# Towards Resolving Security Smells in Microservices, Model-Driven

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*

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

Abstract: Resolving security issues in microservice applications is crucial, as many IT companies rely on microservices to deliver their core businesses. Security smells denote possible symptoms of such security issues. However, detecting security smells and reasoning on how to resolve them through refactoring is complex and costly, mainly because of the intrinsic complexity of microservice architectures. This paper presents the first idea towards supporting a model-driven resolution of microservices' security smell. The proposed method relies on LEMMA to model microservice applications by suitably extending LEMMA itself to enable the modeling of microservices' security aspects. The proposed method then enables processing LEMMA models to automatically detect security smells in modeled microservice applications and recommend the refactorings known to resolve the identified security smells. To assess the feasibility of the proposed method, this paper also introduces a proof-of-concept implementation of the proposed LEMMA-based, automated microservices' security smell detection and refactoring.

## 1 INTRODUCTION

Microservice Architecture (MSA) (Newman, 2015) is becoming increasingly popular for building enterprise applications, with companies like Amazon, Netflix, and Twitter already relying on MSAs to deliver their core businesses (Thönes, 2015). The popularity of microservices is mainly due to their *cloud-native* (Gannon et al., 2017) nature, which enables microservice applications to fully exploit the potentials of cloud computing, and to the fact that microservices natively align with the increasingly popular DevOps practices (Balalaie et al., 2016).

MSA is essentially service-oriented architecture, adhering to an extended set of design principles that make microservice applications highly distributed, dynamic, and fault-resilient (Zimmermann, 2017). As a result, MSA inherits the traditional security concerns and practices for service-oriented architectures whilst also bringing up new security challenges, including the so-called *security smells*, which were first elicited in (Ponce et al., 2022b).

Microservice security smells are possible symptoms of (typically unintentional) bad design decisions, which can negatively affect the overall application's security (Ponce et al., 2022b). The impact of microservice security smells can be resolved by applying known refactorings to them, which contribute to securing the application while obviously avoiding altering the functionalities provided to clients. Although refactoring security smells requires effort from development teams, it can help to improve the overall application quality (Bass et al., 2012).

However, detecting microservice security smells and reasoning on how to refactor them is complex, costly, and error-prone. This is mainly due to the intrinsic complexity of MSA itself, which typically results in applications comprising many interacting mi-

[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.
Towards Resolving Security Smells in Microservices, Model-Driven.
DOI: 10.5220/0012049800003538
In *Proceedings of the 18th International Conference on Software Technologies (ICSOFT 2023)*, pages 15-26
ISBN: 978-989-758-665-1; ISSN: 2184-2833

croservices (Ponce et al., 2022a). This paper aims to pave the way toward automated detection and refactoring of microservices security smells, focusing on the two most recognized smells according to (Ponce et al., 2022b). More precisely, we investigate the potential of Model-Driven Engineering (MDE) (Combemale et al., 2017) to this purpose by extending the *Language Ecosystem for Modeling Microservice Architecture* (LEMMA) (Rademacher, 2022), which has been specifically designed to apply MDE in microservices design, development, and operation.

The main contributions of this paper are:

- We introduce a first approach to MDE-based resolution of microservice security smells. More precisely, we extend LEMMA to model security aspects of microservices and propose a method to automatically process LEMMA models to detect the two most recognized security smells and recommend refactorings to resolve their effects.
- We present and discuss the feasibility assessment of our proposed method, which includes a proof-of-concept implementation, and which shows that our method facilitates the detection and refactoring of microservice security issues in LEMMA models of a third-party application.

The rest of this paper is as follows. Section 2 provides background on microservice security smells and LEMMA. Section 3 presents a motivating example. Section 4 introduces our method for detecting and refactoring microservice security smells in LEMMA models, whose proof-of-concept implementation is in Section 5. Sections 6 to 8 provide a discussion of our approach, present related work, and draw some concluding remarks, respectively.

## 2 BACKGROUND

We hereafter provide the necessary background on microservice security smells (Section 2.1) and on LEMMA as a concrete approach towards viewpoint-based modeling of microservices (Section 2.2).

### 2.1 Smells and Refactorings for Microservice Security

In an application, a microservice security smell can indicate a poor decision that may have been made unintentionally, potentially harming the overall security of the application (Ponce et al., 2022b). The effects of security smells can be resolved by refactoring the application or the services therein without altering the functionality provided to clients. Although this process may require effort from development teams, it can help to improve the overall quality of the application (Bass et al., 2012).

We hereafter recall the two security smells for microservices that are most recognized according to (Ponce et al., 2022b), viz., *Publicly Accessible Microservices* and *Insufficient Access Control*. We also recall the refactorings allowing to resolve such smells.

**Publicly Accessible Microservices.** The *Publicly Accessible Microservices* security smell occurs whenever the microservices forming an application are directly accessible by external clients. Each publicly accessible microservice should enact authentication by asking users to provide their full set of credentials. The exposure increases the attack surface, raising the risk of confidentiality violations, and reducing the application's overall maintainability and usability.

The suggested refactoring is to *Add an API Gateway*, which should be used as an entry point for the application. The formerly exposed microservices APIs can then be accessed through the gateway, and authentication can be performed centrally, overall reducing the application's attack surface and simplifying authentication auditing tasks. Development teams can also exploit the gateway to secure an application through a firewall, blocking all external requests to the internal microservices.

**Insufficient Access Control.** The *Insufficient Access Control* smell occurs when the microservices in an application do not enforce access control. This may result in confidentiality violations, as attackers can trick a service and violate data or business functions they should be unable to access. In particular, microservices can be vulnerable to the "confused deputy problem", where attackers deceive a service into revealing information or executing operations that should rather be restricted. Traditional identity control models may also not be enough for microservice applications, where client permissions should be verified at each level of the applications, and each microservice must have a system to accept or reject any particular request automatically.

The suggested refactoring is to *Use OAuth 2.0* as an access delegation framework that enables suitable access control to the microservices forming an application. OAuth 2.0 is indeed a token-based access control system that lets a resource owner grant a client access to a particular resource on their behalf.

## 2.2 Viewpoint-Based Microservice Modeling with LEMMA

MSA engineering is inherently complex, and entails challenges in architecture design, implementation, and operation (Soldani et al., 2018), as well as in development organization (Di Francesco et al., 2018; Knoche and Hasselbring, 2019). For example, the capabilities provided by microservices need to be optimized so that they are distinct without being too fine-grained, thereby allowing targeted scaling and preventing overly communication simultaneously (design challenge). On the other hand, given microservices' maximization of independence, developers, and software architects are concerned with a potentially high degree of technology heterogeneity that increases maintainability costs and learning curves (implementation challenge). Additionally, MSA adoption can only be successful by orienting the structure of the development organization towards the software architecture, which involves the alignment of interaction relationships between MSA teams to those of microservices intending to streamline knowledge sharing (organizational challenge).

To tackle these challenges, researchers studied the application of MDE (Combemale et al., 2017) to facilitate MSA engineering by a targeted abstraction from complexity. The resulting modeling approaches are however very heterogeneous, and the majority focuses on a single phase in MSA engineering, i.e., microservice design (Kapferer and Zimmermann, 2020; Hassan et al., 2017), implementation (Terzić et al., 2018; JHipster, 2023), or operation (Soldani et al., 2021; JHipster, 2023). As a result, these approaches neither allow an integrated, model-based expression of concerns across different phases in MSA engineering, nor do they support a consistent sharing of knowledge among MSA teams.

To cope with these drawbacks in model-driven MSA engineering, the MDE ecosystem LEMMA was conceived (Rademacher, 2022). LEMMA provides a set of modeling languages to capture various concerns in MSA engineering from stakeholder-oriented architecture viewpoints. 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. The presented approach for security smell resolution in microservice architectures relies on the following LEMMA modeling languages.

**Technology Modeling Language (TML).** LEMMA's TML targets the Technology Viewpoint on microservice architectures. That is, it allows for the construction of *technology models* that capture technology decisions related to microservices and their 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. To this end, it provides modeling concepts to specify microservices, their interfaces, operations and endpoints 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 together with the available methods to operate on them.

**Operation Modeling Language (OML).** LEMMA's OML focuses on MSA's Operation Viewpoint. Consequently, the language supports the specification and configuration of microservice containers and infrastructure nodes, e.g., for service discovery or load balancing, 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, technologies for microservice deployment and infrastructure usage can flexibly be specified in technology models, making them referenceable from operation models.

Besides model-based description of microservices and their operation, LEMMA also anticipates model processing, and 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., 2020). 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 MOTIVATING EXAMPLE

This section introduces a motivating example to explain our approach toward model-driven microservice security smell resolution. The example application represents a widely-used fictional insurance company called Lakeside Mutual. The application's source

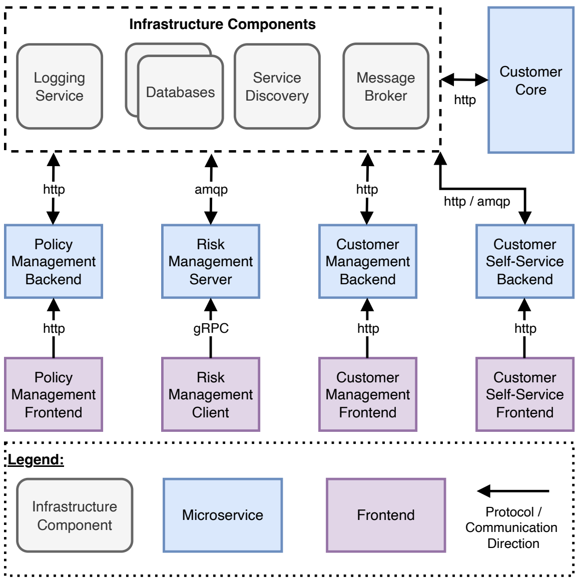

Infrastructure Components
Logging Service
Databases
Service Discovery
Message Broker
Customer Core
http
http
amqp
http
http / amqp
Policy Management Backend
Risk Management Server
Customer Management Backend
Customer Self-Service Backend
http
gRPC
http
http
Policy Management Frontend
Risk Management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

Figure 1: Lakeside Mutual MSA example architecture.

code is publicly accessible on Github[1], and is a common example in MSA-based research (Kapferer and Zimmermann, 2020; Panichella et al., 2021; Sorgalla et al., 2021). Figure 1 depicts the underlying architecture of the motivating example application, including functional microservices, infrastructure components, and frontend services.

Each functional microservice (i.e. `PolicyManagementBackend`, `RiskManagementServer`, `CustomerManagementBackend`, and `CustomerSelfServiceBackend`) features a standalone frontend component for user interaction. Infrastructure components, e.g., `MessageBroker` and `ServiceDiscovery`, provide support functionalities for interservice communication using `HTTP` and `AMQP` for synchronous and asynchronous requests, respectively.

Furthermore, the application data handling has a dedicated `Database` per functional microservice to foster loose coupling among services and enable the independent development of the application's microservices. Finally, the `Logging Service` provides monitoring and operational functionalities.

Being this a demonstrator application, not pretending to be an engineered and production-ready system, we may expect it to feature several security issues. Indeed, if manually analyzing the application in terms of design decisions, technologies used, and security configurations, we may discover such issues, which include the following two:

- The APIs of all functional microservices of the Lakeside Mutual application are publicly exposed. This exposure is an occurrence of the *Publicly Accessible Microservice* smell, which denotes an increased security violation risk by extending the attack surface. Furthermore, the separately exposed APIs of the microservices also reduce maintainability and usability.
- The Lakeside Mutual application also includes instances of the *Insufficient Access Control* smell on its API endpoints, which can be exploited for misuse, possibly leading to confidentiality violations and data leaks.

By knowing this, we may consider implementing the *Add an API Gateway* and *Use OAuth 2.0* refactorings, which are known to resolve the above-listed smells. However, with a complex and costly application analysis, smell detection, and refactoring are currently to be done manually. It would be better to have support for automating the detection of security smells and reasoning on how to refactor them, which is precisely our goal in this paper.

## 4 MODEL-DRIVEN SECURITY SMELL RESOLUTION

This section explores our approach towards model-based microservice security smell resolution based on the motivating example from Section 3 and the identified deficiencies in the architecture design and security configuration. Therefore, Section 4.1 provides insights into the aspect-oriented modeling of security smells using LEMMA's TML. Then, Section 4.2 elaborates the detection possibilities of security smells in MSA depending on modeled security aspects via LEMMA's modeling languages, and the potential refactoring strategies for resolving identified security smells. Finally, Section 4.3 describes the user-guided refactoring process with LEMMA.

### 4.1 Modeling Microservice Security Aspects

The modeling of microservice security configurations leverages LEMMA's TML *aspect* concept to incorporate meta-information into a LEMMA service- or operation model, as discussed in Section 2.2. The process of modeling security configurations in the form of LEMMA aspects consists of four sequential activities (M.1 to M.4), which are depicted in an *UML activity diagram* (OMG, 2017) in Figure 2.

The modeling process in LEMMA begins by selecting a security smell (M.1-Figure 2), such as *Pub-*

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

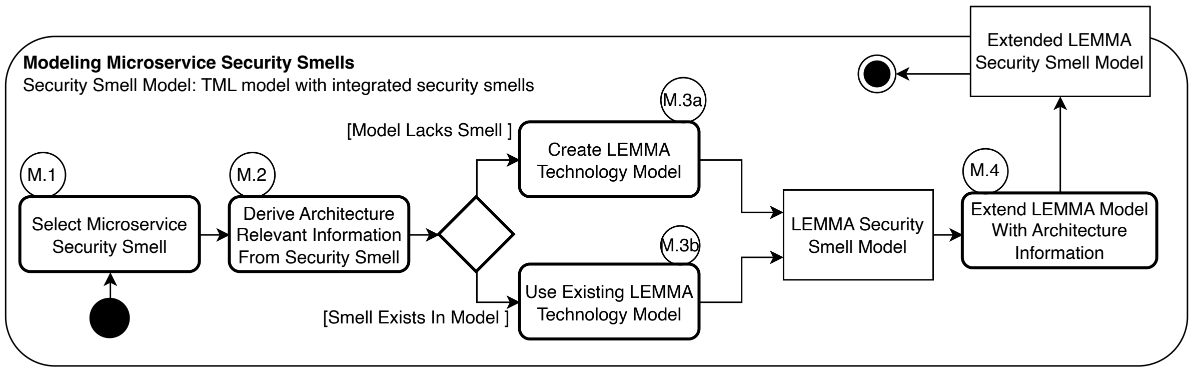

Modeling Microservice Security Smells
Security Smell Model: TML model with integrated security smells
M.1
Select Microservice Security Smell
M.2
Derive Architecture Relevant Information From Security Smell
[Model Lacks Smell ]
[Smell Exists In Model ]
M.3a
Create LEMMA Technology Model
M.3b
Use Existing LEMMA Technology Model
LEMMA Security Smell Model
M.4
Extend LEMMA Model With Architecture Information
Extended LEMMA Security Smell Model

Figure 2: Process of modeling microservices' security aspects with LEMMA.

*licly Accessible Microservices* or *Insufficient Access Control*. The next activity (M.2-Figure 2) involves analyzing the smell to identify the underlying security or architecture decision that gives rise to it. For example, in the case of the *Publicly Accessible Microservices* smell, the decision not to use an API gateway to access all exposed microservices is the root cause of the security smell (Richardson, 2019). Furthermore, an analysis of the API gateway highlights that the corresponding architecture pattern consists of the gateways component, which is responsible for API composition and routing functionalities, and the backend microservices API exposure.

The subsequent activity employs the obtained architecture information to model the security smell using LEMMA's TML. This involves either selecting an appropriate technology model (M.3a-Figure 2) or creating a new one (M.3b-Figure 2) that should be extended with the derived information.

Extending LEMMA's technology model is part of activity M.4-Figure 2, which involves modeling all derived architecture information. The resulting model can be used for automated tests to identify security smells in LEMMA models. The model presented in Listing 1 includes the modeled architecture information derived from the security smells being considered, namely, *Publicly Accessible Microservices* and *Insufficient Access Control*.

Listing 1 Line 1 defines `SecurityAspects` as the name of the LEMMA technology model. The subsequent Lines 2–10 contain `service aspects` that could be applied to the microservice concept of LEMMA's SML. More precisely, the `usesApiGateway` aspect explicitly models that the microservices expose its interface via an API Gateway (Line 3). Line 4 instead introduces the aspect `Authorization` to define a protocol for access delegation, and Line 7 models an aspect to enable role-based access for microservices API endpoints.

Additionally, to model an API Gateway with

Listing 1: LEMMA technology model with derived security smell architecture information for security smell publicly accessible microservices and insufficient access control.

```
1  technology SecurityAspects {
2      service aspects{
3          aspect usesApiGateway for microservices;
4          aspect Authorization for microservices {
5              string protocolName;
6          }
7          aspect Secured for interfaces, operations {
8              string role;
9          }
10     }
11     operation aspects {
12         aspect ApiGateway for infrastructure;
13     }
14 }
```

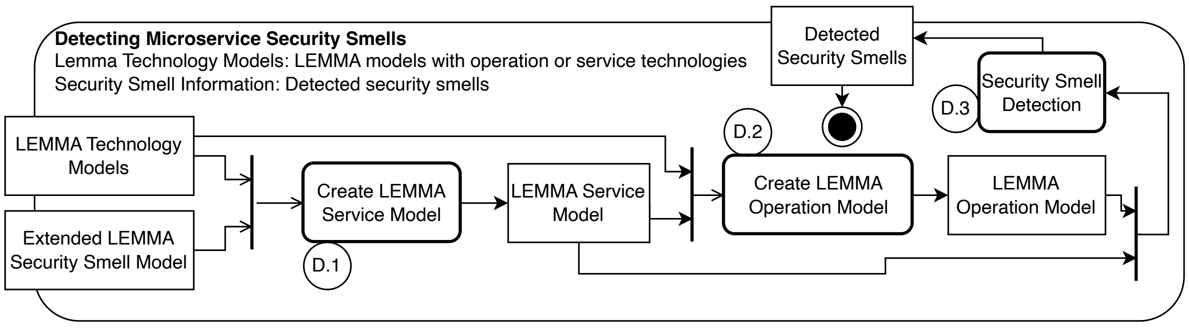
LEMMA's OML, Lines 11–13 of Listing 1 specify an `operation aspect` named `ApiGateway` to identify an infrastructure node in LEMMA's operation models as an API Gateway.

## 4.2 Detecting Security Smells

This section builds upon the previously created technology models (c.f. Section 4.1) as an indicator to detect security smells in LEMMA's service or operation models for microservices. Figure 3 depicts the detection process and includes activities D.1 to D.3. Activity D1 leverages the `Extended LEMMA Security Smell Model` in association with additional `LEMMA-Technology Models`, e.g., Spring[2] or Java[3] models to specify the microservices' API and dependencies to other components of the systems.

For example, Listing 2 depicts a concrete LEMMA service model from the motivating example (c.f. Section 3) for the *Customer Core* microservice. Lines 1 to 2 in listing 2 comprises LEMMA's model import functionality. In this case, the service model imports the `domain` LEMMA domain data model (Rademacher, 2022) for API specification. Additionally, the `spring` and `securityAspect` LEMMA

[2]https://spring.io
[3]https://www.docker.com



Detecting Microservice Security Smells
Lemma Technology Models: LEMMA models with operation or service technologies
Security Smell Information: Detected security smells
Detected Security Smells
Security Smell Detection
D.3
D.2
LEMMA Technology Models
Extended LEMMA Security Smell Model
Create LEMMA Service Model
D.1
LEMMA Service Model
Create LEMMA Operation Model
LEMMA Operation Model

Figure 3: Activities of detecting microservices' security aspects with LEMMA.

Listing 2: LEMMA service model for the Customer Core microservice from the motivating example.

```
1  import datatypes from "customerCore.data" as domain
2  import technology from "spring.technology" as spring
3
4  @technology(spring)
5  @spring::_aspects.ApplicationName("CustomerCore")
6  @spring::_aspects.Port(8080)
7  public functional microservice com.lakeside.CustomerCore {
8      @endpoints(java::_protocols.rest: "/cities";)
9      interface cityStaticDataHolder {
10         ---
11         Get the cities for a particular postal code.
12         @required postalCode the postal code
13         ---
14         @endpoints({spring::_protocols.rest: "/{code}";})
15         @spring::_aspects.GetMapping
16         getCitiesForPostalCode(
17             sync in code : string,
18             sync out cities : domain::customerCore
19                 .CitiesResponseDto
20         );}
21         ...
22 }
```

Listing 3: LEMMA operation model for the Customer Core microservice from the motivating example.

```
1  import microservices from "customerCore.services"
2      as customerCore
3  import technology from "deploymentBase.technology"
4      as deploymentBase
5  import technology from "protocol.technology" as protocol
6  import nodes from "infrastructure.operation" as
       infrastructure
7
8  @technology(deploymentBase)
9  @technology(protocol)
10 container CustomerCoreContainer
11     deployment technology deploymentBase::_deployment.Docker

12     deploys customerCore::com.lakeside.CustomerCore
13     depends on nodes
14         infrastructure::ServiceDiscovery,
15         infrastructure::H2Database {
16         default values {
17             basic endpoints { protocolTechnology::
18                 _protocols.rest: "http://localhost:8110"; }
19         }
20 }
```

technology model is included to enhance the service model with technology and architecture-related information, e.g., `spring` as a framework for the realization of the Customer Core microservice in Line 4.

Lines 5–6 (listing 2) then define service-specific properties, which relate to Spring-based configurations for the microservice application name and operating port. Line 7 specifies the fully qualified name `com.lakeside.CustomerCore` of the microservice, including an excerpt of its API definition in Lines 8–20. Specifying the microservices' API by defining the resource URI in Line 8 and the name in Line 9.

The `cityStaticDataHolder` interface from listing 2 specifies a single endpoint in Lines 10–20. The specification begins with a comment describing the endpoint's general functionality and required parameters, followed by the endpoint-specific extension of the interfaces URI in Line 14 for the `rest` over HTTP protocol. Line 15 then adds the Spring framework-specific aspect `GetMapping` to the endpoint, indicating that the endpoint supports access via the HTTP-method *get*. Lines 16 to 20 finally specify the body definition for the endpoint containing incoming and outgoing parameters, including their corresponding data types.

The next activity, D.2 (c.f. Figure 3), consists of the operation model creation using LEMMA's OML. The operation model contains the deployment and operation specifications. Listing 3 defines the operation model for the Customer Core microservice with dependencies to infrastructure components, e.g., databases or Service Discoveries. The listing begins with an import of different LEMMA models from Line 1 to 6. The first import consists of the Customer Core service model (c.f. Listing 2) to specify the microservice deployment. Lines 3–5 contain the import statements for the `deploymentBase` and `protocol` technology models, with the technologies necessary for deployment specification.

Listing 3 then continues with Lines 8–9 assigning the imported technologies to the `CustomerCoreContainer` (Line 10) to enable their usage in the deployment specification for the container. The container in LEMMA's OML is a component that encapsulates all relevant information for deploying a specific microservice. Therefore, in Line 11, the deployment technology `docker` is assigned to the container, as well as the deployed `CustomerCore` microservice (Line 12).

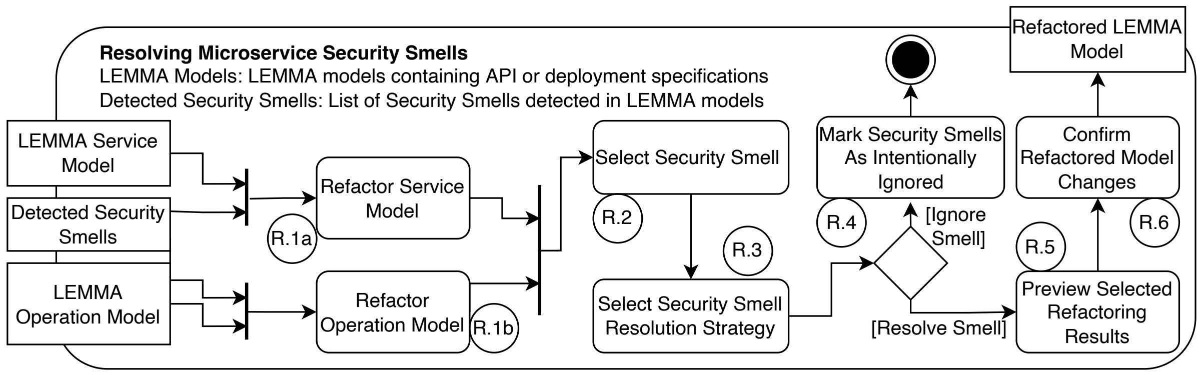

Resolving Microservice Security Smells
LEMMA Models: LEMMA models containing API or deployment specifications
Detected Security Smells: List of Security Smells detected in LEMMA models
LEMMA Service Model
Detected Security Smells
LEMMA Operation Model
Refactor Service Model
R.1a
Refactor Operation Model
R.1b
Select Security Smell
R.2
Select Security Smell Resolution Strategy
R.3
Mark Security Smells As Intentionally Ignored
R.4
[Ignore Smell]
[Resolve Smell]
R.5
Preview Selected Refactoring Results
Confirm Refactored Model Changes
R.6
Refactored LEMMA Model

Figure 4: Activities of resolving microservices' security aspects with LEMMA.

Runtime dependencies to infrastructural components, such as databases or service discoveries, needed for scalability and synchronous service communication, are specified from Line 13 to 15. This includes dependencies to LEMMA infrastructure models, describing the deployment of a *Eureka*[4] service discovery and a *H2*[5] database.

The last part of the listing from Line 16 to 19 defines `default values`, which are used for service operation. In this case, the `basic endpoint` for communication via HTTP and, therefore, as the beginning of the URI extended by endpoint specification of the service model (c.f. Listing 2).

Activity D.3 (c.f. Figure 3) consists of the detection process. For this purpose, the service and operation model is analyzed via model-specific validators. The model validators for security smell can be explicitly enabled in the Eclipse-based model editor of LEMMA. According to the case that the validators detect a security smell in the models, they display a warning in the model editor stating the name of the security smell and possible strategies for resolving it.

## 4.3 Resolving Security Smells

This section introduces the activities followed to resolve the detected security smells. It also presents the user-guided process using the LEMMA's modeling editor. The resolution process, depicted in Figure 4, begins with activity `R.1a` for service and `R.1b` for operation models.

After selecting the model kind, the next activity (`R.2`-Figure 4) involves choosing the security smell that needs to be addressed in the refactoring process. For each security smell, there are different strategies for resolution (Ponce et al., 2022b) (activity `R.3`-Figure 4). For instance, the *Publicly Accessible Microservices* security smell can be resolved by using an API Gateway or by disabling the public exposure of the service. It should be noted that exposing the microservice publicly can also be an intentional design decision that should be marked intentionally (activity `R.4`-Figure 4). As part of the resolution process, deliberately flagging a security smell as "ignored" is a strategy used to notify the user of its presence and acknowledge their design decisions.

Following the selection of the resolution strategy, the next activity is to preview the refactoring results specific to the chosen strategy. LEMMA's modeling editor provides a workflow to guide the user through this process. Finally, the last activity involves confirming the model changes and applying the resolution strategy to the involved models. (activity `R.6`-Figure 4).

To illustrate the results of resolving the *Publicly Accessible Microservices* security smell using the strategy of including an API Gateway for public microservice exposure, an excerpt of the refactored operation model of the *Customer Core* service is shown in Listing 4.

Listing 4: Refactored LEMMA operation model for the *Customer Core* microservice from the motivating example.

```
1 ...
2 @technology(deploymentBase)
3 @technology(protocolTechnology)
4 container CustomerCoreContainer ...
5     depends on nodes
6         infrastructure::ServiceDiscovery,
7         infrastructure::H2Database,
8         infrastructure::APIGateway
9 ...}
```

Generally, the operation model remains unchanged except for adding Line 8. The line specifies a `depends on` dependency to the infrastructural component of an `API Gateway`. Due to the inclusion of the API Gateway in the resolution strategy, a corresponding operation model is also created in the refactoring process of the *Customer Core* model.

Listing 5 specifies the deployment of the API Gateway infrastructure component using Netflix Zuul

[4]https://github.com/Netflix/eureka
[5]https://www.h2database.com

Listing 5: LEMMA operation model for an API Gateway using the Zuul technology.

```
1   import ...
2   @technology(Zuul)
3   APIGateway is Zuul::_infrastructure.Zuul
4       depends on nodes ServiceDiscovery
5       used by services coreService::com.lakeside.CustomerCore,

6       used by nodes coreContainer::CustomerCoreContainer {
7       default values {
8           hostname = "APIGateway"
9           port = 8080
10          apiUri = "eureka:8080"
11      }
12  }
13   ...
```

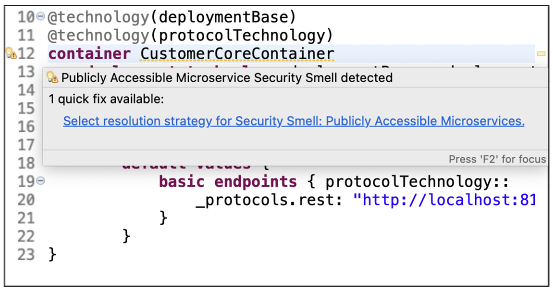
as a concrete technology. Lines 2–3 assign the Zuul technology to the corresponding specification of an `APIGateway`. The next line defines that the API Gateway depends on a `ServiceDiscovery` to fulfill its functionalities. Additionally, the `used by` specifications in Lines 5–6 define that the Customer Core microservices API is exposed via the Gateway. The remaining lines of the listing define `default values`, e.g., the hostname or operation port.

## 5 PROOF OF CONCEPT IMPLEMENTATION

This section presents the proof-of-concept implementation towards the MDE-based resolution of microservices security smells using LEMMA. The proof-of-concept is intended to demonstrate the practical feasibility of model-driven security smell resolution process for publicly accessible microservices (c.f. section 2) described in Section 4, being this the first step towards a full-fledged, validated support for LEMMA-based resolution of security smells in microservice applications.

The proof-of-concept implementation has been developed by suitably extending the current Eclipse-based support for LEMMA. Figure 5 depicts LEMMA's Eclipse-based editor with the opened *Customer Core* Operation model presented in Listing 3. The figure displays a warning in the Eclipse editor indicating that a *Publicly Accessible Microservices* smell was detected in the model. Furthermore, by hovering over the warning, LEMMA's security smell resolution functionality provides the option for resolving the security smell by leveraging the Eclipse Quickfix[6] function, addressing Activity `R.2` of the resolution process.

The selection of the quick fix option starts the resolution process by addressing activity `R.3` (c.f. Figure 4) with the possibility to select a resolution strategy. The strategies are security smell specific, and Figure 6 depicts the option for resolving publicly accessible microservices.



```
@technology(deploymentBase)
@technology(protocolTechnology)
container CustomerCoreContainer
```

Publicly Accessible Microservice Security Smell detected

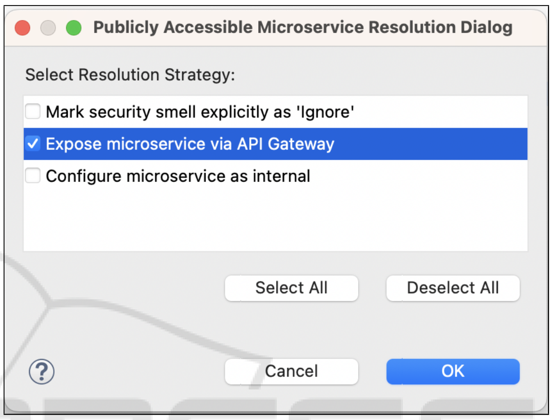
1 quick fix available:

Select resolution strategy for Security Smell: Publicly Accessible Microservices.

Press 'F2' for focus

```
            basic endpoints { protocolTechnology::
                _protocols.rest: "http://localhost:81
            }
        }
}
```

Figure 5: LEMMA Eclipse editor presenting the Customer Core operation model with security smell publicly accessible microservices.


Publicly Accessible Microservice Resolution Dialog
Select Resolution Strategy:
Mark security smell explicitly as 'Ignore'
Expose microservice via API Gateway
Configure microservice as internal
Select All
Deselect All
Cancel
OK

Select Resolution Strategy:

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

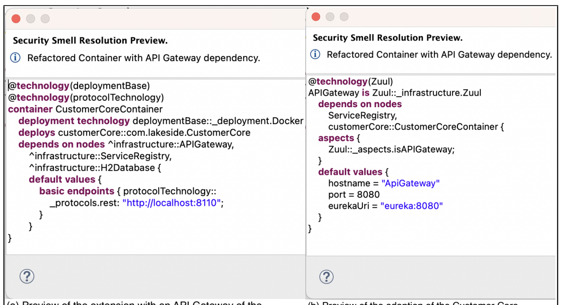
Figure 6: Selection of the security smell-specific strategy for resolving publicly accessible microservices.

Besides the selected option to resolve the security smell via an API Gateway, there is also the option to mark the smell explicitly as to be ignored to disable the warning by a design decision. Additionally, configuring the Customer Core microservice as `internal` is an option to resolve the security smell by disabling public exposure.

The next steps of the process addressing activities `R.5` and `R.6` consist of previewing and confirming the proposed refactoring adaption of the service and operation model kind. The LEMMA editor previews every modified model during the security smell resolution process to guide the user. Figure 7 depicts such a preview for integrating an API Gateway. The preview displays the infrastructure component of an API Gateway that is included in the infrastructure operation model (c.f. Figure 7 (a.)) and the adaption of the Customer Core operation model (c.f. Figure 7 (b.))

The final activity of LEMMA's security smell resolution process is confirming the proposed changes. In addition to the presented security smell resolution functionality, LEMMA provides means towards code

[6]https://www.eclipse.org/Xtext/documentation/310_eclipse_support.html#quick-fixes



Security Smell Resolution Preview.

Refactored Container with API Gateway dependency.

```
@technology(deploymentBase)
@technology(protocolTechnology)
container CustomerCoreContainer
   deployment technology deploymentBase::_deployment.Docker
   deploys customerCore::com.lakeside.CustomerCore
   depends on nodes ^infrastructure::APIGateway,
      ^infrastructure::ServiceRegistry,
      ^infrastructure::H2Database {
      default values {
         basic endpoints { protocolTechnology::
            _protocols.rest: "http://localhost:8110";
         }
      }
}
```

(a) Preview of the extension with an API Gateway of the infrastructure operation model.

Security Smell Resolution Preview.

Refactored Container with API Gateway dependency.

```
@technology(Zuul)
APIGateway is Zuul::_infrastructure.Zuul
   depends on nodes
      ServiceRegistry,
      customerCore::CustomerCoreContainer {
   aspects {
      Zuul::_aspects.isAPIGateway;
   }
   default values {
      hostname = "ApiGateway"
      port = 8080
      eurekaUri = "eureka:8080"
   }
}
```

(b) Preview of the adaption of the Customer Core operation model with API Gateway dependency.

Figure 7: Refactoring previews of the Customer Core and API Gateway operation model for resolving publicly accessible microservices.

generation, e.g., the generation of infrastructure components like API gateways and service discoveries.

The process for resolving *Insufficient Access Control* in microservices includes the same activities as described for the Publicly Available Microservices security smell except for the resolution strategy. The strategy depends on the proposed refactorings for the corresponding microservice security smell (section 2). One solution to resolve the issue of insufficient access control for microservices is to incorporate an authorization protocol such as OAuth2 (Ponce et al., 2022a).

Section 4.2 introduces in listing 2 the Customer Core LEMMA service model. Due to the lack of a specification of an authorization protocol for the microservice, the security of the service suffers from insufficient access control. Integrating the OAuth2 as an authorization protocol resolves the security smell. Listing 1 specifies a LEMMA technology model with an `Authorization` and `Secured` aspect to specify a protocol for role-based access control of microservices. The following listing 6 shows the LEMMA model for the `CustomerCore` microservice with the resolved security smell. To resolve the insufficient access control security, Line 3 imports the `security-Aspcet` technology model and Line 5 applies the security technology to the specified microservice. Line 8 defines `OAuth2` as an authorization protocol to the Customer Core microservice. Additionally, to enable role-based authorization at a microservice endpoint granularity, Line 18 applies the `Secured` aspect to the `getCitiesForPostalCode` endpoint.

## 6 DISCUSSION

Generating software system architecture models using LEMMAs is a manual process that can challenge development teams. Furthermore, as the software system evolves, the models must be updated to reflect the changes in the source code. To overcome this challenge, we plan to integrate Software Architecture Reconstruction (SAR) (Bass et al., 2013) into our approach to derive LEMMA models from source code artifacts automatically. This will eliminate the need for manual model generation and enable the models to be updated automatically as the system evolves.

While our approach currently detects only two of the microservices' security smells identified in (Ponce et al., 2022b), these two smells are among the three most recognized in MSA. Nonetheless, one limitation of our approach is that it does not yet cover all the microservices' security smells. However, we believe that the aspect-oriented modeling capabilities of LEMMA show promising capabilities to detect additional security smells.

The current implementation of our approach focuses on resolving microservices' security smells detected in the LEMMA models, providing developers with awareness of these security smells and offering automated strategies for their resolution. The actual process of resolving security smells in the source code still requires manual intervention from software developers. However, by leveraging LEMMA's code generation functionalities in conjunction with the updated model, it is possible to resolve selected security smells at the implementation level as well.

Listing 6: LEMMA service model for the Customer Core microservice from the motivating example.

```
 1  ...
 2  import technology from "securityAspects.technology"
 3      as securityAspects
 4  @technology(spring)
 5  @technology(securityAspects)
 6  @spring::_aspects.ApplicationName("CustomerCore")
 7  @spring::_aspects.Port
 8  @securityAspects::_aspects.Authorization(^protocol="OAuth2")

 9  public functional microservice com.lakeside.CustomerCore {
10      @endpoints(java::_protocols.rest: "/cities";)
11      interface cityStaticDataHolder {
12          ---
13          Get the cities for a particular postal code.
14          @required postalCode the postal code
15          ---
16          @endpoints({spring::_protocols.rest: "/{postalCode}"
                ;})
17          @spring::_aspects.GetMapping
18          @securityAspects::_aspects.Secured("ROLE_USER")
19          getCitiesForPostalCode(...);}
20          ...
21  }
```

## 7 RELATED WORK

Ponce et al. propose a set of microservice security smells in (Ponce et al., 2022b), including refactorings that resolve their effects. Automatically detecting such smells in microservices and refactoring applications to resolve their effects is however still an open issue. Indeed, to the best of our knowledge, the only available work in this direction is that by (Ponce et al., 2022a), which assumes that smells have been identified and proposes a trade-off analysis to decide whether it is worth (or not) to apply a refactoring, depending on how the security smell and refactoring impact the overall application quality.

There are already some methods and tools for analyzing microservices applications' security, which can also be used to detect other security smells. For instance, (Rahman et al., 2019) proposes a static analysis technique to detect security smells in infrastructure-as-code (Morris, 2020) scripts. (Rahman et al., 2019) however,» differs from our proposal in its objectives, as it focuses on detecting security smells for infrastructure-as-code only, while we consider the detection and refactoring microservice security smells for different viewpoints, e.g., the service or operation viewpoint.

Production-ready tools for security analysis, e.g., such as Kubesec.io,[7] Checkov,[8] OWASP Zed Application Proxy (ZAP),[9] and SonarQube.[10] 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 in (Ponce et al., 2022b), in addition to the vulnerabilities and security weaknesses they identify.

Additional existing approaches provide the possibility to identify and resolve architectural smells for microservices. (Pigazzini et al., 2020) and (Soldani et al., 2021) propose two different solutions for detecting architectural smells in microservice applications. They both share our baseline idea of starting from smells identified with industry-driven reviews, with (Pigazzini 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) actually 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 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 detection and refactoring of microservice security 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 including well-known architectural smells therein. We indeed complement (Balalaie et al., 2018) and (Haselböck et al., 2017) in their effort towards resolving smell occurrences in microservice applications by enabling to detect microservices' security smells and to refactor 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, (Garcia et al., 2009), (Arcelli et al., 2019) and (Sanchez et al., 2015) present three different MDE approaches to detect architectural smells in service, with (Garcia et al., 2009) and (Arcelli et al., 2019) 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 analyze 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 (Garcia et al., 2009), (Arcelli et al., 2019), (Sanchez et al., 2015), (Arcelli Fontana et al., 2017), and (Vidal et al., 2015) resides in the type of considered smells, with our proposal complementing their results by enabling to detect and refactor security smells in microservice applications.

## 8 CONCLUSIONS

We have introduced an approach for model-driven resolution of microservices' security smells based on extending LEMMA to the purpose. Our approach process extended LEMMA functionalities to detect the two most recognized microservices' security smells automatically and to recommend refactoring strategies to resolve their effects.

To assess the feasibility of the proposed approach, we have also presented its proof-of-concept implementation, also discussing how such implementation enables detecting and refactoring microservice security smells in the LEMMA model of an existing third-party application. The presented approach introduced a first step towards automated, MDE-based security smell resolution.

For future work, we plan to follow the exact modeling and analysis methodology to extend the current implementation into a full-fledged prototype featuring a model-driven resolution of security smells occurring in MSA. Furthermore, we plan to include software architecture reconstruction to automatically

[7]https://kubesec.io

[8]https://www.checkov.io

[9]https://owasp.org/www-project-zap/

[10]http://sonarqube.org/

derive security-aware LEMMA models based on the current implementation of the software system to ease the integration of our approach in MSA development.

We also plan to exploit the full-fledged prototype to validate and evaluate our method on real-world applications, with the goal of demonstrating how our approach facilitates the development process of MSA by providing means for security smell resolution.

In this perspective, we also plan to assist developers in deciding whether/how to refactor a security smell detected in an MSA, e.g., by integrating our full-fledged prototype with trade-off analyses and code generation functionalities to automatically resolve the security smell also on the level of implementation. Additionally, we plan to extend our approach to work with other microservice-related smells, e.g., architectural smells.

## ACKNOWLEDGMENTS

This work was partially supported by ANID under grant *PIA/APOYO AFB180002*, *Instituto de tecnología para la innovación en salud y bienestar, facultad de ingeniería* (Universidad Andrés Bello, Chile), and by the project *hOlistic Sustainable Management of distributed softWARE systems* (OSMWARE, UNIPI PRA_2022_64), funded by the University of Pisa, Italy.

## REFERENCES

Arcelli, D., Cortellessa, V., and Pompeo, D. D. (2019). Automating performance antipattern detection and software refactoring in UML models. In Wang, X., Lo, D., and Shihab, E., editors, *2019 International Conference on Software Analysis, Evolution and Reengineering,* , pages 639–643. SANER 2019, IEEE Computer Society.

Arcelli Fontana, F., Pigazzini, I., Roveda, R., Tamburri, D., Zanoni, M., and Di Nitto, E. (2017). Arcan: A tool for architectural smells detection. In Malavolta, I. and Capilla, R., editors, *2017 IEEE International Conference on Software Architecture Workshops,* , pages 282–285. ICSA 2017 Workshops, IEEE Computer Society.

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

Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D. A., and Lynn, T. (2018). Microservices migration patterns. *Software: Practice and Experience*, 48(11):2019–2042.

Bass, L., Clements, P., and Kazman, R. (2012). *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition.

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

Combemale, B., France, R. B., Jézéquel, J.-M., Rumpe, B., Steel, J., and Vojtisek, D. (2017). *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. CRC Press, first edition.

Di Francesco, P., Lago, P., and Malavolta, I. (2018). Migrating towards microservice architectures: An industrial survey. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 29–38. IEEE.

Gannon, D., Barga, R., and Sundaresan, N. (2017). Cloud-native applications. *IEEE Cloud Computing*, 4(5):16–21.

Garcia, J., Popescu, D., Edwards, G., and Medvidovic, N. (2009). Identifying architectural bad smells. In Winter, A., Ferenc, R., and Knodel, J., editors, *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering,* , pages 255–258, USA. CSMR 2009, IEEE Computer Society.

Haselböck, S., Weinreich, R., and Buchgeher, G. (2017). Decision models for microservices: Design areas, stakeholders, use cases, and requirements. In Lopes, A. and de Lemos, R., editors, *Software Architecture,* , pages 155–170, Cham. Springer International Publishing.

Hassan, S., Ali, N., and Bahsoon, R. (2017). Microservice ambients: An architectural meta-modelling approach for microservice granularity. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 1–10. IEEE.

JHipster (2023). *JHipster Domain Language (JDL). https://www.jhipster.tech/jdl/intro*.

Kapferer, S. and Zimmermann, O. (2020). Domain-driven service design: Context modeling, model refactoring and contract generation. In *Service-Oriented Computing: 14th Symposium and Summer School on Service-Oriented Computing, SummerSOC 2020, Crete, Greece, September 13-19, 2020 14*, pages 189–208. 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.

Morris, K. (2020). *Infrastructure as code*. O'Reilly Media.

Neri, D., Soldani, J., Zimmermann, O., and Brogi, A. (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.

OMG (2017). OMG Unified Modeling Language (OMG UML) version 2.5.1. Standard formal/17-12-05, Object Management Group.

Panichella, S., Rahman, M. I., and Taibi, D. (2021). Structural coupling for microservices. *arXiv preprint arXiv:2103.04674*.

Pigazzini, I., Fontana, F. A., Lenarduzzi, V., and Taibi, D. (2020). Towards microservice smells detection. In *Proceedings of the 3rd International Conference on Technical Debt*, page 92–97, New York, NY, USA. TechDebt 2020, Association for Computing Machinery.

Ponce, F., Soldani, J., Astudillo, H., and Brogi, A. (2022a). Should microservice security smells stay or be refactored? towards a trade-off analysis. In Gerostathopoulos, I. et al., editors, *Software Architecture*, pages 131–139. Springer International Publishing.

Ponce, F., Soldani, J., Astudillo, H., and Brogi, A. (2022b). Smells and refactorings for microservices security: A multivocal literature review. *Journal of Systems and Software*, 192:111393.

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

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

Rahman, A., Parnin, C., and Williams, L. (2019). The seven sins: Security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 164–175.

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

Sanchez, A., Barbosa, L. S., and Madeira, A. (2015). Modelling and verifying smell-free architectures with the archery language. In Canal, C. and Idani, A., editors, *Software Engineering and Formal Methods,* , pages 147–163, Cham. SEFM 2015, Springer International Publishing.

Soldani, J., Muntoni, G., Neri, D., and Brogi, A. (2021). The $\mu$tosca toolchain: Mining, analyzing, and refactoring microservice-based architectures. *Software: Practice and Experience*, 51(7):1591–1621.

Soldani, J., Tamburri, D. A., and Heuvel, W.-J. V. D. (2018). The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232. Elsevier.

Sorgalla, J., Wizenty, P., Rademacher, F., Sachweh, S., and Zündorf, A. (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.

Terzić, B., Dimitrieski, V., Kordić, S., Milosavljević, G., and Luković, I. (2018). Development and evaluation of MicroBuilder: a model-driven tool for the specification of REST microservice software architectures. *Enterprise Information Systems*, 12(8-9):1034–1057. Taylor & Francis.

Thönes, J. (2015). Microservices. *IEEE Software*, 32(1):116–116.

Vidal, S., Vazquez, H., Diaz-Pace, J. A., Marcos, C., Garcia, A., and Oizumi, W. (2015). JSpIRIT: A flexible tool for the analysis of code smells. In Marín, B. and Soto, R., editors, *34th International Conference of the Chilean Computer Science Society,* , pages 1–6. SCCC 2015, IEEE Computer Society.

Zimmermann, O. (2017). Microservices tenets. *Computer Science - Research and Development*, 32(3):301–310.