because it has no significant impact on system deployment. Connect the services to the VMs/containers.</td>
</tr>
<tr>
<td>D1.S4: Introduce VMs/Containers in a new execution environment for each service and additional infrastructure elements. Connect the services and elements to their stack.</td>
<td>Fix the smell, e.g. because it has no significant impact on system deployment. Connect the services to the VMs/containers.</td>
</tr>
</tbody>
</table>

D2: System Coupling through Infrastructure Stack Grouping

- All service elements (i.e., service components) are part of the infrastructure stack. The detector operation returns the list of all relevant model elements. The detector operation returns each such service-execution connector that is part of their service stack.
- If multiple services are clustered in groups on at least one of the stacks, the smell is found. The detector operation returns the list of all relevant model elements. The detector operation returns each such service-execution connector that is part of their service stack.
- All service elements (i.e., service components) are part of the execution environment (e.g., a VM or a container). The detector operation returns the list of all relevant model elements. The detector operation returns each such service-execution connector that is part of their service stack.

D1: System Coupling through Deployment Strategy

- All service elements (i.e., service components) are part of the deployment strategy. If at least two services are deployed on the same execution environment, an instance of the smell is found. The detector operation returns the list of all relevant model elements. The detector operation returns each such service-execution connector that is part of their service stack.

TABLE IV: Detected Smells and Smell Detection Algorithms

<table>
<thead>
<tr>
<th>Smell</th>
<th>Fit Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2: System Coupling through Infrastructure Stack Grouping</td>
<td>Fix Summary</td>
</tr>
<tr>
<td>D2.S4: Move the service-independent infrastructure elements in the same service stack</td>
<td>Fix the smell, e.g. because it has no significant impact on system deployment. Connect the services to the VMs/containers.</td>
</tr>
<tr>
<td>D2.S3: Move the service-dependent infrastructure elements to the infrastructure stack they are dependent on</td>
<td>Fix the smell, e.g. because it has no significant impact on system deployment. Connect the services to the VMs/containers.</td>
</tr>
<tr>
<td>D2.S2: Two or more services are part of a single infrastructure stack</td>
<td>Fix the smell, e.g. because it has no significant impact on system deployment. Connect the services to the VMs/containers.</td>
</tr>
<tr>
<td>D2.S1: Additional infrastructure elements are part of their service stack.</td>
<td>Fix the smell, e.g. because it has no significant impact on system deployment. Connect the services to the VMs/containers.</td>
</tr>
</tbody>
</table>

TABLE V: Detected Fixes and Fix Algorithms

<table>
<thead>
<tr>
<th>Fix</th>
<th>Fit Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1: System Coupling through Deployment Strategy</td>
<td>Fix Summary</td>
</tr>
<tr>
<td>D1.S1: Do not fix the smell</td>
<td>Fix the smell, e.g. because it has no significant impact on system deployment. Connect the services to the VMs/containers.</td>
</tr>
<tr>
<td>D1.S2: Deploy each service in a separate execution environment</td>
<td>Fix the smell, e.g. because it has no significant impact on system deployment. Connect the services to the VMs/containers.</td>
</tr>
<tr>
<td>D1.S3: Integrate the services deployed on the same execution environment</td>
<td>Fix the smell, e.g. because it has no significant impact on system deployment. Connect the services to the VMs/containers.</td>
</tr>
<tr>
<td>D1.S4: Introduce VMs/Containers in a new execution environment for each service and additional infrastructure elements. Connect the services and elements to their stack.</td>
<td>Fix the smell, e.g. because it has no significant impact on system deployment. Connect the services to the VMs/containers.</td>
</tr>
</tbody>
</table>

D2: System Coupling through Infrastructure Stack Grouping

- All service elements (i.e., service components) are part of the infrastructure stack. The detector operation returns the list of all relevant model elements. The detector operation returns each such service-execution connector that is part of their service stack.
- If multiple services are clustered in groups on at least one of the stacks, the smell is found. The detector operation returns the list of all relevant model elements. The detector operation returns each such service-execution connector that is part of their service stack.
- All service elements (i.e., service components) are part of the execution environment (e.g., a VM or a container). The detector operation returns the list of all relevant model elements. The detector operation returns each such service-execution connector that is part of their service stack.
Figure IV shows an excerpt model of CS1.V2 from Table II. 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 III 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 IV). 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 III. 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, 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 II to evaluate them. In Table IV 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 IV.

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 III 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 IV). 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.

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
models, which ranged in architectural complexity and the algorithms in more complex scenarios. Our analysis of these systems to test the performance of our smell detection algorithms in more complex scenarios. We introduced smells or refactorings to 9 variants of these microservice-based systems to evaluate our approach.

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: I 4731-N.

REFERENCES


