Model-Driven End-to-End Resolution of Security Smells in Microservice Architectures

Philip Wizenty[3][a], Francisco Ponce[2,4][b], Florian Rademacher[3][c], Jacopo Soldani[1][d], Hernán Astudillo[2,4][e], Antonio Brogi[1][f] and Sabine Sachweh[3][g]

[1] *University of Pisa, Pisa, Italy*

[2] *Universidad Técnica Federico Santa María, Valparaíso, Chile*

[3] *IDiAL Institute, University of Applied Sciences and Arts Dortmund, Germany*

[4] *ITiSB, Universidad Andrés Bello, Viña del Mar, Chile*

ﬂ

Keywords: Microservice Architecture, Model-Driven Engineering, Security, Refactoring.

Abstract: Microservice Architecture (MSA) is a popular approach to designing, implementing, and deploying complex software systems. However, MSA introduces inherent challenges associated with distributed systems—one of them is the detection and mitigation of *security smells*. This paper draws on recent works that identified and categorized security smells in MSAs to propose a novel end-to-end approach for resolving security smells in existing MSAs. To this end, the presented approach extends a modeling ecosystem for MSAs with (i) reconstruction capabilities that automatically map MSA source code to viewpoint-specific architecture models; (ii) validations that detect security smells from reconstructed models; and (iii) model refactorings that support the interactive resolution of security smells and solutions' reflection back to source code. Our approach allows for (i) uncovering security smells, which originate from the combination of different places in source code with possibly heterogeneous purposes, technologies, and software languages; as well as (ii) clustering, reifying, and fixing smells using a level of abstraction that is directed towards MSA stakeholders. The applicability and effectiveness of our approach are evaluated utilizing a standard case study from MSA research.

1 INTRODUCTION

Microservice Architecture (MSA) is a popular approach for developing complex and scalable software applications (Newman, 2015). MSA involves decomposing a software architecture into independent services with distinct functionalities, which leads to cloud-native applications, leveraging network-based communication and supporting state isolation, horizontal scaling, and flexible deployment on cloud platforms (Kratzke and Quint, 2017).

The distributed nature of MSAs makes them inherently prone to *security smells*, which denote poor (often unintentional) design decisions that harm application security (Ponce et al., 2022a). The resolution of *security smells* often requires adapting code in different places of the application with heterogeneous purposes, technologies, and software languages. For example, the resolution of the Publicly Accessible Microservices smell (Ponce et al., 2022b)—a bad practice in MSA engineering that exposes microservice interfaces to architecture-external callers instead of hiding them behind API gateways (Balalaie et al., 2016)—requires (i) introduction of gateway programming and configuration code; (ii) adaptation of microservice programming and configuration code to connect with the introduced gateway; and (iii) adaptation of deployment code to cover the gateway. This scattering of smell across heterogeneous architecture components significantly aggravates their detection and holistic resolution.

This paper presents an end-to-end approach for resolving security smells in existing MSAs that autom-

[a] https://orcid.org/0000-0002-3588-5174
[b] https://orcid.org/0000-0002-6411-0511
[c] https://orcid.org/0000-0003-0784-9245
[d] https://orcid.org/0000-0002-2435-3543
[e] https://orcid.org/0000-0002-6487-5813
[f] https://orcid.org/0000-0003-2048-2468
[g] https://orcid.org/0000-0003-1343-3553

Wizenty, P., Ponce, F., Rademacher, F., Soldani, J., Astudillo, H., Brogi, A. and Sachweh, S.
Model-Driven End-to-End Resolution of Security Smells in Microservice Architectures.
DOI: 10.5220/0012671700003711
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 14th International Conference on Cloud Computing and Services Science (CLOSER 2024)*, pages 204-215
ISBN: 978-989-758-701-6; ISSN: 2184-5042
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

atizes smell detection and provides users with an interactive mechanism for smell resolution across the concerned MSA components. MSA security smells have been recently proposed in (Ponce et al., 2022b), and how to automatically detect and resolve them is still an open issue (Cerny et al., 2023). Our approach relies on the previous work on MSA security smells, as well as stakeholder-oriented Model-Driven Engineering (MDE) (Combemale et al., 2017) of MSAs (Rademacher et al., 2020b). More precisely, it first maps the source code of existing MSAs to MSA-specific architecture models based on the Language Ecosystem for Modeling Microservice Architecture (LEMMA) (Rademacher et al., 2020b). These models are then validated to automatically detect security smells and make them visible to MSA stakeholders, who can then decide, per smell, for a model refactoring that solves the smell. In the final step, a LEMMA-based code generator reflects the refactorings to the original MSA implementation, thereby fixing all places in the application that pertain to a certain smell. As a result, our approach contributes support for the following actions in MSA engineering:

- Automated uncovering of security smells that are scattered across architecture components.
- Automated reporting of those smells via MSA-oriented architecture models that abstract from components' heterogeneity, thus facilitating stakeholder reasoning about smells.
- Deciding for the most suitable smell resolution and subsequent automatic reflection back to the original application code.

We assess the applicability and effectiveness of our approach by executing it on two microservice-based applications. First, we use the student management application to illustrate the steps for resolving security smells. Additionally, we validate our results and demonstrate their applicability on Lakeside Mutual, a standard case study in MSA research (Sorgalla et al., 2021). Our results show the effectiveness of recovering a software application design in architecture models, the capability to detect and resolve security smells in the recovered models, and the capability to resolve the smell in implementing the application.

The rest of this paper is organized as follows. Section 2 provides the necessary background. Section 3 presents our approach. Section 4 assesses its applicability and effectiveness using the Lakeside Mutual case study. Sections 5 and 6 present related work and conclude the paper, respectively.

2 BACKGROUND

We hereafter provide the necessary background on microservice security smells and LEMMA for viewpoint-based microservices modeling.

Smells and Refactorings for Microservice Security. A microservice security smell is a symptom of a potentially bad decision (often unintentional), that can negatively impact the application's security (Ponce et al., 2022b). The effects of security smells can be resolved by refactoring the application without altering the functionality provided to external clients. We hereafter recall two popular MSA security smells that are part of the taxonomy proposed in (Ponce et al., 2022b), and the refactorings that allow to resolve them.

Publicly Accessible Microservices. A microservice of an application is publicly accessible when external clients can directly access it. This increases the application's attack surface and reduces its overall maintainability and usability. Also, if each publicly accessible microservice performs authentication, the full set of a user's credentials is required each time, increasing the likelihood of confidentiality violations (e.g., with the exposure of long-term credentials).

The suggested refactoring is making such microservices accessible *only* through a newly added API Gateway, which would act as an entry point for the application. This would enable centralizing authentication, reducing the application's attack surface and simplifying the authentication itself.

Insufficient Access Control. This smell occurs on the microservices of an application that is not enforcing access control. This can violate the confidentiality of the microservices where access control is lacking, as attackers can trick a service and get data that they should not have access to.

The possible effects of this smell can be resolved by exploiting OAuth 2.0, which would enable microservices to control accesses. OAuth 2.0 indeed provides a token-based access control system that lets a resource owner grant a client access to a particular resource on their behalf.

LEMMA. The Model-Driven Engineering ecosystem LEMMA provides a set of modeling languages to capture concerns in MSA engineering from stakeholder-oriented architecture viewpoints (Rademacher, 2022). MSA models constructed with those languages can be integrated based on an import mechanism that enables referencing between elements of heterogeneous models to support reuse and increase the information content of captured viewpoints in a microservice architecture. The presented approach for security smell resolution in microservice architectures relies on the following LEMMA modeling languages.

Domain Data Modeling Language (DDML). The Domain Data Modeling Language (DDML) of LEMMA addresses the concerns of domain experts and microservice developers in the Domain Viewpoint. To this end, the language supports the construction of *domain models* that cluster the relevant concepts from the application domain. These concepts may be enriched with patterns from Domain-Driven Design (DDD) (Evans, 2004), which is a popular methodology for microservice design (Garriga, 2018, Márquez et al., 2018, Mazlami et al., 2017, Nadareishvili et al., 2016, Newman, 2015). The DDML also implements LEMMA's type system so that domain concepts are usable, e.g., for the typing of parameters of modeled microservice operations. Among others, such typing relationships identify the portion of the application domain on which a microservice operates and for which it is thus responsible.

Technology Modeling Language (TML). LEMMA's TML targets the Technology Viewpoint on microservice architectures and allows for the construction of *technology models* that capture technology decisions related to microservices and their implementation and deployment, e.g., communication protocols and deployment technologies. Additionally, the TML supports the definition of *technology aspects* that apply to specific elements in LEMMA models, e.g., modeled microservices and their interfaces, or infrastructure nodes. Given their flexibility, technology aspects can also be exploited to enable subsequent augmentation of LEMMA models with additional metadata.

Service Modeling Language (SML). LEMMA's SML reifies the Service Viewpoint in MSA engineering and provides modeling concepts to specify microservices, their interfaces, operations, endpoints, and dependencies to other microservices in *service models*. Among others, the SML integrates with the TML so that LEMMA service models can import LEMMA technology models to specify, e.g., protocol-dependent communication endpoints such as HTTP addresses, and the available methods to operate on them.

Operation Modeling Language (OML). LEMMA's OML focuses on MSA's Operation Viewpoint supporting the specification and configuration of microservice containers and infrastructure nodes, e.g., for service discovery, in *operation models*. Similarly to the SML, the OML integrates with the TML to cope with MSA's technology heterogeneity w.r.t. microservice operation and deployment (Knoche and Hasselbring, 2019). More precisely, microservice deployment and infrastructure usage technologies can flexibly be specified in technology models, making them referenceable from operation models.

Next to the model-based description of microservices and their operation, LEMMA also anticipates model processing, and in this context has already been used to foster MSA team integration by *model transformation* (Sorgalla et al., 2021) and increase microservice development efficiency by *code generation* (Rademacher et al., 2020a). In the following, we rely on LEMMA's capabilities in model processing to identify microservices' security smells by *static analysis* of service and operation models and suggest resolution actions by *interactive model refactoring*.

3 END-TO-END SMELL RESOLUTION

This section introduces our approach for end-to-end microservice security smell resolution. Figure 1 depicts the successive steps of resolving security smells using existing or specifically for this approach created LEMMA components.

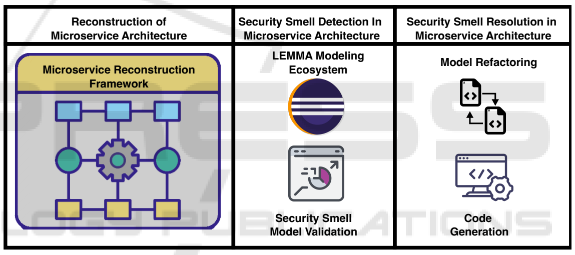

Reconstruction of Microservice Architecture
Security Smell Detection In Microservice Architecture
Security Smell Resolution in Microservice Architecture
Microservice Reconstruction Framework
LEMMA Modeling Ecosystem
Security Smell Model Validation
Model Refactoring
Code Generation

Figure 1: Approach to resolve security smells in MSA.

The first step consists of the automated reconstruction of the MSA of an existing application (Section 3.1). For this purpose, we extended LEMMA's functionalities by integrating the Microservice Reconstruction Framework (MAR) to recover the architecture design of the application with a focus on domain concepts, API management, and deployment specifications as LEMMA models.

In the second step, we use LEMMA's existing modeling ecosystem to modify the reconstructed models from the previous step and extend LEMMA's model validation functionalities with the capability to detect microservice security smells (Section 3.2).

To resolve the security smell in the application architecture, the final step consists of model refactoring and code generation. The model refactoring is a new addition to LEMMA's functionalities for our security smell resolution approach to resolve the smell in the reconstructed models. Moreover, LEMMA's existing code generation functionalities also resolve the security smell in the existing source code.

The rest of this section consists of a detailed description of how our approach enables reconstructing the MSA of an existing application (Section 3.1), detecting the security smells therein (Section 3.2), and resolving them in the source code of the application (Section 3.3) on a concrete MSA-based application.

To illustrate our approach, we will use the MSA-based student management application, depicted in Figure 2 to highlight the separate steps to resolve the security smells and exemplify them with a descriptive example. The application consists of the `Student`- and `ExamService` as functional microservices. Additionally, the services rely on a `Database` and `Service Discovery` to provide their functionalities.

3.1 Reconstruction

Software Architecture Reconstruction (SAR) is a reverse engineering approach to recover the architectural design of an application that is outdated or to ensure conformance between implementation and design (Bass et al., 2013). Figure 3 depicts the structure of our toolchain to reconstruct a technology heterogeneous software application architecture using an ecosystem of MDE tools.

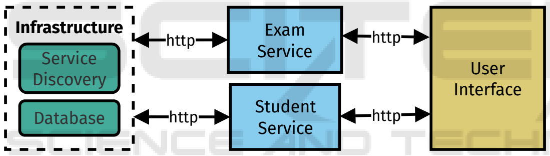


Figure 2: MSA-based student management application.

The `Framework` orchestrates the process of deriving architecture information from static development artifacts, e.g., `Source Code` and `Deployment Specification`. The framework provides a plugin functionality to support a heterogeneous technology stack to reconstruct the architectural design from heterogeneous source code artifacts, e.g., Spring annotations and Docker deployment specifications.

For this purpose, the framework manages the development artifacts and invokes the `Reconstruction Plugins`. The plugins implement functionalities for the technology-specific reconstruction of architectural information. When the framework invokes the plugins, they derive the architecture information and forward the reconstructed architecture information to the framework.

The next step of the process consists of aggregating the reconstructed information from the plugins into a coherent architectural design of the software application. The framework stores the design in a database to enable the possibility of enhancing the design with runtime information, e.g., traces or message

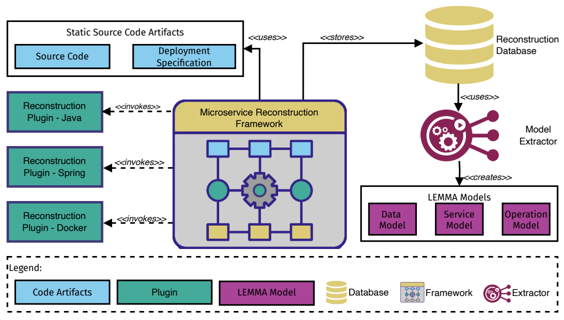


Figure 3: Structure of the reconstruction framework.

```
@Entity
@Table("Student")
public class Student {
    @Id
    @GeneratedValue
    private Id id;
    private String name;
    @ElementCollection
    private List<String> exams;
...}
```

(a) Source code artifact.

```
context Student {
    structure Student<entity, aggregate> {
        Id id,
        string name,
        Exams exams}
    collection Exams { string }
...}
```

(b) LEMMA domain data model.

Figure 4: Example of recovered domain concepts.

broker logging information. The data format consists of the specific concepts for each viewpoint in MSA to store the reconstructed architecture design.

The final step to recovering the software application architecture is to derive viewpoint-specific models from the architectural design stored in the database. Therefore, the `LEMMA Model Extractor` (Rademacher et al., 2020c) uses the information to create LEMMA models from it. The recovered models, with their corresponding viewpoints, address different stakeholders in the software engineering process for MSA, e.g., the domain data model captures domain concepts for service developers and domain experts. Figure 4 shows the source code artifact and the recovered domain data model using LEMMA DDML (Section 2).

Figure 4(a) shows the implementation of the `Student` Java class from the student microservice (Figure 2). The class consists of the two complex attributes `id` and `exams` and the primitive data type attribute `name`. Additionally, the annotations `Entity`, `Table`, and `Embedded/Id` enrich the class with technology-specific information, e.g., for database

```
@RestController
@RequestMapping("/resources")
public class StudentRestApi {
    @Autowired
    private final StudentService service;
    @PutMapping("/student")
    public Student createStudent(
        @RequestBody Student student) {
        return studentService
            .createStudent(student);}}
```

(a) Java / Spring source code artifact.

```
import datatypes from "Student.data"
    as Student
import technology from "Spring.technology"
    as Spring
@technology(Spring)
public functional microservice
    de.fhdo.sep.student.StudentService {
    interface StudentRestApi {
        @Spring::_aspects.Put
        createStudent( sync inout student
            : Student::Student.Student);}}
```

(b) LEMMA service model.

Figure 5: Example of recovered interface specifications.

persistency and to enable the Inversion of Control (IoC) functionalities from the Spring Framework. Therefore, the Spring Reconstruction Plugin uses this information to derive domain concepts from the source code and maps them to the DDD pattern.

In this case, Figure 4(b) features the technology-agnostic recovered LEMMA domain data model. The model contains the `Student` context in accordance with the *Bounded Context* (Evans, 2004) in DDD. The excerpt of the recovered context includes the complex data structure `Student`. The concept is derived from the Java class with the eponymous name. The `Entity` annotation from the Spring Framework, in combination with the name of the Java class, maps to the complex data structure in the domain data model, including the `entity` and `aggregate` pattern from DDD.

For the specification of microservices APIs and service dependencies in the recovered architectural design, our approach uses LEMMA service models to display this information for stakeholders such as service developers in the development process of MSA. Figure 5 shows the Java source code for interface specification from the student microservice, including technology-specific information and the corresponding LEMMA service model.

Figure 5(a) contains the implementation of a REST controller by using the `RestController` and `RequestMapping` annotation. The figure shows the `createStudent` endpoint of the microservice, including URI and method specification. The listing specifies the incoming `student` data type as a `RequestBody` and outgoing `student` parameter.

The service model in Figure 5(b) features the interface specification of the Student microservice derived from the REST controller specification. The figure starts with import statements for domain data and technology models. The `datatype` statement imports the student data model from Figure 4(b), used as data types in the interface specification. Additionally, to enhance the service model with technology-specific information, the subsequent import statement enables using the `Spring` technology model in the microservices interface modeling. Figure 5(b) is then completed by the modeling of the `Student` functional microservice, including the fully qualified name and technology, including the specification of the `StudentRestAPI` interface derive from the Java source code. The specification includes the REST method and URI specification. Furthermore, the imported data types from the student domain model define the incoming and outgoing parameters.

LEMMA operation models contain the deployment specification for microservices and infrastructure components of the application and, therefore, address the concerns of the service developers and operators. Our approach currently supports the recovery of Docker-specific deployment specifications. It can, therefore, be used to recover information from the *docker-compose* file of the student management application, an excerpt of which is in Figure 6(a). Figure 6(a) displays the deployment specification for the infrastructure component `discovery-service` and microservice `student-service`. The DiscoveryService and StudentService deployment specifies, among others, the `image`, `port`, and `depends_on` dependencies in the software application.

The reconstruction process uses these specifications for reconstructing operation models, capturing the architectural design from the operation viewpoint. The operation model in Figure 6(b) describes the deployment of the Student microservice referencing the recovered service model from Figure 5(b). The specification includes `Docker` as the deployment technology for the Student microservice and the runtime dependency to the infrastructure component of a `DiscoveryService` (Figure 6(c)). Since Eureka is part of the Netflix OSS stack and implements the architecture pattern of a *Service Registry* (Bass et al., 2013), the `discovery-service` is reconstructed as an infrastructure node named `DiscoveryService` with the aspect `isServiceRegistry` indicating the reference to the eponymous pattern for microservices.

The result of the end-to-end resolution process of the reference application is the recovered MSA of the student management application. The recovered MSA is in the form of LEMMA models addressing different software engineering viewpoints used to de-

```
services:
  discovery-service:
    build: discovery-service
    image: discovery-service
    ports:
      -"8761:8761"
    volumes:
      -"maven_repo:/root/.m2"
  student-service:
    build: student-service
    image: student-service
    depends_on:
      -discovery-service
    ports:
      -"8081:8081"
```

(a) Docker-Compose specification.

```
@technology(Docker)
container StudentServiceContainer
  deployment technology
    Docker::_deployment.Docker
  deploys StudentService::
    de.fhdo.sep.student.StudentService
  depends on nodes DiscoveryService::
  DiscoveryService {
    default values { basic endpoints {
        Docker::_protocols.http :
          "localhost:8081"; }}}
```

(b) LEMMA student microservice operation model.

```
@technology(Eureka)
DiscoveryService is
  Eureka::_infrastructure.Eureka {
  aspects {Eureka::_aspects
    .isServiceRegistry;}
  default values { port = 8761 }}
```

(c) LEMMA Discovery Service operation model.

Figure 6: Example of our approach to recover deployment specifications from docker artifacts (a) into a LEMMA operation model with microservice deployments (b) and infrastructure components (c).

tect smells, as described in the following section.

3.2 Smell Detection

The detection step uses the LEMMA models recovered in Section 3.1 to identify microservice security smells. Therefore, our approach leverages the expressiveness of LEMMA's aspect functionality (Rademacher et al., 2019) to enhance models with metadata, enabling the possibility to include architecture and security-specific information into the models that can be used to identify security smells. Figure 7 represents the LEMMA models associated with the smell detection process.

Figure 7(a) presents a LEMMA technology model that contains the specification of metadata to enrich operation models with information related to archi-

```
technology Architecture {
  operation aspects {
    aspect isApiGateway
      for infrastructure;
    aspect isServiceRegistry
      for infrastructure; }}
```

(a) LEMMA technology model.

```
technology Zuul {
  infrastructure technologies {
    Zuul {
      operation environments =
        "openjdk:11-jdk-slim";
      service properties
        { string hostname;} }}}
```

(b) LEMMA technology model for Zuul.

```
@technology(Zuul)
@technology(Architecture)
APIGateway is
  Zuul::_infrastructure.Zuul {
    aspects {
      Architecture::_aspects
        .isApiGateway; }}
```

(c) LEMMA operation model for Zuul.

```
@technology(Docker)
container StudentServiceContainer
    deployment technology
      Docker::_deployment.Docker
    deploys de.fhdo.sep.student
     .StudentService
    depends on nodes
      DiscoveryService::DiscoveryService
      APIGateway::ApiGatway {}
```

(d) Resolved LEMMA student model.

Figure 7: LEMMA technology and operation models for security smell detection.

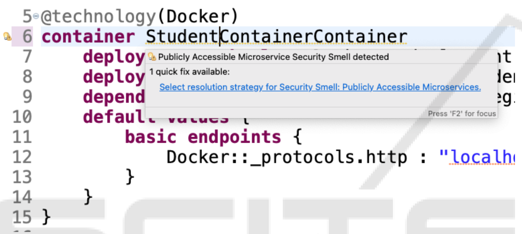
tectural patterns, e.g., *Service Registries* or *API Gateways* (Richardson, 2019), to enable the identification of infrastructural components in the software systems architecture. To identify those patterns in the recovered architecture, the model specifies metadata as `operation aspects`. Specifically, the eponymous infrastructure components `isApiGateway` and `isServiceRegistry`.

Figure 7(b) displays the `Zuul` technology model containing the technological specification for a concrete implementation for an API Gateway. The Zuul technology model is created to be used in an operation model to specify the deployment and operation of an API Gateway.

Figure 7(c) then imports both the architecture and Zuul technology models to define the infrastructure node with the name `APIGateway`. The node uses the infrastructure technology `Zuul` for the implementation specified in Figure 7(b). The operation aspect `isApiGateway` from Figure 7(a) is applied to the node, making it identifiable as an API Gateway.

For security smell identification in MSA, the model validation functionalities of LEMMA allow one to analyze the models regarding the occurrence of security smells, e.g., the absence of infrastructure components that implement specific microservice patterns or the lack of authorization specifications. The model incorporates these patterns by using LEMMA technology aspects.

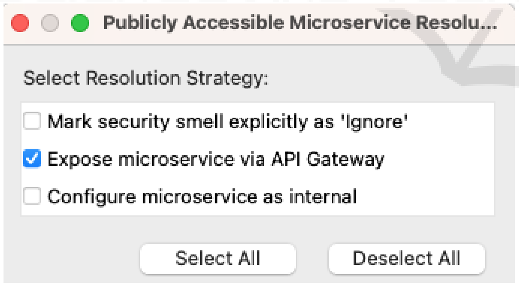
In the case of the deployment specification of the student management application (Figure 8), the model validation gives a warning because the student microservice misses the dependency on an API Gateway, which is normally modeled leveraging LEMMAs abstract concept (c.f. Figure 7a). Therefore, the missing gateway may lead to the *Publicly Accessible Microservices* smell. Figure 9 shows the interactive user interface that presents the smell resolution strategies.



```
@technology(Docker)
container StudentContainerContainer
    deploy                                              nt
    deploy                                              de
    depend                                              eg
    default values {
        basic endpoints {
            Docker::_protocols.http : "localh
        }
    }
}
```

Publicly Accessible Microservice Security Smell detected
1 quick fix available:
Select resolution strategy for Security Smell: Publicly Accessible Microservices.
Press 'F2' for focus

Figure 8: LEMMA Eclipse editor presenting the student microservice operation model with the *Publicly Accessible Microservices* smell detection.



Publicly Accessible Microservice Resolu...

Select Resolution Strategy:

- [ ] Mark security smell explicitly as 'Ignore'
- [x] Expose microservice via API Gateway
- [ ] Configure microservice as internal

Figure 9: LEMMA Eclipse editor presenting the student microservice operation model with the *Publicly Accessible Microservices* smell with the resolution strategy.

LEMMA provides the functionality to refactor the models automatically to resolve the identified microservice smells. Figure 7(d) presents the automatically refactored operation model with the dependency on an API Gateway. In addition to the refactored operation model, the refactoring process also creates technology (b) and operation (c) models if they are not already present in the software architecture, including architectural and infrastructure components.

3.3 Smell Resolution

The security smell resolution uses the refactored LEMMA models to implement the refactoring in the application's source code. Since we have detected the *Publicly Accessible Microservices* security smell in the student management application, our approach provides three strategies as shown in Figure 9.

The first strategy explicitly ignores the security smell, e.g., because the software architect intentionally exposed the microservice, which implements some gateway functionality. The second and third strategies resolve the security smell by exposing the microservices interfaces via an API Gateway or configuring the microservice not to expose it externally (e.g., by removing its external network access configuration). Hereafter, we assume that the user selected the second strategy to integrate an API Gateway into the student management application automatically. The implementation of this refactoring requires the following tasks:

1. Source code generation for the API Gateway.
2. Enable the request routing by the API Gateway.
3. Adapt the deployment specification of the microservices.

To execute these tasks, LEMMA model processing functionalities provide the *Deployment_Base* and *Spring Cloud Zuul* code generators to adapt or create source code artifacts. The Zuul code generator uses the technology model from Figure 7(b) and the operation model from Figure 7(c) to generate the API Gateway implementation, including source code and configuration files based on the Spring Cloud technology stack and using Java as a programming language. Moreover, the *Deployment_Base* generator adapts or creates deployment specifications for the refactored architecture for Docker technology, e.g., *docker-compose* files for service composition and a *Dockerfile* for containerization.

Figure 10(a) shows the source code of the API Gateways implementation using the `@EnableZuulProxy` annotation to enable the routing functionalities of the node. In addition to that, the code generator for the API gateway also generates a configuration file containing the routing information based on the `depends_on` dependencies between the functional service and the API gateway in LEMMA operation models (Figure 7(d)).

For service composition, the *Deployment_Base* generator adapts the *docker-compose* file from Figure 6(a) to address the resolved security smells in the source code. Therefore, the code generator removes the exposure of the ports for the `student` service for

```java
@SpringBootApplication
@EnableZuulProxy
@EnableDiscoveryClient
public class APIGateway {
  public static void main(String[] a){
    SpringApplication.run(APIGateway
      .class, args);}}
```

(a) Java / Spring API Gateway.

```yaml
services:
  student-service:
    depends_on:
      -discovery-service
      -api-gateway
  api-gateway:
    build: api-gateway
    ports:
      -"8080:8080"
    depends_on:
      -discovery-service
```

(b) Adapted Docker-Compose artifact.

```properties
server.port=8080
spring.application.name=APIGateway
zuul.routes.student.path=/STUDENT/**
zuul.routes.student.serviceId=STUDENT
```

(c) Generated API Gateway configuration artifact.

Figure 10: Artifacts created by the *Deployment_Base* and Zuul code generators.

accessing the endpoints. The code generator adds the `api-gateway` component to enable further access to the microservices interfaces. The code generator creates executable source code that is runnable without further configuration, so the end-to-end security smell resolution provides all artifacts needed for the refactoring implementation.

4 VALIDATION

In this section, we validate our approach to model-driven end-to-end resolution of security smells in MSA-based applications. For validation purposes, we use the Lakeside Mutual[1] microservice reference application (Figure 11) to assess our proposed approach since the *Publicly Accessible Microservices* and the *Insufficient Access Control* security smells occur in its implementation. Moreover, we raise research questions (RQ) to investigate the results of the different stages of the presented approach.

[1]https://github.com/Microservice-API-Patterns/LakesideMutual

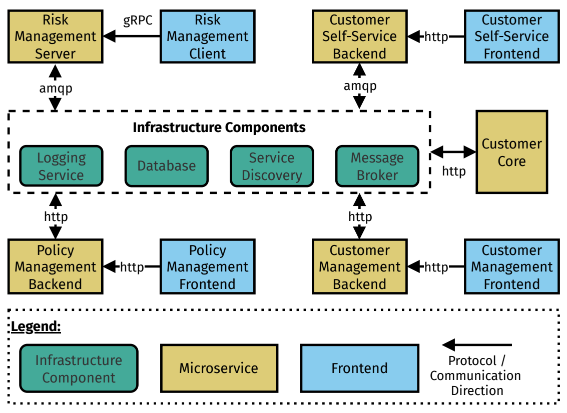

Risk Management Server
gRPC
Risk Management Client
Customer Self-Service Backend
http
Customer Self-Service Frontend
amqp
amqp
Infrastructure Components
Logging Service
Database
Service Discovery
Message Broker
http
Customer Core
http
http
Policy Management Backend
http
Policy Management Frontend
Customer Management Backend
http
Customer Management Frontend
Legend:
Infrastructure Component
Microservice
Frontend
Protocol / Communication Direction

Figure 11: Architecture of the Lakeside Mutual application.

4.1 Research Questions

In the validation process for our presented approach, we aim to answer the following research questions:

RQ1: *How accurate is the reconstructed architecture design of the application?* This research question addresses the accuracy of the reconstructed architecture design related to concepts, e.g., recovered microservices, interfaces, endpoints, data structures, deployment specifications, and infrastructure components.

RQ2: *Could the security smells be detected in the reconstructed models?* The detection of the *Publicly Accessible Microservices* and *Insufficient Access Control* security smell in the reconstructed models is addressed by this research question.

RQ3: *Could the security smell be resolved in the reconstructed models?* We verify if the detected security smell can be resolved in the recovered models capturing the application's architectural design. **RQ4:** *Is resolving the security smell in the application's source code possible?* This research question addresses the end-to-end smell resolution, and we check if the smell is also resolved in the application's source code.

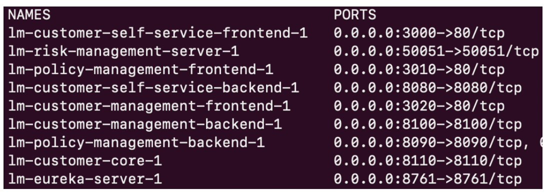
4.2 Validation Implementation

As described in Section 3.1, the end-to-end security smell resolution process starts by recovering an application's MSA, whose modeling in LEMMA is then used to enact smell resolution. Therefore, it is important to assess the effectiveness of the reconstruction step, and here, we measure it for the reference application we considered in our case study.

Table 1 presents the results of the reconstruction process by relying on *Recall*(eq. (1)), *Precision*(eq. (2)), and $F_{measure}$ to measure its overall effectiveness.

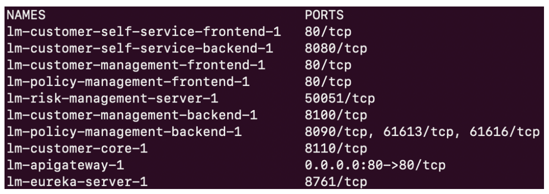
Table 1: Results from the reconstruction process.

Element	Expected	TP	FP	FN	Recall	Precision	$F_{measure}$
Microservices	5	4	0	1	80%	100%	88%
Interfaces	16	14	0	2	87%	100%	93%
Endpoints	61	50	3	8	86%	94%	90%
Data Structures	161	117	29	14	89%	80%	84%
Deployment	5	4	0	1	80%	100%	88%
Infrastructure	2	2	0	0	100%	100%	100%
Sum	250	191	32	26	88%	86%	87%



```
NAMES                              PORTS
lm-customer-self-service-frontend-1   0.0.0.0:3000->80/tcp
lm-risk-management-server-1           0.0.0.0:50051->50051/tcp
lm-policy-management-frontend-1       0.0.0.0:3010->80/tcp
lm-customer-self-service-backend-1    0.0.0.0:8080->8080/tcp
lm-customer-management-frontend-1     0.0.0.0:3020->80/tcp
lm-customer-management-backend-1      0.0.0.0:8100->8100/tcp
lm-policy-management-backend-1        0.0.0.0:8090->8090/tcp,
lm-customer-core-1                    0.0.0.0:8110->8110/tcp
lm-eureka-server-1                    0.0.0.0:8761->8761/tcp
```

Figure 12: Accessibility of running microservices in the original Lakeside Mutual application.



```
NAMES                                PORTS
lm-customer-self-service-frontend-1  80/tcp
lm-customer-self-service-backend-1   8080/tcp
lm-customer-management-frontend-1    80/tcp
lm-policy-management-frontend-1      80/tcp
lm-risk-management-server-1          50051/tcp
lm-customer-management-backend-1     8100/tcp
lm-policy-management-backend-1       8090/tcp, 61613/tcp, 61616/tcp
lm-customer-core-1                   8110/tcp
lm-apigateway-1                      0.0.0.0:80->80/tcp
lm-eureka-server-1                   8761/tcp
```

Figure 13: Accessibility of running microservices in the refactored version obtained with our approach.

$$Recall = \frac{TP}{TP+FN}, \; Precision = \frac{TP}{TP+FP} \qquad (1)$$

$$F_{\text{measure}} = 2 * \frac{Recall * Precision}{Recall + Precision} \qquad (2)$$

The table shows that the reconstruction framework and plugins effectively supported the reconstruction of the microservices composing the Lakeside Mutual application but for the case of the *Risk-Management server*, which is developed in NodeJS, which is not yet supported by our approach, but could be seamlessly integrated with a plugin. A similar consideration applies to the reconstructed data structures, viz., the operation's in- and outgoing data types. The discrepancy in the data structures results from the fact that the plugins do not support the reconstruction of external dependencies, where the source code is not present for the reconstruction, e.g., Spring dependencies. The framework and plugins also recover complex data types as data structures with LEMMA's domain models. Moreover, we recover the deployment specifications for all Spring-based microservices, including infrastructure components, e.g., the Eureka Server or the Spring Boot Admin application.

We used the reconstructed LEMMA models to automatically detect the microservices' security smells present in the application (Section 3.2), by focusing on the two smells currently supported by our approach, viz., *Publicly Accessible Microservices* and *Insufficient Access Control*, which we observed on the application by inspecting its source code and a running instance of the system security smells (Ponce et al., 2022a) (while also proposing resolution strategies for both of them). However, due to the expressiveness of LEMMA's aspect functionality to include metadata in the models, we believe it is possible to extend our current approach to detect and resolve other security smells. For instance, while running the Lakeside Mutual application, we observed that all its microservices were configured to be exposed on different ports of the hosts, therefore all being affected by instances of the *Publicly Accessible Microservices* smell, and possibly subject to direct attacks by external, malicious clients. We also observed that no access control was configured in Lakeside Mutual, meaning that its microservices were all affected by instances of the *Insufficient Access Control* smell.

With the above knowledge in mind, we checked whether the smell detection and refactoring featured by our approach was capable of detecting and resolving the *Publicly Accessible Microservices* and *Insufficient Access Control* smells affecting Lakeside Mutual's microservices by processing the reconstructed LEMMA models of its MSA. Notably, all smell instances were detected by our LEMMA-based framework, and we selected the suggested refactorings to resolve them. More precisely, we first selected the resolution of *Insufficient Access Control* smells, which adapted the LEMMA models by specifying that all microservices should use OAuth 2.0, as recommended in the literature (Ponce et al., 2022b). We then selected the introduction of an API gateway to resolve the *Publicly Accessible Microservices* smells, and this adapted the LEMMA models by introducing the infrastructural components implementing the gateway itself, and configuring the Lakeside Mutual's microservices not to be exposed externally.

The resolution of *Publicly Accessible Microservices* smells was also automatically implemented by our LEMMA-based framework by adapting the Docker-based deployment to reflect what is specified in LEMMA. This can be observed in Figure 13, which shows that (a) Lakeside Mutual's microservices were all publicly accessible in the original version of the application, whereas (b) they were reachable only internally after the refactoring by relying on an API gateway to expose them externally. As a result, Lakeside Mutual's microservices were no more exposed to direct attacks by malicious external clients, with the gateway allowing the enforcement of further security measures, e.g., firewalling or rate limiting.

4.3 Answer to Research Questions

This subsection elaborates on the results of the validation processes related to RQ1 to RQ4.

Answer to RQ1: *The overall accuracy of the recovered architecture design of the Lakeside Mutual application is 87 percent.* Table 1 shows the results of the reconstructed microservice-specific concepts for the architecture design of the software application. The unrecovered concepts, e.g., the missing microservice, occurred due to a yet unsupported technology of the MAR framework. However, due to the extensibility of the plugin functionality, the technology can be integrated seamlessly. The reconstructed LEMMA models, adapted source code and recovery results for the student management application are provided in the auxiliary materials[2].

Answer to RQ2: *LEMMA's validation functionalities can detect both security smells in the reconstructed models.* Our approach extended the validation functions (c.f. Figure 8) of LEMMA to detect the microservice security smell of Insufficient Access Control and Publicly Accessible Microservices in the reconstructed models of capturing the architecture design of the Lakeside Mutual Application.

Answer to RQ3: *LEMMA's model refactoring functionality enables the resolution of both security smells in the reconstructed models.* For the end-to-end security smell resolution approach, we extended LEMMA with the functionality to resolve the detected security smells (c.f. Figure 7) by using a model-to-model transformation (Combemale et al., 2017) and, therefore, adapt the model automatically to resolve the smell by a given smell-specific refactoring strategy.

Answer to RQ4: *The security smells are also resolved in the source code using code generation based on the refactored models (c.f. Figure 13)* The final step of our approach uses code generation to resolve the security smell in the implementation of the application. The security smell of Publicly Accessible Microservices is resolved automatically without manual adaption of the source code. However, to resolve the security smell of Insufficient Access Control, LEMMA functionality provides a guide to enforce sufficient access control by manually adapting the source code.

[2]https://drive.google.com/drive/folders/1W1WE0P_YSc_xx-q_DWHSAaTcRpLngjGG?usp=drive_link

5 RELATED WORK

Microservice security smells have been recently proposed in (Ponce et al., 2022b), and how to automatically detect and refactor them is still an open issue (Cerny et al., 2023). Indeed, to the best of our knowledge, the only available work in this direction are (Dell'Immagine et al., 2023) and (Ponce et al., 2022a). (Dell'Immagine et al., 2023) introduces KubeHound, a tool for detecting security smells in MSAs deployed with Kubernetes. (Ponce et al., 2022b) instead proposes a trade-off analysis to support deciding whether to refactor smells, assuming them to have already been detected. Our approach is, therefore, the first enabling to automatically detect security smells and to support deciding how to refactor them, while also enabling the automatic implementation of chosen refactorings.

Methods and tools for securing MSAs exist, however. For instance, (Ünver and Britto, 2023) proposes Pomegranate, a fully automated test tool suite that can help developers detect security issues in MSAs. Pomegranate essentially encapsulates open-source vulnerability scanning tools into one suite, exploiting them to detect security vulnerabilities in MSAs. We differ from Pomegranate in our objectives, as we focus on detecting security smells in MSAs and on supporting developers in choosing the refactoring for resolving detected smells while also enabling the implementation of chosen refactorings automatically.

Other solutions for securing MSAs are given by the production-ready tools for security analysis, e.g., Kubesec.io, Checkov, and SonarQube. These analysis tools provide validated solutions for vulnerability assessment and security weaknesses detection, which can also be used for microservices applications. Our proposal complements the analyses enacted by the above-listed tools, enabling the detection and refactoring of the microservice security smells proposed in (Ponce et al., 2022b), in addition to the vulnerabilities and security weaknesses they identify. Additionally, production-ready tools such as Fortify and Coverity analyze the source code for the occurrence of code smells, whereas our proposed solution addresses security and architecture smells, resulting from a bad design.

Additional existing approaches provide the possibility to identify and resolve other types of smells for microservices. (Pigazzini et al., 2020), (Walker et al., 2020), and (Soldani et al., 2021) propose two different solutions for detecting architectural smells in MSAs. They both share our baseline idea of starting from smells identified with industry-driven reviews, with (Pigazzini et al., 2020) and (Walker et al., 2020) picking those from (Taibi and Lenarduzzi, 2018), while (Soldani et al., 2021) picking those from (Neri et al., 2020). (Soldani et al., 2021) also shares our baseline idea of using MDE to detect and refactor smells. The main difference between (Pigazzini et al., 2020), (Soldani et al., 2021), and our proposal however relies on the considered types of smells, with (Pigazzini et al., 2020) and (Soldani et al., 2021) focusing on architectural smells. We rather complement their results by enabling the automatic detection and refactoring of microservice smells from (Ponce et al., 2022b).

Similar considerations apply to (Balalaie et al., 2018) and (Haselböck et al., 2017), which both organize information retrieved from practitioners or industry-scale projects into guidelines for designing microservice applications while avoiding the inclusion of well-known architectural smells. We complement these works by enabling the detection of microservices' security smells and refactoring them to resolve their possible effects.

Finally, it is also worth relating our microservice-oriented proposal with existing solutions for detecting smells in classical services. For instance, (Arcelli et al., 2019), (Garcia et al., 2009), and (Sanchez et al., 2015) present three different MDE approaches to detect architectural smells in a service, with (Arcelli et al., 2019) and (Garcia et al., 2009) relying on UML to model services, while (Sanchez et al., 2015) relying on Archery. (Arcelli Fontana et al., 2017) and (Vidal et al., 2015) instead allow to analysis of the source code of a service to detect the smells therein, also supporting refactoring to resolve the occurrence of identified smells. Similarly to the above-discussed approaches, the difference between our proposal and those in (Arcelli et al., 2019), (Arcelli Fontana et al., 2017), (Garcia et al., 2009), (Sanchez et al., 2015), and (Vidal et al., 2015) resides in the considered smells, with our proposal complementing their results by enabling to detect and refactor microservices' security smells.

6 CONCLUSIONS

We have introduced an end-to-end model-driven approach for resolving microservices security smells in MSAs. Our approach recovers the software application architectural design using LEMMA models. The models address different viewpoints in the MSA development process and contain, among others, information about security aspects of Java-based MSAs and to automatically detect the two most recognized security smells for microservices (viz., *Publicly Accessible Microservices* and *Insufficient Access Control*). We demonstrated that our approach enables selecting the refactorings to apply to resolve detected security smells, as well as how it automatically updates LEMMA models and adapts the microservices' source code by implementing the selected refactoring when it is possible or providing detailed information about manual refactoring possibilities. We have also validated our approach in practice, by illustrating its use in a case study based on a third-party application.

For future work, we plan to extend the current implementation into a full-fledged prototype, featuring model-driven detection and refactoring of *all* the microservice security smells from (Ponce et al., 2022b). We also plan to exploit the full-fledged prototype to validate our method on real-world applications, to demonstrate how our approach facilitates the development process of MSAs by providing means for security smell resolution. Finally, we plan to further assist developers by supporting them in deciding whether to refactor a detected security smell, e.g., by integrating with trade-off analyses like that proposed in (Ponce et al., 2022a), and by extending our approach to work with other microservice-related smells, e.g., the architectural smells from (Neri et al., 2020) or (Taibi and Lenarduzzi, 2018). Moreover, we aim to allow for the generic extensibility of our approach in that developers can add new resolutions of security smells based on the abstracted specification of (i) model traversals and element filtering; and (ii) operations for model-based refactorings on traversed elements.

ACKNOWLEDGMENTS

This work was partly supported by the project *FREEDA* (CUP: I53D23003550006), funded by the frameworks PRIN (MUR, Italy) and Next Generation EU.

REFERENCES

Arcelli, D. et al. (2019). Automating performance antipattern detection and software refactoring in UML models. In *26th Int. Conf. on Software Analysis, Evolution and Reengineering (SANER 2019)*, pages 639–643. IEEE.

Arcelli Fontana, F. et al. (2017). Arcan: A tool for architectural smells detection. In *2017 Int. Conf. on Software Architecture Workshops (ICSA-W)*, pages 282–285. IEEE.

Balalaie, A. et al. (2016). Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52.

Balalaie, A. et al. (2018). Microservices migration patterns. *Software Pract. Exper.*, 48(11):2019–2042.

Bass, L., Clements, P., and Kazman, R. (2013). *Software Architecture in Practice*. Addison-Wesley, third edition.

Cerny, T. et al. (2023). Catalog and detection techniques of microservice anti-patterns and bad smells: A tertiary study. *Journal of Systems and Software*, 206:111829.

Combemale, B. et al. (2017). *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. CRC Press, first edition.

Dell'Immagine, G. et al. (2023). Kubehound: Detecting microservices' security smells in kubernetes deployments. *Future Internet*, 15(7).

Evans, E. (2004). *Domain-Driven Design*. Addison-Wesley.

Garcia, J. et al. (2009). Identifying architectural bad smells. In *Proc. of the 2009 Europ. Conf. on Software Maintenance and Reengineering*, pages 255–258. IEEE.

Garriga, M. (2018). Towards a taxonomy of microservices architectures. In *Software Engineering and Formal Methods*, pages 203–218. Springer.

Haselböck, S. et al. (2017). Decision models for microservices: Design areas, stakeholders, use cases, and requirements. In *Software Architecture*, pages 155–170. Springer.

Knoche, H. and Hasselbring, W. (2019). Drivers and barriers for microservice adoption – a survey among professionals in Germany. *Enterprise Modelling and Information Systems Architectures*, 14(1):1–35. German Informatics Society.

Kratzke, N. and Quint, P.-C. (2017). Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study. *Journal of Systems and Software*, 126:1–16. Elsevier.

Márquez, G. et al. (2018). A pattern language for scalable microservices-based systems. In *Proc. of the 12th Europ. Conf. on Software Architecture: Companion Proceedings*, ECSA '18, pages 24:1–24:7. ACM.

Mazlami, G. et al. (2017). Extraction of microservices from monolithic software architectures. In *2017 Int. Conf. on Web Services (ICWS)*, pages 524–531. IEEE.

Nadareishvili, I. et al. (2016). *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly.

Neri, D. et al. (2020). Design principles, architectural smells and refactorings for microservices: a multivocal review. *SICS Software-Intensive Cyber-Physical Systems*, 35(1):3–15.

Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly.

Pigazzini, I. et al. (2020). Towards microservice smells detection. In *Proc. of the 3rd Int. Conf. on Technical Debt (TechDebt 2020)*, page 92–97. ACM.

Ponce, F. et al. (2022a). Should microservice security smells stay or be refactored? towards a trade-off analysis. In *Software Architecture*, pages 131–139. Springer.

Ponce, F. et al. (2022b). Smells and refactorings for microservices security: A multivocal literature review. *J. Syst. Softw.*, 192:111393.

Rademacher, F. (2022). *A Language Ecosystem for Modeling Microservice Architecture*. PhD thesis, University of Kassel.

Rademacher, F. et al. (2019). Aspect-oriented modeling of technology heterogeneity in microservice architecture. In *2019 IEEE Int. Conf. on Software Architecture (ICSA)*, pages 21–30. IEEE.

Rademacher, F. et al. (2020a). Deriving microservice code from underspecified domain models using DevOps-enabled modeling languages and model transformations. In *2020 46th Euromicro Conf. on Software Engineering and Advanced Applications (SEAA)*, pages 229–236. IEEE.

Rademacher, F. et al. (2020b). Graphical and textual model-driven microservice development. In Bucchiarone, A. et al., editors, *Microservices: Science and Engineering*, pages 147–179. Springer.

Rademacher, F. et al. (2020c). A modeling method for systematic architecture reconstruction of microservice-based software systems. In *Enterprise, Business-Process and Information Systems Modeling*, pages 311–326. Springer.

Richardson, C. (2019). *Microservices Patterns*. Manning Publications.

Sanchez, A. et al. (2015). Modelling and verifying smell-free architectures with the archery language. In *Software Engineering and Formal Methods*, pages 147–163. SEFM 2015, Springer.

Soldani, J. et al. (2021). The μtosca toolchain: Mining, analyzing, and refactoring microservice-based architectures. *Software Pract. Exper.*, 51(7):1591–1621.

Sorgalla, J. et al. (2021). Applying model-driven engineering to stimulate the adoption of devops processes in small and medium-sized development organizations: the case for microservice architecture. *SN Computer Science*, 2(6):459.

Taibi, D. and Lenarduzzi, V. (2018). On the definition of microservice bad smells. *IEEE Software*, 35(3):56–62.

Ünver, B. and Britto, R. (2023). Automatic detection of security deficiencies and refactoring advises for microservices. In *2023 IEEE/ACM Int. Conf. on Software and System Processes (ICSSP)*, pages 25–34.

Vidal, S. et al. (2015). JSpIRIT: A flexible tool for the analysis of code smells. In *34th Int. Conf. of the Chilean Computer Science Society (SCCC 2015)*, pages 1–6. IEEE.

Walker, A., Das, D., and Cerny, T. (2020). Automated code-smell detection in microservices through static analysis: A case study. *Applied Sciences*, 10(21).