Model-Driven Engineering of Microservice Architectures—The LEMMA Approach

Florian Rademacher, Philip Wizenty, Jonas Sorgalla, Sabine Sachweh, and Albert Zündorf

Abstract This chapter presents LEMMA (Language Ecosystem for Modeling Microservice Architecture). LEMMA enables the application of Model-Driven Engineering (MDE) to Microservice Architecture (MSA). LEMMA mitigates the complexity of MSA by decomposing it along four viewpoints on microservice architectures, each capturing the concerns of different MSA stakeholders in dedicated architecture models. LEMMA formalizes the syntax and semantics of these models with specialized modeling languages that are integrated based on an import mechanism, thus enabling holistic MSA modeling. LEMMA also bundles its own model processing framework (MPF) to facilitate model processor implementation for technology-savvy MSA stakeholders without a background in MDE.

We describe the design and development of LEMMA and exemplify the usage of its modeling languages and MPF for a case study microservice architecture. In addition, we present practical applications of LEMMA for microservice code generation, architecture reconstruction, quality analysis, defect resolution, and establishing a common architecture understanding. A comparison of LEMMA with related approaches reveals that LEMMA has particular strengths in (i) language-level extensibility, allowing model-based reification of architectural patterns; (ii) model processing, by bundling sophisticated code generators and static quality analyzers; and (iii) versatility, making LEMMA applicable in microservice development, operation, architecture reconstruction, and quality assessment.

F. Rademacher (✉)
Software Engineering, RWTH Aachen University, Aachen, Germany
e-mail: rademacher@se-rwth.de

P. Wizenty · J. Sorgalla · S. Sachweh
IDiAL Institute, University of Applied Sciences and Arts Dortmund, Dortmund, Germany
e-mail: philip.wizenty@fh-dortmund.de; jonas.sorgalla@fh-dortmund.de; sabine.sachweh@fh-dortmund.de

A. Zündorf
Software Engineering Research Group, University of Kassel, Kassel, Germany
e-mail: zuendorf@uni-kassel.de

E. Bodden et al. (eds.), *Ernst Denert Award for Software Engineering 2022*,
https://doi.org/10.1007/978-3-031-44412-8_5

1 Introduction

Microservice Architecture (MSA) [65] is an approach to architecting distributed software systems that promotes system decomposition into *microservices*. The notion of microservice comprises all characteristics of a *service*, i.e., it is a functional software component that (i) minimizes dependencies to other components; (ii) clusters coherent business logic; (iii) agrees on *contracts* that specify communication relationships with other components by means of *interfaces*; and (iv) interacts with other components to realize coarse-grained tasks [23]. While MSA emerged from Service-Oriented Architecture (SOA) [17, 23], other than an SOA service, a microservice aims to maximize *service-specific independence*. From the aspects that are concerned by this maximization, the notion of microservice can be defined as follows [6, 13, 14, 17, 44, 64, 65, 93, 101]:

Definition 1 (Microservice) A microservice is a service with the following characteristics:

- It provides a distinct capability to other components, and all of its functionalities address a single concern of either functional or infrastructure nature.
- It is as independent as possible from other components in terms of implementation, data management, testing, deployment, and operation.
- It is fully accountable for its interaction with other components including, e.g., the actual decision for interaction, communication protocol determination, data format conversion, and failure handling. Without a sound technical reason, a microservice supports at most two communication protocols—one for synchronous one-to-one and one for asynchronous one-to-many interactions.
- It is owned by exactly one team. The team is fully responsible for all aspects related to the microservice's design, implementation, and operation.

Starting from these characteristics, MSA is expected to benefit the architectures of distributed software systems in several ways. First, microservices can improve performance efficiency, and especially *scalability* [42], making it possible to scale heavily frequented functionalities independently and horizontally [19]. Second, microservices may have a positive impact on maintainability and, more precisely, *modifiability* [42], because they facilitate isolated replacement of functionality as long as interfaces remain stable [65]. Third, MSA can increase the *testability* [42] of software systems by demanding stand-alone component executability.

While performance efficiency and maintainability are the most important quality attributes of MSA and key drivers for its adoption [17, 119], microservices can benefit further quality attributes [42] such as (i) *reliability*, due to each microservice being expected to include its own failure handling mechanisms for preventing failure cascades across service boundaries [13, 65]; (ii) *portability*, by deploying microservices using lightweight virtualization technologies like containers [9, 18, 111]; and (iii) *compatibility*, as independent executability and standardized communication protocols foster interoperability and gradual migration of legacy systems toward MSA [13, 119].

However, these benefits come at the cost of an increased complexity that affects architecture design, implementation, and operation [109]. For example, granularity determination is a major challenge in MSA design as too fine-grained microservices induce frequent interactions and thus network overhead [119]. Concerning implementation, MSA aggravates technology management by supporting dedicated technology choices per service and delegation of decisions for technology stacks to microservice teams [52]. The resulting increase of *technology heterogeneity* [65] also concerns operation because the corresponding infrastructure of a microservice architecture usually consists of loosely coupled components for diverse tasks like service discovery, API provisioning, load balancing, and monitoring [7].

Model-Driven Engineering (MDE) [15] is an approach to software engineering that leverages *models* as a means to abstract from selected details of a software system to mitigate complexity. More precisely, MDE focuses on the systematic construction, evolution, and maintenance of software models, and making them actionable within one or more phases of the software engineering process. For a certain set of purposes, models can then act as substitutes of more complex artifacts. For instance, models may abstract from implementation details of the conversion between data-format-specific network messages and data-format-agnostic in-memory objects. Yet they can enable the automated derivation of source code for this purpose [94]. On another note, models are well suited to reify structures of software systems and facilitate reasoning about them, e.g., for quality assessment and improvement [16].

As an orthogonal approach to software architecting that strives for purposeful complexity mitigation, MDE is a predestined means for the description, development, and analysis of complex software systems [28, 104]. Indeed, it has successfully been applied in different domains of software architectures such as cyber-physical systems [62], Industry 4.0 [127], Internet of Things [51], and SOA [2]. Hence, it is evident to investigate the applicability of *MDE-for-MSA* [31].

This chapter presents recent findings of this investigation by (i) summarizing the main results of a corresponding dissertation [93], which manifested in the Language Ecosystem for Modeling Microservice Architecture (LEMMA) [108]; and (ii) showing how LEMMA stimulates ongoing research on MDE-for-MSA beyond the dissertation.

The remainder of the chapter is structured as follows. Section 2 presents background information on MDE-for-MSA. Section 3 describes LEMMA's design and implementation. Section 4 focuses on its applications, e.g., for microservice code generation, architecture reconstruction, and defect resolution. Sections 5 and 6 compare LEMMA with related works and conclude the chapter.

2 Preliminaries

This section describes challenges in MSA engineering (Sect. 2.1), the MDE paradigm (Sect. 2.2), and the adoption of MDE to tackle MSA engineering challenges (Sect. 2.3).

2.1 Challenges in Microservice Architecture Engineering

Following the *taxonomy for pains of microservices* by Soldani et al. [109], we summarize challenges in MSA engineering along the dimensions Design (Sect. 2.1.1), Implementation (Sect. 2.1.2), and Operation (Sect. 2.1.3). Given MSA's impact on development organizations [64], we also consider the Organization dimension (Sect. 2.1.4).

2.1.1 Design Challenges

The *identification of microservices* is pivotal in MSA design [29, 109, 119]. It entails the decomposition of functionality and is closely related to granularity determination (see below). Domain-Driven Design (DDD) is a popular methodology for microservice identification [24, 30, 56, 57, 64, 65]. It provides model-based techniques and patterns to identify coherent parts in an application domain and eventually derive *bounded contexts* from them. A bounded context clusters coherent domain concepts, their structures, and relationships in a *domain model* [24]. Similar to microservices, bounded contexts gather coherent functionality, belong to one team, and require interactions via well-defined interfaces. Despite the perceived closeness of DDD and MSA [65], the adoption of the former in the context of the latter is often considered complex [13, 29] and additional effort when domain models act as mere documentation artifacts [24].

Determining the optimal granularity of a microservice is a major challenge in MSA design [109, 119]. Besides the vague suggestion to align a microservice to a distinct capability (Definition 1), there exist no broadly accepted guidelines on how to tailor a microservice's responsibilities. Additionally, the independence of microservice teams fosters divergent intuitions of microservice granularity and, in the worst case, may result in a centralized architecture team that balances varying granularities by frequent refactoring [13]. On the other hand, certain microservices may intentionally be more coarse-grained than others to decrease network load, eliminate interaction dependencies, or reduce the number of microservices [13, 18].

By contrast to SOA, MSA considers *APIs as contracts* [128], thereby rendering the formal specification of interactions and *explicit contract negotiation* [66] redundant. Instead, the interaction relationship between two microservices concludes an *implicit service contract*, which reduces design complexity. As a drawback, microservices are confronted with API versioning and assuring consumer compliance [109]. Moreover, the waiver of explicit contracts fosters ad hoc communication and thus the accidental occurrence of cyclic service dependencies [118].

2.1.2 Implementation Challenges

As already mentioned (Sect. 1), MSA can increase the technology heterogeneity of a software system. While it may be beneficial that each microservice can rely on those implementation technologies that are the most suitable for its capability [65], this level of freedom in technology choices incurs risks for increased technical debt, additional maintainability costs, and steeper learning curves for new team members [13, 52, 118].

Moreover, MSA's emphasis on loose coupling also leads to a decoupling of technical concerns [38]. Hence, technology management is additionally aggravated due to an increased number of *technology variation points* [95] like the following:

- **Programming languages:** The programming language of a microservice is opaque to clients. Java, JavaScript, and C# are among the most popular service programming languages [106]. However, certain specifics of a service's capability can motivate the adoption of an alternative programming language. For instance, the built-in support for data collection handling and the availability of sophisticated frameworks for scientific computing or time series processing make Python a viable choice for corresponding microservices [46, 54, 95].
- **Database management systems (DBMSs):** To decrease coupling, each microservice should have its own database [63, 102]. A service's capability may also favor DBMS mechanisms like NoSQL or graphs over the relational paradigm [52].
- **Communication protocols:** A microservice architecture should employ at most two communication protocols (Sect. 1). However, some situations may require more than two protocols, e.g., when gradually modernizing legacy systems [58].
- **Data formats:** The interaction scenario or choice of a communication protocol may impact the selection of a format for data encoding and decoding.

2.1.3 Operation Challenges

Microservices are usually packaged, deployed, and executed in virtualized *containers* [111]. Containers enable the combined deployment of software components and pre-configured runtime environments while being more resource-efficient than virtual machines due to kernel and library sharing with the host operating system. Containers benefit microservices' scalability and portability [18] but typically require additional orchestration platforms, e.g., to fulfill elasticity requirements [40, 52]. These platforms expose microservice architectures to continuous service partitioning and relocation with additional effort to keep track of [109].

Next to container orchestration platforms, MSA requires additional infrastructure components, e.g., for service discovery, API provisioning, load balancing, and monitoring [7]. The loose coupling of these components increases technology heterogeneity on the operation level. Furthermore, each component may have its own requirements w.r.t. configuration and life cycle management [109].

2.1.4 Organizational Challenges

MSA assumes an alignment of the development organization with the software architecture to be effective [4]. A common practice is to decompose homogeneous development organizations into teams, possibly assembled from members with heterogeneous skill sets, of which each is responsible for one or more microservices. Consequently, MSA fosters DevOps [64] and thus faces challenges like establishing and maintaining a *collaborative culture*, *automation*, and *knowledge sharing* [55].

2.2 *Model-Driven Engineering*

To unfold its potential for complexity mitigation, MDE anticipates systematic model construction, evolution, and maintenance. *Modeling languages* specify models' *syntaxes* and *semantics* [15]. The syntax consists of an *abstract syntax* and one or more *concrete syntaxes*. The former defines modeling concepts' structures, relationships, and tool-internal representation. The latter determine user-facing notations of modeling concepts, e.g., as graphical constituents of box-and-line diagrams or grammar-based textual strings. Modeling language syntaxes may impose constraints on model well-formedness, thus contributing to the definition of language-specific model validity. The semantics of a modeling language assigns meaning to modeling concepts and their instantiation as model elements; and can restrict the set of valid models even further [37].

Model processors turn models into actionable software engineering artifacts [15]. *Code generation* is often perceived to drive MDE adoption because of an expected increase of development productivity [123]. However, there exists a plethora of other model processing approaches with relevance to software architecting, e.g., *reverse engineering* [117] and *static model analysis* [15]. Most of these approaches resort to *model transformation* [15], i.e., the (semi-) automated conversion of one or more source models into one or more target models based on *transformation rules* [59]. The syntaxes of source and target models may differ, e.g., when model elements are transformed into programming language constructs (code generation) or implementation artifacts are transformed into models (reverse engineering).

2.3 *Employing Model-Driven Engineering to Cope with Challenges in Microservice Architecture Engineering*

Table 1 maps MDE means (Sect. 2.2) to MSA engineering challenges (Sect. 2.1) and substantiates our hypothesis that MDE-for-MSA can cope with MSA's complexity.

Sections 2.3.1 to 2.3.4 describe the mapping per MSA engineering dimension.

Table 1 Mapping of supportive MDE means to MSA engineering challenges [93]

Dimension	Challenge	Summary	MDE means
Design	C.1	Manage[a] services' granularities	Modeling languages
	C.2	Facilitate domain-driven service identification	Modeling languages
	C.3	Increase the value of domain models	Model processing
	C.4	Manage services' APIs	Modeling languages
	C.5	Cope with cyclic service dependencies	Context conditions, static analysis
Implementation	C.6	Manage technology heterogeneity	Abstraction, code generation
Operation	C.7	Cope with complexity in service partitioning and location	Abstraction
	C.8	Manage architecture components for service deployment and infrastructure provisioning	Modeling languages
Organizational	C.9	Automate as much manual tasks as possible	Model processing
	C.10	Provide formats and guidelines for knowledge sharing	Modeling languages

[a] In the context of the table, the term "manage" covers the actions elicitation, adaptation, and consistent documentation of managed entities

2.3.1 Design

Modeling languages have proven suitable for granularity and API specification in other approaches to service-based architecting [2, 94]. Hence, they can tackle Challenges C.1 and C.4. Modeling languages are also a natural choice for Challenge C.2 because DDD (Sect. 2.1) constructs domain models with modeling languages [24]. Model processing then increases domain models' value (Challenge C.3) by elevating them from documentation artifacts to first-class citizens in software engineering. When resorting to MDE-based microservice design, the detection of cyclic service dependencies (Challenge C.5) is possible at design time using (i) context conditions that constrain model validity to non-cyclic service dependencies; and (ii) static analysis to detect cycles across models.

2.3.2 Implementation

MDE's abstraction from technology [70] predestines it for coping with technology heterogeneity (Challenge C.6). Nonetheless, model-based technology abstraction can be tailored per stakeholder group [95, 110]. Code generation then produces technology-specific code from technology-agnostic models [15, 96]. For MSA, code generation can even increase maintainability (Sect. 1) by automating steps in migrating microservice implementations to other technology stacks.

2.3.3 Operation

For Challenge C.7, we rely on abstraction as it allows capturing of service partitioning and location agnostic to deployment and orchestration technologies. For Challenge C.8, the existence of model-based approaches to operation environment specification [25, 67] proves modeling languages well suited for infrastructure management in MSA.

2.3.4 Organizational

Model processing supports task automation and is therefore inherently suited to deal with Challenge C.9. Modeling languages facilitate sharing of architecture knowledge (Challenge C.10) by formalizing its model-based expression [123], especially in combination with stakeholder-oriented viewpoints for knowledge decomposition [28].

3 LEMMA—A Language Ecosystem for Modeling Microservice Architecture

Here, we present LEMMA [93] in detail. Section 3.1 specifies *architecture viewpoints* [43] for MSA to which LEMMA's modeling languages (Sect. 3.2) and model processing facilities (Sect. 3.3) align. Section 3.4 illustrates LEMMA's usage.

3.1 Microservice Architecture Viewpoints

We leverage the notion of architecture viewpoint ("viewpoint" henceforth) from ISO 42010 [43] to decompose MSA's complexity (Sect. 2.1). A viewpoint frames stakeholder concerns toward a software system. It prescribes languages and techniques to construct *architecture models* as well as operations to process them [43].

For LEMMA, we focused on the following stakeholder groups and their concerns in MSA engineering [13, 29, 36, 38, 102]:

- **Domain experts:** Domain experts demand a software that covers the relevant domain-specific requirements in a cost-effective manner in the expected quality.

- **Microservice developers:** Microservice developers implement and test owned microservices w.r.t. specifications, e.g., of requirements or the architecture.
- **Microservice operators:** Microservice operators are concerned with microservice and operation infrastructure deployment and configuration.
- **Software architects:** Software architects deal with architecture specification and examination to assess quality attribute satisfaction. In addition, they communicate with and across development teams.

From these stakeholders and their concerns, we derived viewpoints for the model-based description of microservice architectures:

- **Domain viewpoint:** This viewpoint supports the construction of *domain models* for microservice architectures. Following DDD, domain model construction may be a collaborative activity by domain experts and microservice developers [24].
- **Technology viewpoint:** This viewpoint reifies the technology heterogeneity of microservice architectures. It captures the concerns of microservice developers and operators toward technology management within *technology models*.
- **Service viewpoint:** The viewpoint addresses the concerns of microservice developers by the construction of *service models* for microservices, their interfaces, and operations. Service models may refer to technology models to reify technology decisions. To enable model reuse and facilitate technology exchange, the viewpoint also considers the construction of *service technology mapping models*, which externalize technology decisions from service models.
- **Operation viewpoint:** This viewpoint allows microservice operators the capturing of microservice deployment and operation in *operation models*.

To increase information content and reusability, MSA models are composable by element references. For example, operation parameters in service models may refer to domain concepts in domain models as types. Figure 1 shows LEMMA's viewpoints and composition relationships between model kinds of different viewpoints. Model composition inherently addresses the concerns of software architects by fostering architecture specification and examination with a coherent architecture representation.

3.2 Modeling Languages

Following ISO 42010, we devised modeling languages (Sect. 2.2) for the construction of MSA viewpoint models (Fig. 1). Figure 2 shows the language development process.

Sections 3.2.1 to 3.2.5 describe the activities of the development process.

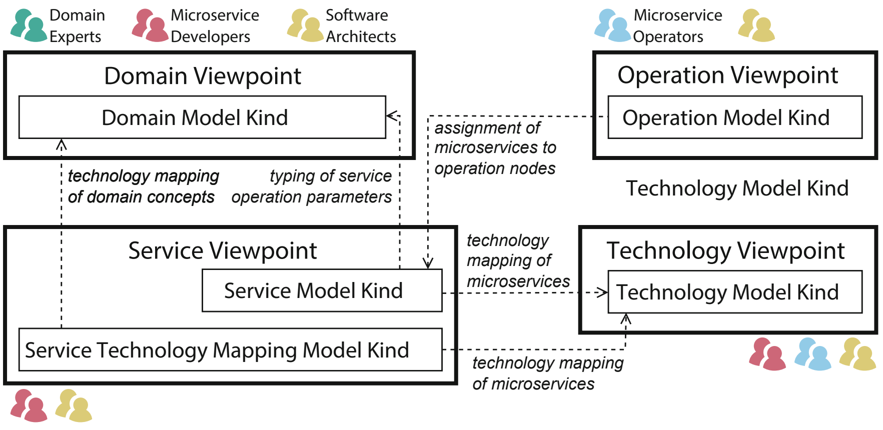

Fig. 1 LEMMA's MSA viewpoints and reference-based composition relationships between model kinds of different viewpoints. The relationships are depicted as dashed arrows from referencing to referred model kinds. Colored icons on a viewpoint box identify the stakeholder groups whose concerns are framed by the viewpoint [43]

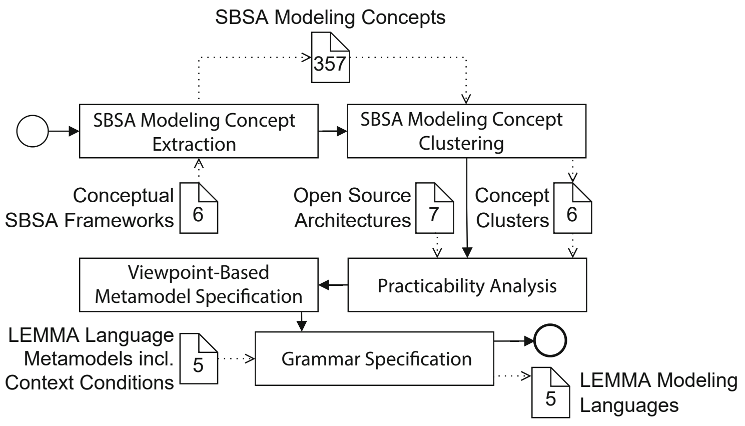
SBSA Modeling Concepts
357
SBSA Modeling Concept Extraction
SBSA Modeling Concept Clustering
Conceptual SBSA Frameworks
6
Open Source Architectures
7
Concept Clusters
6
Viewpoint-Based Metamodel Specification
Practicability Analysis
LEMMA Language Metamodels incl. Context Conditions
5
Grammar Specification
5
LEMMA Modeling Languages

Fig. 2 BPMN diagram [69] of LEMMA's language development process

3.2.1 SBSA Modeling Concept Extraction

We identified and extracted an initial set of potential modeling concepts for LEMMA's modeling languages from six conceptual frameworks [10, 60, 66–68, 121] for the model-based description of service-based software architectures (SBSAs) [23]. We selected these frameworks because they explicitly consider various stakeholder groups, viewpoints, and engineering phases, without prescribing a certain solution architecture. In total, we extracted 357 SBSA modeling concepts and their definitions [92].

Table 2 Count of SBSA modeling concepts per Essential SOA Elements category

Essential SOA Elements category	Concept count
Contract	53
Governance	19
Implementation	31
Infrastructure & Management	77
Interface	177
SLA	19
Not classifiable	58
Sum	**434**

3.2.2 SBSA Modeling Concept Clustering

We clustered the SBSA modeling concepts into the categories of the Essential SOA Elements taxonomy [2], i.e., "Contract," "Governance," "Implementation," "Infrastructure & Management," "Interface," and "Service Level Agreement (SLA)." For the clustering, we relied on concepts' definitions extracted in the previous activity. The clustering enabled us to relate a concept to one or more of the MSA model kinds described in Sect. 3.1. Table 2 summarizes the clustering results.

The mismatch between the sums of clustered modeling concepts (434) and extracted modeling concepts (357) stems from ambiguous concept definitions. For example, based on its definition, SoaML's Capability concept [68] was clustered into the Implementation and Interface categories. On the other hand, some concept definitions were too narrow to permit classification, e.g., the Clipped Structural Modeling Connector concept from the Service-Oriented Modeling Framework [60]. Section 4.3.1 and Appendix B of the dissertation that conceived LEMMA [93] provide more details on the clustering activity.

3.2.3 Practicability Analysis

The previous activities established a conceptual baseline for LEMMA's model languages on the basis of SBSA modeling concepts. This focus on SBSA was necessary as no conceptual frameworks for MSA modeling existed. To assess the applicability of the extracted SBSA modeling concepts for MSA engineering and balance conceptual rigor with practice orientation, we analyzed concepts' manifestation and actual usage in seven open-source microservice architectures. We derived the set of these architectures by joining two subsets of microservice architectures that (i) provide their source code on GitHub;[1] and (ii) have already been academically investigated to gain insights about MSA implementation concepts and patterns [63, 100]. Table 3 lists the considered architectures. They account for 51.35% of the overall lines of code of all architectures in the unified set. For further

[1] https://www.github.com.

Table 3 Open-source microservice architectures selected for practicability analysis of SBSA modeling concepts. Table entries are arranged in descending order by the value in the "Lines of Code" column as of February 1st, 2020

Architecture name	Academic reference	GitHub path[a]	Programming languages	Lines of code
eShopOnContainers	[100]	/dotnet-architecture/eShopOnContainers/tree/20238d53	C#, JavaScript	94,660
Micro company	[63]	/idugalic/micro-company/tree/5a4ee50	Java, JavaScript	83,685
Lakeside Mutual Insurance company	[100]	/Microservice-API–Patterns/Lakeside-Mutual/tree/35a67ac	Java, JavaScript	83,181
Pitstop - garage management system	[63, 100]	/EdwinVW/pitstop/tree/e3afc74	C#, JavaScript	53,591
Microservices reference	[63]	/mspnp/microservices-reference-implementation/tree/69a8f63	C#, Java, JavaScript	18,751
WeText	[63]	/daxnet/we-text/tree/6bab01c	C#, JavaScript	18,523
FTGO - restaurant management	[100]	/microservices-patterns/ftgo-application/tree/9f85c77	Cucumber, Java, JavaScript	15,069

[a] Relative to host https://www.github.com

details, we refer to Sect. 4.3.2 and Appendix C of the dissertation that conceived LEMMA [93].

3.2.4 Viewpoint-Based Metamodel Specification

For each of the viewpoint-specific model kinds in Fig. 1, we defined the abstract syntax of a LEMMA modeling language (Sect. 2.2) as *metamodel* [15]. Consequently, LEMMA comprises five modeling languages, each targeting a different MSA viewpoint. Table 4 provides an overview of these languages.

LEMMA's modeling languages support MSA stakeholders as follows:

- **Domain Data Modeling Language (DDML):** The DDML enables the collaborative construction of domain models by domain experts and microservice developers (Sect. 2.1). It integrates constructs for the model-based expression of domain concepts and their augmentation with DDD patterns.
- **Technology Modeling Language (TML):** The TML addresses microservice developer and operator concerns (Sect. 3.1) in capturing technology decisions.

Table 4 LEMMA's languages per stakeholder group, viewpoint, and model kind

Stakeholder group	Viewpoint	Modeling languages (# Modeling concepts/ context conditions)	Model kind
Domain experts	Domain	Domain Data Modeling Language (28/36)	Domain Model Kind
Microservice developers	Domain	Domain Data Modeling Language	Domain Model Kind
	Service	Service Modeling Language (24/39)	Service Model Kind
		Service Technology Mapping Modeling Language (20/39)	Service Technology Mapping Model Kind
	Technology	Technology Modeling Language (24/37)	Technology Model Kind
Microservice operators	Operation	Operation Modeling Language (12/36)	Operation Model Kind
	Technology	Technology Modeling Language	Technology Model Kind
Software architects	All	All	All

- **Service Modeling Language (SML):** The SML targets microservice developer concerns. Service models constructed with the SML thus specify microservice APIs including operation signatures and physical or logical endpoints.
- **Service Technology Mapping Modeling Language (STMML):** The STMML enables the construction of service technology mapping models to augment service model elements with technology information, thereby keeping service models technology-agnostic and reusable across technologies.
- **Operation Modeling Language (OML):** The OML supports operators in specifying microservice deployment, infrastructure configuration and usage.

We base the composition relationships between model kinds (Fig. 1) on imports, i.e., specific elements in a model can refer to specific elements in imported models.

For each LEMMA modeling language, Table 4 also shows the number of modeling concepts and *context conditions* [15] that prescribe model well-formedness.

Figure 3 shows an excerpt of the SML's metamodel and thus illustrates the influence of the practicability analysis on the eventual definition of metamodel concepts.

An SML `ServiceModel` comprises an arbitrary number of `Microservices`, each having a `name`, `type`, `visibility`, and, optionally, a `version`. A microservice can require other microservices to express service dependencies. Required microservices may originate from the same or an imported service model (`Import` and `PossiblyImportedMicroservice` concepts). A non-imported microservice has one or more `Interfaces`. The `notImplemented` flag specifies whether an interface lacks an implementation, which is useful for iterative API refinement prior to API exposure. An interface has one or more `Operations` that model the respective microservice's behavioral signatures. An

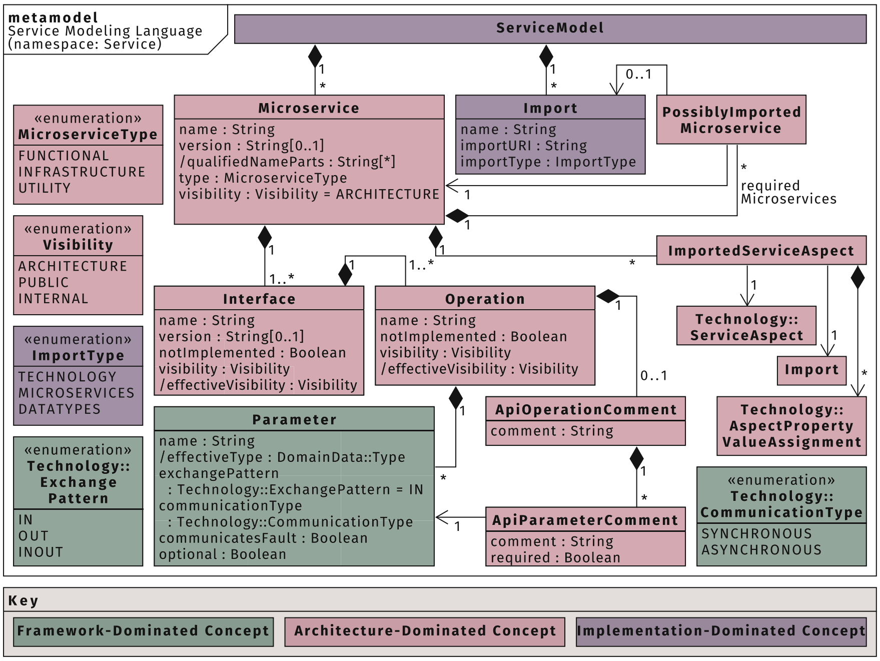

Fig. 3 Excerpt of the SML's metamodel with concepts' structures and relationships in a UML class diagram [73]. The term "dominated" identifies the driving source for a concept's eventual metamodel reification

operation consists of incoming or outgoing `Parameters`. Each parameter has a `CommunicationType` that allows, e.g., modeling of synchronously activated operations that yet exhibit asynchronous behavior. With its `ApiOperationComment` and `ApiParameterComment` concepts, the SML supports API documentation. Since LEMMA relies on *aspects* [105] to augment model elements with metadata, the SML associates microservices with `ImportedServiceAspects`. While originally intended for capturing technology decisions, aspects can also incorporate architectural patterns into LEMMA models [93].

Listing 1 illustrates our usage of OCL [71] to specify metamodel constraints that exceed class diagram expressivity.

Listing 1 Excerpt of the OCL-based [71] context conditions for the SML's metamodel

```
-- Imports in a service model must be unique
context ServiceModel inv uniqueImports:
  self.imports->forAll(i1, i2 | i1 <> i2 implies
    i1.name <> i2.name and i1.importURI <> i2.importURI)
-- Aspects on microservices must have the correct join point
context Microservice inv validJoinPointTypes:
  self.aspects->forAll(a | a.importedAspect.joinPoints
    ->includes(technology::JoinPointType::MICROSERVICES))
-- Interfaces must define at least one operation
context Interface inv notEmpty: self.operations->size() > 0
```

We used the Eclipse Modeling Framework (EMF) [116] and, more precisely, Xcore[2] and Xbase[3] for metamodel implementation. All LEMMA metamodel implementations can be found on Software Heritage [86–90].

3.2.5 Grammar Specification

We specified *concrete syntaxes* [15] for LEMMA's metamodels to make the resulting modeling language practically usable. Based on our experiences with the development and application of a graphical MSA modeling language [98, 114], we decided for textual concrete syntaxes. Since microservice architectures usually involve many services and infrastructure components, graphical models quickly become unclear.

We employed the Xtext framework [11] for grammar specification. Listing 2 shows an excerpt of the Xtext grammar for LEMMA's SML.

Listing 2 Excerpt of the Xtext grammar for LEMMA's SML

```
enum Visibility returns Visibility:
  INTERNAL='internal' | ARCHITECTURE='architecture' | PUBLIC='public';
enum MicroserviceType returns MicroserviceType:
  FUNCTIONAL='functional' | UTILITY='utility' |
  INFRASTRUCTURE = 'infrastructure';
Microservice returns Microservice:
  visibility=Visibility? type=MicroserviceType
  'microservice' name=QualifiedNameWithAtLeastOneLevel
  ('version' version=ID)? '{' interfaces+=Interface+ '}';
```

First, the grammar determines keywords for the literals of the metamodel enumerations `Visibility` and `MicroserviceType` (Fig. 3). Next, it specifies the grammar for the `Microservice` metamodel concept. A microservice is introduced by a visibility modifier and type, followed by the `microservice` keyword and the service's name, which must exhibit at least one qualifying level to support service clustering. The `version` keyword sets the service's version. The `interface` keyword introduces an interface definition of the service within curly brackets. Listing 3 illustrates the SML's usage for modeling the `OrderService` of the microservice architecture used by Richardson to exemplify MSA [102] (Sect. 3.4).

Listing 3 Example of a microservice definition based on the metamodel (Fig. 3) and concrete syntax (Listing 2) of LEMMA's SML

```
public functional microservice org.example.OrderService {
  interface Orders { ... }
}
```

The grammar specifications of LEMMA's modeling languages and the SML code for the `OrderService` can be found on Software Heritage [76–80, 91].

[2] https://wiki.eclipse.org/Xcore.

[3] https://wiki.eclipse.org/Xbase.

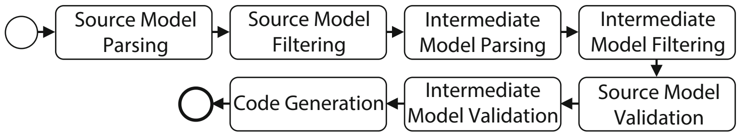
Source Model Parsing
Source Model Filtering
Intermediate Model Parsing
Intermediate Model Filtering
Source Model Validation
Intermediate Model Validation
Code Generation

Fig. 4 Built-in phases of LEMMA's MPF

3.3 Model Processing Framework

We devised a modeling processing framework (MPF) to make LEMMA models actionable as envisioned by MDE (Sect. 2.2). The MPF targets technology-savvy MSA stakeholders, e.g., Microservice Developers and Operators (Sect. 3.1), who not necessarily have an MDE background. Therefore, the MPF (i) structures model processing into phases w.r.t. the Phased Construction pattern [53]; (ii) focuses on Java as the most popular service programming language [13, 106]; (iii) supports programming approaches common in Java-based microservice development, e.g., annotation-based Inversion of Control (IoC) [47] in combination with the Abstract Class pattern [107]; (iv) abstracts from MDE technologies used by LEMMA; and (v) yields stand-alone executable model processors for continuous integration [48].

Figure 4 shows the MPF's model processing phases. It is possible to add custom phases for other model processing purposes like simulation [15].

The phases' responsibilities are as follows:

- **Source/Intermediate Model Parsing:** These phases parse LEMMA models (*source models*; Sect. 3.2) and their intermediate representations (*intermediate models*). LEMMA intermediate models incorporate preprocessed data such as explicit configuration values resulting from implicit default values. Model processors thus need not calculate this data, which also imposes consistency in model processing. Moreover, intermediate models expressed in the generic XML Metadata Interchange format [72] decouple model processors from EMF.
- **Source/Intermediate Model Filtering:** These phases allow the selection of model elements for subsequent processing phases. Each phase expects an OCL file whose queries [71] are evaluated against the source or intermediate model. The MPF then applies follow-up phases only on elements matching the queries.
- **Source/Intermediate Model Validation:** These phases support the provisioning of model validity checks with specific severities. Source model validation may also happen interactively via the Language Server Protocol (LSP).[4] That is, MPF-based model processors leverage the LSP to connect with the Eclipse editor of the respective LEMMA modeling language to display validation results during model construction. Hence, modelers need not invoke a processor separately from the IDE and trace validation results to erroneous model elements manually.

[4] https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification.

- **Code Generation:** The MPF incorporates this phase as code generation is one of the key drivers for MDE adoption [5, 61, 123].

  The Kotlin[5]-based MPF can be found on Software Heritage [83].

3.4 *Illustrative Example*

We illustrate the construction of a microservice with LEMMA's modeling languages (Sect. 3.2) and the processing of the resulting models with LEMMA's MPF (Sect. 3.3).

Listing 4 shows four coherent LEMMA model excerpts in the DDML, TML, and SML (Sect. 3.2). The complete models can be found on Software Heritage [81]. They cover the Order and Restaurant microservices of Richardson's MSA case study [102].

Listing 4a contains the domain model of the Order microservice in LEMMA's DDML. The model consists of the two bounded contexts (Sect. 2.1), `Order` and `API`.

The `Order` context comprises two domain concepts. `Order` is a structured domain concept that consists of five fields of which four have the built-in primitive type `long`, while the `state` field is typed by the enumeration domain concept `OrderState` (Lines 11–16). LEMMA's DDML also integrates keywords for *DDD patterns* [24], e.g., Aggregate and Entity. The `Order` structure combines both these patterns. As an `aggregate`, its instances cluster instances of other domain concepts, which are only accessible from the `Order` instance. As an `entity`, two `Order` instances are distinguishable by a domain-specific `identifier` (see the `id` field in Line 4).

The `API` context comprises three domain concepts for the Order microservice's interactions. The `CreateOrderRequest` concept is a DDD `valueObject` [24], i.e., its instances transport information between architecture components. Therefore, all of its fields are `immutable` and receive a value once during instance initialization.

Listing 4b shows a technology model for Java and Spring[6] in the TML (Sect. 3.2). The `types` section defines technology-specific *type synonyms* for LEMMA's built-in primitive types. During model processing, these synonyms replace all instances of LEMMA primitive types in models that apply the technology model. Since LEMMA's type system is based on Java [33], the mapping of built-in types to technology-specific synonyms Listing 4b is straightforward. For example, the `boolean` type has the synonym `Boolean` (Lines 4–5). The technology model also specifies the `PostMapping` aspect (Lines 20–21). It maps to the

[5] https://www.kotlinlang.org.

[6] https://www.spring.io.

```
// File: Order.data
context Order {
  structure Order<aggregate, entity> {
    long id<identifier>,
    immutable long ^version,
    immutable OrderState state,
    immutable long consumerId = -1,
    immutable long restaurantId = -1
  }

  enum OrderState {
    APPROVED,
    APPROVAL_PENDING,
    REJECTED,
    REVISION_PENDING
  }
}

context API {
  structure CreateOrderRequest
    <valueObject> {
    immutable long consumerId,
    immutable long restaurantId,
    immutable LineItems lineItems
  }

  structure LineItem {
    immutable string menuItemId,
    immutable int quantity
  }

  collection LineItems { LineItem i }
}
```

(a)

```
// File: JavaSpring.technology
technology JavaSpring {
  types {
    primitive type Boolean
      based on boolean default;
    primitive type Byte
      based on byte default;
    primitive type Character
      based on char default;
    primitive type Date
      based on date default;
    primitive type Double
      based on double default;
    primitive type Float
      based on float default;
    ...
  }

  service aspects {
    aspect PostMapping<singleval>
      for operations;
  }
}
```

(b)

```
// File: Protocols.technology
technology Protocols {
  protocols {
    sync rest data formats "json"
      default with format "json";
  }
}
```

(c)

```
// File: Order.services
import datatypes from "Order.data" as OrderDomain
import technology from "JavaSpring.technology" as JavaSpring
import technology from "Protocols.technology" as Protocols

@technology(JavaSpring)
@technology(Protocols)
public functional microservice org.example.OrderService {
  @endpoints(Protocols::_protocols.rest: "/orders";)
  interface Orders {
    ---
    Create order
    @required request Request
    ---
    @JavaSpring::_aspects.PostMapping
    create(
      sync in request : OrderDomain::API.CreateOrderRequest,
      sync out response : OrderDomain::API.CreateOrderResponse
    );
  }
}
```

(d)

Listing 4 Example models in LEMMA's (a) DDML, (b, c) TML, and (d) SML. The models are excerpts from the models for the order and restaurant microservices of Richardson's MSA case study [102] used to illustrate LEMMA's model processing capabilities [81]

eponymous Spring annotation[7] and is applicable to microservice operations (`for operations`) exactly once (`singleval`).

Listing 4c constructs a technology model with the `rest` protocol for the REST architectural style [26]. REST is often applied in synchronous microservice interactions [13]. The `rest` protocol uses the JSON data format [21] and is the `default` synchronous protocol when a microservice applies the technology model.

Listing 4d shows a service model in the SML. It exemplifies the imported-based composition of LEMMA models (Sect. 3.2) as it imports the domain model in Listing 4a under the alias `OrderDomain`, and the two technology models in Listing 4b and c under the aliases `JavaSpring` and `Protocols` (Lines 2–4). The two technology models are applied to the `OrderService` microservice (Line 8) by the built-in `@technology` annotation. These applications lead to implicit replacement of types with synonyms (Listing 4b) and the assumption of default protocols (Listing 4c).

The `OrderService` has a `public` visibility, which allows its exposure to external clients, and a `functional` type, which identifies the service's capability to stem from the application domain. The `OrderService` consists the `Orders` interface (Lines 10–20) with a `rest` endpoint (Line 9). In LEMMA, an endpoint is a combination of a protocol from a technology model being applied to a microservice (Line 7 and Listing 4c) and one or more addresses, i.e., "/orders" (Line 9).

The `Orders` interface consists of the `create` operation (Lines 16–19). The API comment (Lines 11–14) informs about the operation's function. `create` defines the synchronous input parameter `request` and the synchronous output parameter `response`. The type of the former is the structured domain concept `CreateOrderRequest` imported from the domain model in Listing 4a. The type of the latter is the response-specific counterpart of `CreateOrderRequest`, i.e., `CreateOrderResponse` [81]. `create` applies the `PostMapping` aspect from the technology model in Listing 4b to specify that the operation is invokable by HTTP POST requests [27].

Listing 5 shows excerpts from an MPF-based model processor (Sect. 3.3), whose complete Java sources are available on Software Heritage [81]. The processor yields the number of microservices' interfaces in a LEMMA service model and also distinguishes between interfaces with only asynchronous or synchronous operations. Such classifications of interfaces are crucial to MSA-specific quality metrics [22].

Listing 5a shows the Java class of the processor's source model validator (Sect. 3.3). The annotation `@SourceModelValidator` allows LEMMA's MPF to find source model validators on the classpath. A source model validator must extend `AbstractXtextModelValidator` and override its `getSupport-edFileExtensions` method to inform the MPF about the validator's supported

[7] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/PostMapping.html.

```
@SourceModelValidator
public class ServiceModelValidator extends AbstractXtextModelValidator {
  @Override
  public Set<String> getSupportedFileExtensions() {
    return new HashSet<>() {{ add("services"); }};
  }

  @Check
  private void checkMicroservice(Microservice microservice) {
    info("Model contains a microservice named " + microservice.getName(),
      ServicePackage.Literals.MICROSERVICE__NAME);
  }
}
```

(a)

```
@CodeGenerationModule(name = "main")
public class CodeGenerationModule extends AbstractCodeGenerationModule {
  @Override
  public String getLanguageNamespace() {
    return IntermediatePackage.eNS_URI;
  }

  @Override
  public Map<String, Pair<String, Charset>> execute(String[] phaseArguments,
    String[] moduleArguments) {
    var resultFileContents = new StringBuilder();
    var serviceModel = (IntermediateServiceModel) getResource()
      .getContents().get(0);
    for (IntermediateMicroservice m : serviceModel.getMicroservices()) {
      var syncCount = 0;
      var asyncCount = 0;
      for (IntermediateInterface i : m.getInterfaces()) {
        var paramTypes = i.getOperations().stream()
          .map(IntermediateOperation::getParameters)
          .flatMap(Collection::stream)
          .map(IntermediateParameter::getCommunicationType)
          .collect(Collectors.toList());
        if (paramTypes.isEmpty() ||
          paramTypes.stream().noneMatch(t -> t.equals("ASYNCHRONOUS")))
          syncCount++;
        else if (paramTypes.stream()
          .noneMatch(t -> t.equals("SYNCHRONOUS")))
          asyncCount++;
      }
      resultFileContents.append(String.format(
        "Interface count for microservice %s: %d\n" +
        "\tSynchronous interface count: %d\n" +
        "\tAsynchronous interface count: %d\n",
        m.getQualifiedName(), m.getInterfaces().size(), syncCount,
        asyncCount
      ));
    }
    var resultFilePath = getTargetFolder() + File.separator + "results.txt";
    var resultMap = new HashMap<>()
      {{ put(resultFilePath , resultFileContents.toString()); }};
    return withCharset(resultMap, StandardCharsets.UTF_8.name());
  }
}
```

(b)

```
public class ModelProcessor extends AbstractModelProcessor {
  public static void main(String[] args) {
    new ExampleModelProcessor().run(args);
  }

  public ExampleModelProcessor() {
    super("de.fhdo.lemma.examples.model_processing");
  }
}
```

(c)

Listing 5 Example model processor written in Java based on LEMMA's MPF. For each microservice in the input service model, the processor (a) prints an information message; and (b) generates a file with the overall interface count as well as with the share of asynchronous and synchronous interfaces. The code in (c) shows the processor's entry point

model file types. Methods with the `@Check` annotation are validation methods. The types of their first parameters must correspond to the metamodel concepts for whose instances in a model the validation methods shall be invoked by the MPF. Thus, the `checkMicroservice` method in Lines 9–12 validates `Microservice` instances in LEMMA service models (Fig. 3).

Listing 5b contains a code generation module of the example model processor. The MPF modularizes the Code Generation phase (Sect. 3.3) to enable separation of concerns in complex model processors [49]. A code generation module is a Java class that (i) exhibits the `@CodeGenerationModule` annotation; (ii) extends `AbstractCodeGenerationModule`; and (iii) overrides the `getLanguageNamespace` method to signal the MPF the namespace of the metamodel targeted by code generation. In case of the module in Listing 5b, this namespace identifies the intermediate representation of LEMMA service models (Sects. 3.2 and 3.3). The `execute` method (Lines 9–42) implements the module's logic. The `for`-loop in Lines 17–29 iterates over all microservices and interfaces in the given intermediate service model and counts the number of interfaces whose operations have only asynchronous or synchronous parameters. Lines 30–36 map this information to a string and buffer it in the `resultFileContents` variable. Finally, Lines 38–41 inform the MPF about the generated file content for its eventual serialization.

Listing 5c comprises the processor's entrypoint, i.e., a Java class that inherits from `AbstractModelProcessor` and has a `main` method that delegates execution to the MPF. This delegation informs the MPF about the processor's Java package that shall be scanned for phase implementations like those in Listing 5a and b.

4 Applications of LEMMA

This section presents applications of LEMMA for microservice code generation (Sect. 4.1), model-based architecture reconstruction (Sect. 4.2), static quality analysis (Sect. 4.3), defect resolution by model refactoring (Sect. 4.4), and establishing a common architecture understanding (Sect. 4.5).

4.1 Plugin-Based Generation of Technology-Specific Microservice Code

MSA's technology heterogeneity (Sect. 1) not only concerns architecture models (Sect. 3.1) but also microservice implementations [95]. Therefore, we devised a code generator for Java-based microservice programming that maps LEMMA models in their intermediate representations (Sect. 3.3) to basic Java abstract syntax trees (ASTs). Besides Java, these basic ASTs are technology-agnostic in that they do not leverage a specific microservice implementation technology. For the specification of the mapping between intermediate LEMMA model element types and Java AST node types, we refer to Appendix K of the dissertation that conceived LEMMA [93].

Our Java Base Generator (JBG) [82] draws on LEMMA's MPF and is also a framework for the development of technology-specific code generation plugins, called Genlets. The JBG may load an arbitrary number of pre-compiled Genlets and execute them in a specific order after the creation of basic Java AST nodes from traversed intermediate model elements. Genlets consist of a set of code generation handlers, which are Java classes that exhibit the `@CodeGenerationHandler` annotation and implement the `GenletCodeGenerationHandlerI` interface.

A Genlet requests the JBG to invoke it for a combination of model element type and AST node type and pass to it both the element and the mapped node. The JBG then passes the element and node to the Genlet for technology-specific adaptation after which the JBG integrates the adapted node in the Java AST. After the execution of all given Genlets, the JBG serializes the adapted Java ASTs, which may involve a reordering of the ASTs to comply with patterns that preserve manual changes to generated code upon re-generation [35].

Figure 5 exemplifies the interaction between the JBG and its Genlets in the context of the `Orders` interface from Listing 4d.

The JBG maps an interface modeled in LEMMA's SML to an eponymous Java class (`NormalClassDeclaration` instance [33] in Compartment 1 of Fig. 5). Modeled operations become Java methods (`MethodDeclaration` instance [33] in Compartment 1). The Spring Genlet adapts the generated class to behave as a REST controller that invokes the `create` operation when receiving an HTTP POST request (Compartment 2 in Fig. 5). This adaptation follows from the modeled `rest` endpoint of the `Orders` interface (Listing 4d) and the application of the `PostMapping` aspect to the `create` operation. In the serialization phase (Compartment 3 in Fig. 5), the JBG adapts the AST to be compatible with the Generation Gap to preserve manual changes to generated code. Next to this pattern, the JBG also supports its extended variant [35], which reduces the amount of pattern-specific boilerplate code.

LEMMA currently bundles Genlets for Spring, the Kafka message broker,[8] DDD, and the Domain Event and CQRS patterns [75, 102]. Since a Genlet is inherently a LEMMA model processor, it can leverage functionality provided by the MPF including stand-alone execution for interactive model validation (Sect. 3.3). We applied the JBG in a research project from the Electromobility domain and were able to generate the implementations of all domain concepts, microservice interfaces, and extensible infrastructure for asynchronous interaction. The generation efficiency ranged between 5.90 and 6.26, i.e., from one line of LEMMA model, roughly six lines of Java microservice code were producible, making generative microservice development with LEMMA basically efficient. For details, we refer to

[8] https://kafka.apache.org.

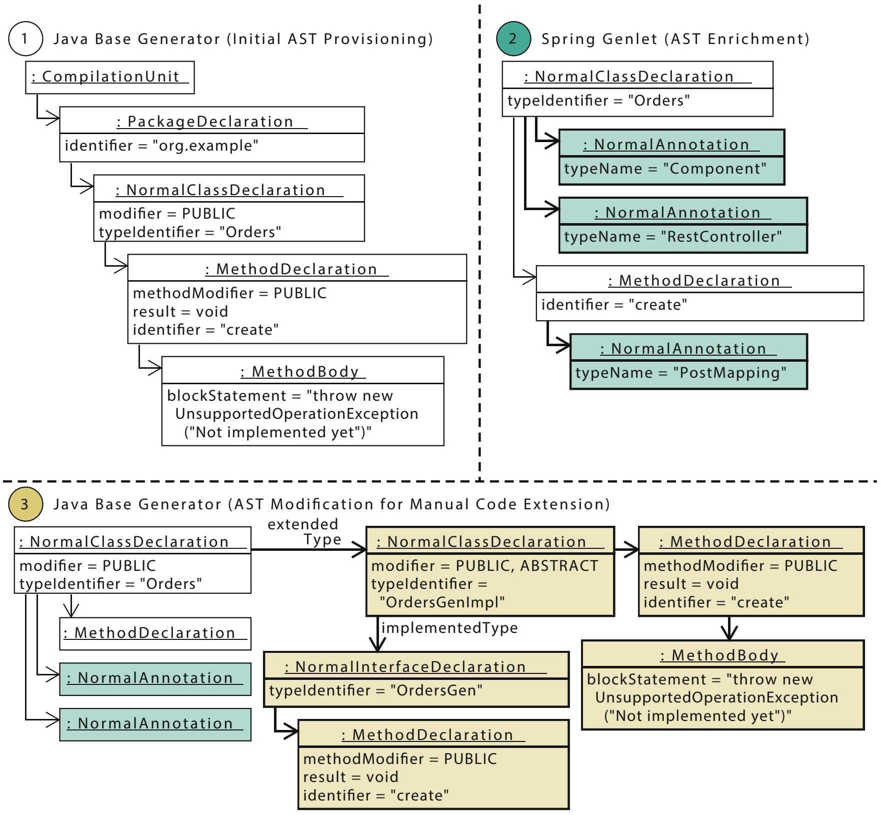

Fig. 5 Sample Java AST enrichment with the JBG and the Spring Genlet [99]

Sect. 8.7 of the dissertation that conceived LEMMA [93]. Recent works leveraged the JBG and its Genlets to derive microservice code from underspecified domain models [96], integrate blockchain technology into microservice architectures [122], and realize asynchronous microservice interactions [99].

4.2 *Model-Based Reconstruction of Microservice Architectures*

MSA's emphasis on service-specific independence (Sect. 1) may lead to service proliferation and the subsequent erosion of the anticipated architecture design because teams can autonomously advance different architecture parts [13]. Software Architecture Reconstruction (SAR) [8] is thus an important area in MSA research [1]. This section describes the design, development, and evaluation of an extensible LEMMA-based SAR approach that automates the translation of the source code of existing microservice architectures into LEMMA models, thereby

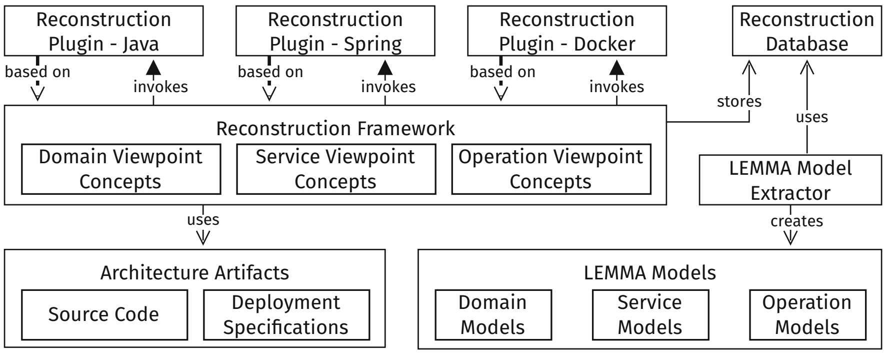
Reconstruction Plugin - Java
Reconstruction Plugin - Spring
Reconstruction Plugin - Docker
Reconstruction Database
based on
invokes
stores
uses
Reconstruction Framework
Domain Viewpoint Concepts
Service Viewpoint Concepts
Operation Viewpoint Concepts
LEMMA Model Extractor
creates
Architecture Artifacts
Source Code
Deployment Specifications
LEMMA Models
Domain Models
Service Models
Operation Models

Fig. 6 Core components of our LEMMA-based SAR approach

facilitating the reasoning about the architecture and enabling the application of MDE techniques like quality assessment (Sect. 4.3) and defect resolution (Sect. 4.4).

4.2.1 An Extensible Approach for LEMMA-Based Microservice Architecture Reconstruction

Figure 6 depicts the core components of our LEMMA-based SAR approach.

The Reconstruction Framework (RF) orchestrates the SAR process according to Bass et al. [8].

In the first phase, the RF recovers architecture information from the artifacts of a microservice architecture including its source code and deployment specifications. To this end, the RF iterates over all artifacts of a given architecture and invokes reconstruction plugins on artifacts. These plugins cover different microservice technologies and LEMMA viewpoints. They are responsible for extracting architecture information from given artifacts, translate the information into the format expected by the RF, and return it to the RF. In the sense of Bass et al., the plugins perform a *raw view extraction* [8].

In the second phase, the RF stores all extracted architecture information in a reconstruction database. For this purpose, we specified data formats for each MSA viewpoint (Sect. 3.1). The database enables the RF's future extension by dynamic analyses where gathered architecture information originates from continuous monitoring. This phase corresponds to *database construction* and *view fusion* in the SAR process of Bass et al. [8] where heterogeneous architecture information are harmonized and stored in a common format.

In the third phase, the RF enables subsequent, LEMMA-based processing of reconstructed architecture information. In a first step, the RF invokes the LEMMA model extractor [97] to serialize information from the reconstruction database into LEMMA model files for the reconstructed viewpoints (Sect. 3.2). Starting from these reconstructed view models, software architects can perform efficient *architecture analyses*, e.g., for quality assessment (Sect. 4.3) or defect detection (Sect. 4.4), as suggested by Bass et al. [8].

4.2.2 Evaluation of the LEMMA-Based Reconstruction Approach

We evaluated our LEMMA-based SAR approach on Lakeside Mutual,[9] which is an MSA case study application for a fictitious insurance company. The architecture consists of a generic Customer Core microservice and four more specific microservices for customer, policy, and risk management as well as customer self-service. The Customer Self-Service and Policy Management microservices interact via asynchronous messaging. All remaining microservices rely on HTTP-based interaction. The services (i) use a registry to discover each other; (ii) are primarily implemented in Java and Spring (Sect. 3.4); (iii) produce logs for runtime monitoring; and (iv) store information in their own databases. We selected Lakeside Mutual for the evaluation of our SAR approach because its architecture is well documented [129, 130].

Table 5 shows the results of the evaluation of our SAR approach on Lakeside Mutual. The evaluation used reconstruction plugins for Java and Spring that cover LEMMA's Domain and Service viewpoints (Sect. 3.1 and Fig. 6). We use the Recall, Precision, and $F_{measure}$ metrics to assess the preciseness of the reconstruction process.

The evaluation showed that the current implementation of our SAR approach is able to reconstruct four of the five microservices of Lakeside Mutual in LEMMA service model. The RF did not recover the Risk Management microservice because it is based on Node.js[10] and our reconstruction plugins currently target Java. However, all expected interfaces and operations of the reconstructed microservices could be recovered with reconstructed data structures in LEMMA domain models originating from operations' parameter types (Sect. 3.4). The discrepancy between expected and recovered structures results from classes defined in external dependencies whose source code is currently not available to the RF.

4.3 Assessment of Microservice Maintainability with Static Model Analysis

Next to scalability, maintainability is the most crucial quality attribute in MSA [17, 119] (Sect. 1). There exist several metrics suites that define metrics for maintainability assessment of microservices [3, 22, 39, 41]. While the majority of these metrics does not target MSA, but SOA [3, 41] or REST [39], they are still known

[9] https://www.github.com/Microservice-API-Patterns/LakesideMutual/tree/bc79075.

[10] https://www.nodejs.org.

Table 5 Preliminary reconstruction results from the reconstruction process

Element	**Expected**	**True positives**	**False positives**	**False negatives**	**Recall**	**Precision**	$\mathbf{F_{measure}}$
Microservices	5	4	0	1	80%	100%	88%
Interfaces	16	14	0	2	87%	100%	93%
Operations	61	50	3	8	86%	94%	90%
Data structures	161	117	29	14	89%	80%	84%

to be applicable to MSA [12]. We were interested in the assessment of these metrics by static LEMMA model analysis to provide fast feedback about modeled microservices' quality.

We investigated the LEMMA-based calculation of each of the 26 metrics from the aforementioned four metrics suites. They can be characterized as follows:

- **Hirzalla et al. [41]:** Hirzalla et al. define ten metrics for single SOA services and complete architectures. Among such metrics are (i) NOVS (Number of Versions per Service), which measures architecture complexity by calculating the ratio between service versions and services; and (ii) SRP (Service Realization Pattern), which measures the share of services exposed by intermediaries in the overall share of exposed services with a lower share hinting at lesser complexity. While NOVS is directly assessable from LEMMA service models, SRP requires operation models and a notion of intermediary like API gateway or edge server [7].
- **Athanasopoulos et al. [3]:** This suite consists of three metrics that measure service interface cohesion on the message, conversation, and domain level. Cohesion is important for microservices as it has a direct impact on maintainability [109, 119]. All metrics of the suite rely on *interface-level graphs* (ILG) which are undirected, labeled, and weighted graphs whose vertices represent interface operations and whose weighted edges inform about operations' similarity. The *ideal ILG* is a complete ILG with similarity weight 1, i.e., all interface operations are maximal similar. The *lack of interface-level cohesion* is then computable as the relative difference between the ILG and the ideal ILG.

  The metrics in the suite differ by their calculation rules for ILG similarity weights. For instance, the Message-Level Cohesion Lack metric considers the similarity of operations' message types, whereas the Domain-Level Cohesion Lack metric focuses on operation similarity based on domain terms. All metrics in the suite are directly computable from LEMMA service models.
- **Haupt et al. [39]:** Haupt et al. define seven metrics for structural REST API analysis. The metrics rely on *managed resources*, i.e., objects of information maintained via REST [26]. For their LEMMA-based computation, the majority of the metrics require a technology model that indicates REST application (Sect. 3.4) as well as domain and service models. For example, to assess the Number of Resources metric, it is mandatory to identify REST operations (technology and service model; cf. Listing 4) and managed resources as structural types of service operation parameters (domain model).
- **Engel et al. [22]:** This suite comprises six metrics for MSA core principles like loose coupling. Those metrics include Number of (A)Synchronous Interfaces and Average Size of Asynchronous Messages. While the former is computable from LEMMA service models (Listing 5b), the latter requires runtime monitoring and is only heuristically assessable. That is, parameter types of modeled asynchronous operations allow lower-bound assessment of message sizes.

From the 26 metrics defined in the presented suites, 20 were computable from LEMMA models. The remaining six metrics either require dynamic analysis or process modeling, which is currently out of LEMMA's scope. For details, we refer to Sects. 9.5.2 to 9.5.5 of the dissertation that conceived LEMMA [93].

We implemented a library for the computation of supported metrics [84] and an MPF-based static analyzer [85] (Sect. 3.3) allowing the library's usage. We evaluated the analyzer on the LEMMA reconstruction models of Lakeside Mutual (Sect. 4.2) and revealed weaknesses in service cohesion. In our ongoing works, we integrate the analyzer library with LEMMA's Eclipse plugins (Sect. 3.2) to provide MSA stakeholders with ad hoc visual feedback about microservice maintainability.

4.4 Defect Resolution by Model Refactoring

Defects of a software architecture refer to issues in its design that may cause unwanted behavior of the implemented system. They are often made unintentionally, and without the awareness of software architects and developers [74]. Furthermore, their manifestation and occurrence is impacted by the architectural style, e.g., MSA. For defect resolution, the architecture design and implementation usually need to undergo a refactoring process. In the following, we describe a preliminary approach for the LEMMA-based detection and resolution of security defects in microservice architectures [125].

We illustrate our approach for a common security defect in MSA, i.e., Publicly Accessible Microservices (PAM), where interfaces are not exposed in a restricted and controlled fashion by an intermediary but are instead freely accessible by architecture-external clients [74]. This public and complete exposure of service interfaces increases the risk for confidentiality violations and other security issues significantly. To resolve the defect, an intermediary for interface exposure, e.g., an API Gateway [7], should be integrated into the architecture.

Listing 6 shows LEMMA technology and operation models (Sect. 3.2) that allow detection of the PAM defect and eventually resolve it.

The technology model in Listing 6a defines aspects that allow the enrichment of `infrastructure` nodes in LEMMA operation models with functional semantics. For instance, the `isApiGateway` aspect can be used to communicate the intent that a certain infrastructure node represents an API Gateway independent of the actual technology used to realize this capability. Listing 6b is technology model for a concrete API Gateway technology, i.e., Zuul.[11] Listing 6c is a LEMMA operation model that applies the technology models in Listing 6a and b to specify a Zuul-based infrastructure node called `Gateway` and identify it as an API Gateway using the `isApiGateway` aspect. Listing 6d contains an operation model with a specification for container-based microservice deployment. More precisely, it models the Docker[12] deployment of the Lakeside Mutual's Customer Core

[11] https://github.com/Netflix/zuul.

[12] https://www.docker.com.

```
1 technology ComponentSemantics {
2   operation aspects {
3     aspect isApiGateway<singleval> for infrastructure;
4     ...
5 }}
```

(a)

```
1 technology Zuul {
2   infrastructure technologies {
3   Zuul {
4     operation environments =
5       "openjdk:11-jdk-slim";
6       service properties
7         { string hostname; }
8 }}}
```

(b)

```
@technology(ComponentSemantics)  1
@technology(Zuul)                2
Gateway is                       3
  Zuul::_infrastructure.Zuul {   4
      aspects {                  5
      ComponentSemantics::       6
        _aspects.isApiGateway;   7
}}                               8
```

(c)

```
1 @technology(Docker)
2 container CustomercoreContainer
3   deployment technology Docker::_deployment.Docker
4   deploys customercore::com.lakesidemutual.CustomerCore
5   depends on nodes ServiceRegistry::Registry {
6 }
```

(d)

```
1 @technology(Docker)
2 container CustomercoreContainerContainer
3   deployment technology Docker::_deployment.Docker
4   deploys customercore::com.lakesidemutual.customercore.CustomerCore
5   depends on nodes ServiceRegistry::Registry, APIGateway::Gateway {
6 }
```

(e)

Listing 6 Example models for LEMMA-based defect resolution. We rely on aspects to assign semantics to infrastructure components (a). The technology model in (b) describes a concrete API Gateway technology used in the operation model in (c) by an infrastructure node denoting an API Gateway. The operation model in (d) captures the Docker-based deployment of a microservice that does not use the API Gateway, whereas (e) shows the refactored operation model resolving this defect

microservice (Sect. 4.2). The `depends on` directive (Line 5) shows that the Docker container only depends on a service registry from another imported operation model. Hence, it does not leverage the capabilities of an API Gateway, thereby introducing the PAM defect.

For the detection of defects in LEMMA models, we implemented a model processor using LEMMA's MPF (Sect. 3.3). The processor's validation phase identifies defects in given LEMMA models and reports them to the user by hinting at the defect-inducing model element. In order to facilitate defect resolution, we implemented an Eclipse plugin that enriches defect issues reported by the processor with quick fixes that are applicable to resolve the detected defect via automated model refactoring. For the PAM defect, the corresponding refactoring is the addition of an API Gateway and its usage by concerned microservices [74]. Listing 6d illustrates the defect resolution by adding the `Gateway` from Listing 6c to the nodes on which the container depends (Line 5).

We are currently working on integrating the defect resolution processor with LEMMA's JBG (Sect. 4.1) such that refactored models can directly be mapped to microservice code and configuration artifacts. As a result, we can eventually provide MSA stakeholders with means for defect detection, resolution, resolution reasoning, and evaluation, including the subsequent generation of code for the most suitable resolution.

4.5 Model Transformations for a Common Architecture Understanding

MSA poses challenges to organizations in restructuring their development processes to cope with service-specific independence and ownership (Definition 1). The division into different teams, each of which being holistically responsible for one or more microservices, can lead to a lack of architectural understanding across team boundaries. This lack of understanding can have a negative effect, e.g., when setting development priorities. We have observed this effect especially in small- and medium-sized enterprises, whose service landscapes evolve together with their development organizations [113].

Model-driven approaches such as LEMMA are particularly suitable for documenting and transferring knowledge [15]. Therefore, we argue that LEMMA constitutes an effective means to create and maintain a common and organization-wide architecture understanding by making (partial) MSA models available to teams and support their active exchange. For this purpose, due to microservices fostering technology heterogeneity (Sect. 2.1), the application of MDE technologies and techniques, e.g., code generation (Sect. 4.1) or model-based reconstruction (Sect. 4.2), cannot be assumed. However, LEMMA provides bidirectional model transformations [59] to derive LEMMA models from microservice API specifications based on OpenAPI[13] (synchronous APIs) or Apache Avro[14] (asynchronous APIs) and vice versa. Consequently, these transformations allow knowledge documentation and communication without requiring microservice teams to develop their services with MDE.

In the following, we (i) describe a development process for small- and medium-sized enterprises that supports both *code-first* and *model-first* microservice development in an integrated fashion; and (ii) how LEMMA's OpenAPI model transformation enables this process. For more details regarding the process, the transformation, and the corresponding artifacts, we refer to our previous work [115].

[13] https://www.openapis.org.

[14] https://avro.apache.org.

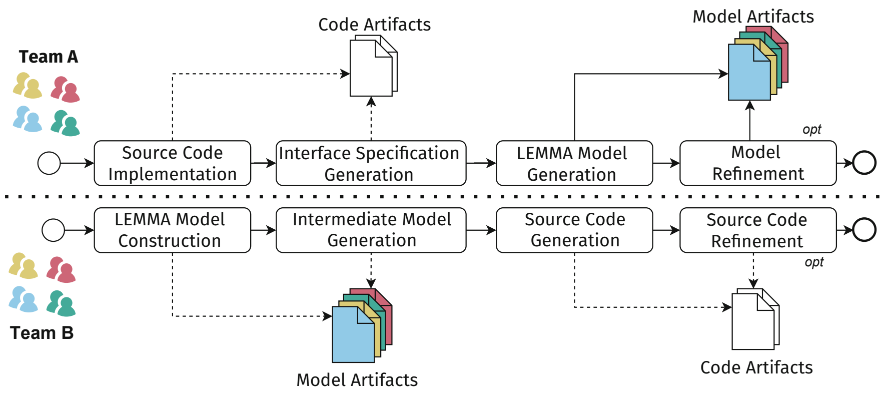

Fig. 7 Code-first vs. model-first microservice development in cross-functional teams

4.5.1 Code-First vs. Model-First Microservice Development

Figure 7 compares code-first with model-first microservice development in two *cross-functional* MSA teams [7].

Team A applies the code-first approach to microservice development in which source code is a first-class citizen and models are used, if at all, for mere documentation and communication. That is, Team A starts to implement a service by writing its source code, followed by the automated generation of interface specification. The latter step follows from insights of an exploratory study [113] in which we found that MSA teams in industry rarely and reluctantly create manual documentation of their interfaces but instead rely on automated approaches, e.g., Swagger[15] to generate OpenAPI specifications. With LEMMA, it is now possible to automatically transform generated interface specifications into corresponding LEMMA domain, service, and technology models (Sect. 3.2). Team A may then refine the derived LEMMA models, if desired, and eventually share them with other teams to stimulate the creation of a common architectural understanding by exploiting MDE's abstraction facilities (Sect. 2.2) and to support model-first development approaches. With the support of model generation in a code-first approach, thus enabling teams to communicate, share knowledge, and create a common understanding, LEMMA addresses a possible lack of expertise on the part of developers, which is a common challenge for the success of MDE tools in practice [124].

Team B in Fig. 7 practices model-first microservice development, which uses LEMMA models as first-class citizens, thereby directly following MDE's line of thought. From such models and their intermediate representations, a code generator

[15] https://www.swagger.io.

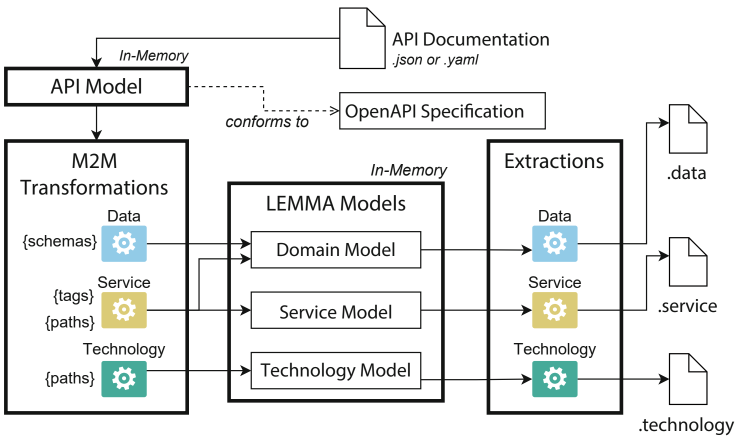

Fig. 8 M2M transformation process for deriving LEMMA domain, service, and technology models from OpenAPI specifications

like the JBG (cf. Sect. 4.1) can be used to produce refinable code. As for the code-first approach, both LEMMA model and source code artifacts exist in the end.

4.5.2 OpenAPI Model Transformation

LEMMA realizes the code-first approach (Fig. 7) through multiple model-to-model (M2M) transformations [15], which we detail on the example of the OpenAPI-to-LEMMA transformation in Fig. 8.

The M2M transformation process starts with an OpenAPI-conform interface specification in a file with the extension ".json" or ".yaml". This specification is parsed into an in-memory API Model fueling multiple M2M transformations. Since the intended use of OpenAPI is to describe HTTP resource APIs, corresponding specifications include utilizable information about data, interfaces, and transfer-specific technology information like media types. This information is translated into LEMMA domain, service, and technology models by means of dedicated M2M transformations.

In detail, the Data transformation operates on `schemas` objects in OpenAPI specifications and generates a data structure in a LEMMA domain model for each traversed schema. The Service transformation processes OpenAPI `tags` and `paths` objects. It creates a LEMMA service model that is populated with interfaces for each encountered `tag`. Paths corresponding to a tag result in interface operations with request and response parameters. Furthermore, the Service transformation generates matching LEMMA collection types for each OpenAPI array. The Technology transformation analyzes the OpenAPI `paths` object for specific media types and creates a corresponding LEMMA technology model. Subsequently, the resulting in-memory LEMMA models are serialized as files by specialized extractors and are thus immediately usable by MSA teams.

5 Related Work

Table 6 compares LEMMA with related MDE-for-MSA approaches. Additional details can be found in Sects. 4.6 and 5.6 of the dissertation that conceived LEMMA [93].

While related approaches cover some of the MSA viewpoints (Sect. 3.1), LEMMA is the only approach to support them all, including viewpoint-specific modeling languages and holistic MSA modeling by viewpoint integration (Sect. 3.2). A further strength of LEMMA is its extensibility on the language level, enabled by the aspect-oriented metadata mechanism of the TML. This extensibility allows, e.g., model-based reification of architecture patterns and selective technology-specificity, which is essential for *agile modeling* [103].

In comparison, LEMMA also facilitates model processing by bundling a specialized MPF (Sect. 3.3) and sophisticated model processors. Additionally, LEMMA has proven to be exceptionally versatile in a variety of MSA engineering scopes, ranging from development and operation over reconstruction to quality assessment (Sect. 4).

6 Conclusion and Future Work

This chapter presented LEMMA (Language Ecosystem for Modeling Microservice Architecture) [93]—an approach for the application of Model-Driven Engineering to Microservice Architecture (MSA) engineering. LEMMA mitigates the complexity of MSA (Sect. 2) by first decomposing it along four viewpoints on microservice architectures, each capturing the concerns of different MSA stakeholders in dedicated architecture models (Sect. 3.1). The Domain viewpoint supports the collaborative construction of domain models by domain experts and microservice developers. Domain models cluster all domain concepts relevant to a microservice architecture. The Technology viewpoint focuses on the concerns of microservice developers and operators and enables them to capture technologies for microservices and operation nodes within technology models. The Service viewpoint provides microservice developers with modeling facilities for microservices, their interfaces, operations, and endpoints. The Operation viewpoint addresses the concerns of microservice operators in deployment and infrastructure operation modeling. We accompanied each viewpoint with a specialized modeling language that formalizes the syntax and semantics of viewpoint-specific MSA models (Sect. 3.2). LEMMA's modeling languages are integrated by means of an import mechanism so that elements in one model can refer to elements in imported models, e.g., to configure the container-

Table 6 Comparison of LEMMA with related MDE-for-MSA approaches

Characteristic/approach	**LEMMA** [93]	**CloudML** [25]	**Context Mapper** [50]	**DCSL** [20]	**JDL** [45]	**MDSL** [50]	**μART** [34]	**Micro Builder** [120]	**MiSAR** [1]
MSA viewpoint (VP) support									
Domain VP	●	–	●	●	–	●	–	●	–
Service VP	●	–	–	–	●	●	●	●	●
Technology VP	●	●	◐	–	●	–	–	●	●
Operation VP	●	●	–	–	●	–	–	–	●
Specialized language per VP	●	–	–	●	–	–	●	–	–
VP Integration	●	●	●	N/A	●	●	N/A	●	●
Modeling languages									
Language-level extensibility	●	–	–	●	–	–	–	–	–
Selective technology-specificity	●	–	–	–	–	–	–	–	–
Technology-agnostic syntax	●	–	◐	●	–	●	–	–	–
Model processing									
Model processor creation	●	–	◐	–	–	–	–	–	–
Bundled code generators	●	●	●	●	●	●	–	●	–
Handwritten extension of generated code	●	–	–	●	–	–	N/A	–	N/A
Bundled static analyzers	●	–	–	–	–	–	–	–	–
MSA engineering scope									
Development	●	–	●	●	●	●	–	●	-
Operation	●	●	–	–	●	–	–	–	–
Reconstruction	●	–	–	–	–	–	●	–	●
Quality assessment	●	–	–	–	–	–	–	–	–

Symbol key: ● = Full support; ◐= Partial support; – = No support based deployment of a modeled microservice within an operation model. The modeling languages are practically usable as plugins for the Eclipse IDE. LEMMA also bundles its own model processing framework (MPF; Sect. 3.3). The MPF facilitates model processor implementation for technology-savvy MSA stakeholders by decoupling model processing into phases and allow phase implementation by mechanisms that are popular in MSA engineering, e.g., Java and annotation-based Inversion of Control [47]. We exemplified the usage of LEMMA's modeling languages and MPF in the context of a case study microservice architecture (Sect. 3.4). Section 4 presented practical applications of LEMMA for microservice code generation (Sect. 4.1), architecture reconstruction (Sect. 4.2), quality analysis (Sect. 4.3), defect resolution (Sect. 4.4), and establishing a common architecture understanding (Sect. 4.5). Section 5 compared LEMMA to related approaches and concluded that LEMMA has particular strengths in (i) holistic MSA modeling based on viewpoint integration; (ii) language-level extensibility, enabling model-based reification of architecture patterns and selective technology-specificity; (iii) model processing, by bundling a specialized MPF together with sophisticated model processors for code generation and quality analysis; and (iv) versatility, making LEMMA applicable in microservice development, operation, architecture reconstruction, and quality assessment.

In our ongoing and future works, we combine LEMMA with formal techniques for correct microservice behavior specification [31, 32]. Moreover, while we have already empirically shown that LEMMA is effective for MSA modeling [112], we plan to evaluate it further in industry-related development processes of small- and medium-sized enterprises. In addition, two doctoral students currently improve LEMMA to (i) better integrate with distributed and non-modeling microservice teams [113, 115]; and (ii) increase the coverage and correctness of LEMMA-based reconstruction processes [126].

References

1. Alshuqayran, N., Ali, N., Evans, R.: Towards micro service architecture recovery: An empirical study. In: 2018 IEEE International Conference on Software Architecture (ICSA), pp. 47–56. IEEE, Piscataway (2018). https://doi.org/10.1109/ICSA.2018.00014
2. Ameller, D., Burgués, X., Collell, O., Costal, D., Franch, X., Papazoglou, M.P.: Development of service-oriented architectures using model-driven development: a mapping study. Informat. Softw. Technol. **62**, 42–66 (2015). Elsevier. https://doi.org/10.1016/j.infsof.2015.02.006
3. Athanasopoulos, D., Zarras, A.V., Miskos, G., Issarny, V., Vassiliadis, P.: Cohesion-driven decomposition of service interfaces without access to source code. IEEE Trans. Serv. Comput. **8**(4), 550–562 (2015). IEEE. https://doi.org/10.1109/TSC.2014.2310195
4. Ayas, H.M., Leitner, P., Hebig, R.: Facing the giant: A grounded theory study of decision-making in microservices migrations. In: Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), ESEM '21. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3475716.3475792

5. Baker, P., Loh, S., Weil, F.: Model-driven engineering in a large industrial context — Motorola case study. In: Briand, L., Williams, C. (eds.) Model Driven Engineering Languages and Systems, pp. 476–491. Springer, Berlin (2005). https://doi.org/10.1007/11557432_36
6. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Microservices architecture enables DevOps: migration to a cloud-native architecture. IEEE Softw. **33**(3), 42–52 (2016). IEEE. https://doi.org/10.1109/MS.2016.64
7. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Migrating to cloud-native architectures using microservices: An experience report. In: Celesti, A., Leitner, P. (eds.) Advances in Service-Oriented and Cloud Computing, pp. 201–215. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33313-7_15
8. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 3rd edn. Addison-Wesley, Boston (2013)
9. Bass, L., Klein, J.: Deployment and Operations for Software Engineers, 1st edn. Self-published (2019)
10. Benguria, G., Larrucea, X., Elvesæter, B., Neple, T., Beardsmore, A., Friess, M.: A platform independent model for service oriented architectures. In: Doumeingts, G., Müller, J., Morel, G., Vallespir, B. (eds.) Enterprise Interoperability, pp. 23–32. Springer, London (2007). https://doi.org/10.1007/978-1-84628-714-5_3
11. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend, 2nd edn. Packt Publishing, Birmingham (2016)
12. Bogner, J.: On the evolvability assurance of microservices: metrics, scenarios, and patterns. Ph.D. Thesis (2020). https://doi.org/10.18419/opus-10950
13. Bogner, J., Fritzsch, J., Wagner, S., Zimmermann, A.: Microservices in industry: Insights into technologies, characteristics, and software quality. In: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), pp. 187–195. IEEE, Piscataway (2019). https://doi.org/10.1109/ICSA-C.2019.00041
14. Cerny, T., Donahoo, M.J., Trnka, M.: Contextual understanding of microservice architecture: current and future directions. SIGAPP Appl. Comput. Rev. **17**(4), 29–45 (2018). ACM. https://doi.org/10.1145/3183628.3183631
15. Combemale, B., France, R.B., Jézéquel, J.M., Rumpe, B., Steel, J., Vojtisek, D.: Engineering Modeling Languages: Turning Domain Knowledge into Tools, 1st edn. CRC Press, Boca Raton (2017)
16. Cortellessa, V., Eramo, R., Tucci, M.: From software architecture to analysis models and back: model-driven refactoring aimed at availability improvement. Informat. Softw. Technol. **127** (2020). https://doi.org/10.1016/j.infsof.2020.106362
17. Di Francesco, P., Malavolta, I., Lago, P.: Research on architecting microservices: Trends, focus, and potential for industrial adoption. In: 2017 IEEE International Conference on Software Architecture (ICSA), pp. 21–30. IEEE, Piscataway (2017). https://doi.org/10.1109/ICSA.2017.24
18. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: Yesterday, today, and tomorrow. In: Mazzara, M., Meyer, B. (eds.) Present and Ulterior Software Engineering, pp. 195–216. Springer, Berlin (2017). https://doi.org/10.1007/978-3-319-67425-4_12
19. Dragoni, N., Lanese, I., Larsen, S.T., Mazzara, M., Mustafin, R., Safina, L.: Microservices: How to make your application scale. In: Petrenko, A.K., Voronkov, A. (eds.) Perspectives of System Informatics, pp. 95–104. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74313-4_8
20. Le, D.M., Dang, D.-H., Nguyen, V.-H.: Domain-driven design using meta-attributes: A DSL-based approach. In: 2016 Eighth International Conference on Knowledge and Systems Engineering (KSE), pp. 67–72. IEEE, Piscataway (2016). https://doi.org/10.1109/KSE.2016.7758031
21. Ecma International: The JSON data interchange syntax. Standard ECMA-404, Ecma International (2017)

22. Engel, T., Langermeier, M., Bauer, B., Hofmann, A.: Evaluation of microservice architectures: A metric and tool-based approach. In: Mendling, J., Mouratidis, H. (eds.) Information Systems in the Big Data Era, pp. 74–89. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92901-9_8
23. Erl, T.: Service-Oriented Architecture (SOA): Concepts, Technology and Design, 1st edn. Prentice Hall, Hoboken (2005)
24. Evans, E.: Domain-Driven Design, 1st edn. Addison-Wesley, Boston (2004)
25. Ferry, N., Rossini, A., Chauvel, F., Morin, B., Solberg, A.: Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In: 2013 IEEE Sixth International Conference on Cloud Computing, pp. 887–894. IEEE, Piscataway (2013). https://doi.org/10.1109/CLOUD.2013.133
26. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. Thesis (2000)
27. Fielding, R.T., Reschke, J.F.: Hypertext Transfer Protocol (HTTP/1.1): Semantics and content. RFC 7231, RFC Editor (2014)
28. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: 2007 Future of Software Engineering, FOSE '07, pp. 37–54. IEEE, Washington, (2007). https://doi.org/10.1109/FOSE.2007.14
29. Francesco, P.D., Lago, P., Malavolta, I.: Migrating towards microservice architectures: An industrial survey. In: 2018 IEEE International Conference on Software Architecture (ICSA), pp. 29–38. IEEE, Piscataway (2018). https://doi.org/10.1109/ICSA.2018.00012
30. Garriga, M.: Towards a taxonomy of microservices architectures. In: Cerone, A., Roveri, M. (eds.) Software Engineering and Formal Methods, pp. 203–218. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74781-1_15
31. Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F.: Model-driven generation of microservice interfaces: From LEMMA domain models to Jolie APIs. In: ter Beek, M.H., Sirjani, M. (eds.) Coordination Models and Languages, pp. 223–240. Springer, Berlin (2022). https://doi.org/10.1007/978-3-031-08143-9_13
32. Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F., Sachweh, S.: Jolie and LEMMA: Model-driven engineering and programming languages meet on microservices. In: Damiani, F., Dardha, O. (eds.) Coordination Models and Languages, pp. 276–284. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78142-2_17
33. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A., Smith, D., Bierman, G.: The Java language specification: Java se 17th edn. Specification JSR-392 Java SE 17, Oracle America, Inc. (2021)
34. Granchelli, G., Cardarelli, M., Francesco, P.D., Malavolta, I., Iovino, L., Salle, A.D.: Towards recovering the software architecture of microservice-based systems. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pp. 46–53. IEEE, Piscataway (2017). https://doi.org/10.1109/ICSAW.2017.48
35. Greifenberg, T., Hölldobler, K., Kolassa, C., Look, M., Mir Seyed Nazari, P., Müller, K., Navarro Perez, A., Plotnikov, D., Reiss, D., Roth, A., Rumpe, B., Schindler, M., Wortmann, A.: Integration of handwritten and generated object-oriented code. In: Desfray, P., Filipe, J., Hammoudi, S., Pires, L.F. (eds.) Model-Driven Engineering and Software Development, pp. 112–132. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27869-8_7
36. Gu, Q., Parkin, M., Lago, P.: A taxonomy of service engineering stakeholder types. In: Abramowicz, W., Llorente, I.M., Surridge, M., Zisman, A., Vayssière, J. (eds.) Towards a Service-Based Internet, pp. 206–219. Springer, Berlin (2011). https://doi.org/10.1007/978-3-642-24755-2_20
37. Harel, D., Rumpe, B.: Meaningful modeling: what's the semantics of "semantics"? Computer **37**(10), 64–72 (2004). IEEE. https://doi.org/10.1109/MC.2004.172
38. Haselböck, S., Weinreich, R., Buchgeher, G.: Decision models for microservices: Design areas, stakeholders, use cases, and requirements. In: Lopes, A., de Lemos, R. (eds.) Software Architecture, pp. 155–170. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65831-5_11

39. Haupt, F., Leymann, F., Scherer, A., Vukojevic-Haupt, K.: A framework for the structural analysis of REST APIs. In: 2017 IEEE International Conference on Software Architecture (ICSA), pp. 55–58. Springer, Berlin (2017). https://doi.org/10.1109/ICSA.2017.40
40. Herbst, N.R., Kounev, S., Reussner, R.: Elasticity in cloud computing: What it is, and what it is not. In: Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13), pp. 23–27. USENIX, San Jose (2013). https://www.usenix.org/conference/icac13/technical-sessions/presentation/herbst
41. Hirzalla, M., Cleland-Huang, J., Arsanjani, A.: A metrics suite for evaluating flexibility and complexity in service oriented architectures. In: Feuerlicht, G., Lamersdorf, W. (eds.) Service-Oriented Computing – ICSOC 2008 Workshops, pp. 41–52. Springer, Berlin (2009). https://doi.org/10.1007/978-3-642-01247-1_5
42. ISO/IEC: Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. Standard ISO/IEC 25010:2011(E), International Organization for Standardization/International Electrotechnical Commission (2011)
43. ISO/IEC/IEEE: Systems and software engineering — Architecture description. Standard ISO/IEC/IEEE 42010:2011(E), International Organization for Standardization/International Electrotechnical Commission/Institute of Electrical and Electronics Engineers (2011)
44. Jamshidi, P., Pahl, C., Mendonça, N.C., Lewis, J., Tilkov, S.: Microservices: The journey so far and challenges ahead. IEEE Softw. **35**(3), 24–35 (2018). IEEE. https://doi.org/10.1109/MS.2018.2141039
45. JHipster: JHipster Domain Language (JDL) (2023). https://www.jhipster.tech/jdl
46. Johanson, A., Flögel, S., Dullo, C., Hasselbring, W.: OceanTEA: Exploring ocean-derived climate data using microservices. In: Proceedings of the 6th International Workshop on Climate Informatics: CI 2016. National Center for Atmospheric Research (2016)
47. Johnson, R.E., Foote, B.: Designing reusable classes. J. Object-Oriented Programm. **1**(2), 22–35 (1988). SIGS Publications
48. Jongeling, R., Carlson, J., Cicchetti, A.: Impediments to introducing continuous integration for model-based development in industry. In: 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 434–441. IEEE, Piscataway (2019). https://doi.org/10.1109/SEAA.2019.00071
49. Kahani, N., Bagherzadeh, M., Cordy, J.R., Dingel, J., Varró, D.: Survey and classification of model transformation tools. Softw. Syst. Model. **18**(4), 2361–2397 (2019). Springer. https://doi.org/10.1007/s10270-018-0665-6
50. Kapferer, S., Zimmermann, O.: Domain-specific language and tools for strategic domain-driven design, context mapping and bounded context modeling. In: Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD, pp. 299–306. INSTICC, SciTePress (2020). https://doi.org/10.5220/0008910502990306
51. Kirchhof, J.C., Rumpe, B., Schmalzing, D., Wortmann, A.: Montithings: Model-driven development and deployment of reliable IoT applications. J. Syst. Softw. **183**, 111087 (2022). https://doi.org/10.1016/j.jss.2021.111087
52. Knoche, H., Hasselbring, W.: Drivers and barriers for microservice adoption – a survey among professionals in Germany. Enterprise Modell. Informat. Syst. Architect. **14**(1), 1–35 (2019). German Informatics Society. https://doi.org/10.18417/emisa.14.1
53. Lano, K., Kolahdouz-Rahimi, S.: Model-transformation design patterns. IEEE Trans. Softw. Eng. **40**(12), 1224–1259 (2014).. IEEE https://doi.org/10.1109/TSE.2014.2354344
54. Le, V.D., Neff, M.M., Stewart, R.V., Kelley, R., Fritzinger, E., Dascalu, S.M., Harris, F.C.: Microservice-based architecture for the NRDC. In: 2015 IEEE 13th International Conference on Industrial Informatics (INDIN), pp. 1659–1664. IEEE, Piscataway (2015). https://doi.org/10.1109/INDIN.2015.7281983
55. Luz, W.P., Pinto, G., Bonifácio, R.: Building a collaborative culture: A grounded theory of well succeeded DevOps adoption in practice. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '18,

pp. 6:1–6:10. ACM, New York (2018). https://doi.org/10.1145/3239235.3240299
56. Márquez, G., Villegas, M.M., Astudillo, H.: A pattern language for scalable microservices-based systems. In: Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings, ECSA '18, pp. 24:1–24:7. ACM, New York (2018). https://doi.org/10.1145/3241403.3241429
57. Mazlami, G., Cito, J., Leitner, P.: Extraction of microservices from monolithic software architectures. In: 2017 IEEE International Conference on Web Services (ICWS), pp. 524–531. IEEE, Piscataway (2017). https://doi.org/10.1109/ICWS.2017.61
58. Mazzara, M., Dragoni, N., Bucchiarone, A., Giaretta, A., Larsen, S.T., Dustdar, S.: Microservices: migration of a mission critical system. IEEE Trans. Serv. Comput., 1–14 (2018). IEEE. https://doi.org/10.1109/TSC.2018.2889087
59. Mens, T., Gorp, P.V.: A taxonomy of model transformation. Electron. Notes Theoret. Comput. Sci. **152**, 125–142 (2006). Elsevier. https://doi.org/10.1016/j.entcs.2005.10.021
60. Methodologies Corporation: Service-oriented modeling framework (SOMF) version 2.1. (2011)
61. Mohagheghi, P., Dehlen, V.: Where is the proof? - A review of experiences from applying MDE in industry. In: Schieferdecker, I., Hartman, A. (eds.) Model Driven Architecture – Foundations and Applications, pp. 432–443. Springer, Berlin (2008). https://doi.org/10.1007/978-3-540-69100-6_31
62. Mohamed, M.A., Challenger, M., Kardas, G.: Applications of model-driven engineering in cyber-physical systems: a systematic mapping study. J. Comput. Lang. **59**, 1–54 (2020). https://doi.org/10.1016/j.cola.2020.100972
63. Márquez, G., Astudillo, H.: Actual use of architectural patterns in microservices-based open source projects. In: 2018 25th Asia-Pacific Software Engineering Conference (APSEC), pp. 31–40. IEEE, Piscataway (2018). https://doi.org/10.1109/APSEC.2018.00017
64. Nadareishvili, I., Mitra, R., McLarty, M., Amundsen, M.: Microservice Architecture: Aligning Principles, Practices, and Culture, 1st edn. O'Reilly, Sebastopol (2016)
65. Newman, S.: Building Microservices: Designing Fine-Grained Systems, 1st edn. O'Reilly, Sebastopol (2015)
66. OASIS: Reference architecture foundation for Service Oriented Architecture version 1.0. Standard OASIS Committee Specification 01, Organization for the Advancement of Structured Information Standards (2012)
67. OASIS: Topology and orchestration specification for cloud applications version 1.0. Standard, Organization for the Advancement of Structured Information Standards (2013)
68. OMG: Service oriented architecture Modeling Language (SoaML) specification version 1.0.1. Standard, Object Management Group (2012)
69. OMG: Business Process Model and Notation (BPMN) version 2.0.2. Standard formal/2013-12-09, Object Management Group (2013)
70. OMG: Model Driven Architecture (MDA) MDA Guide rev. 2.0. Standard ormsc/2014-06-01, Object Management Group (2014)
71. OMG: Object Constraint Language version 2.4. Standard formal/2014-02-03, Object Management Group (2014)
72. OMG: XML Metadata Interchange (XMI) specification. Standard formal/2015-06-07, Object Management Group (2015)
73. OMG: OMG Unified Modeling Language (OMG UML) version 2.5.1. Standard formal/17-12-05, Object Management Group (2017)
74. Ponce, F., Soldani, J., Astudillo, H., Brogi, A.: Smells and refactorings for microservices security: a multivocal literature review. J. Syst. Softw. **192**, 111393 (2022). https://doi.org/10.1016/j.jss.2022.111393
75. Rademacher, F.: Genlets for LEMMA's JBG on Software Heritage. https://archive.softwareheritage.org/browse/directory/7dccb79b9804d8d9459c86ba9721e1197f59b865/?origin_url=https://github.com/SeelabFhdo/lemma&path=code%20generators
76. Rademacher, F.: Grammar specification of LEMMA's Domain Data Modeling Language on Software Heritage. https://archive.softwareheritage.org/browse/origin/content/?origin_

url=https://github.com/SeelabFhdo/lemma&path=de.fhdo.lemma.data.datadsl/src/de/fhdo/lemma/data/DataDsl.xtext
77. Rademacher, F.: Grammar specification of LEMMA's Operation Modeling Language on Software Heritage. https://archive.softwareheritage.org/browse/origin/content/?origin_url=https://github.com/SeelabFhdo/lemma&path=de.fhdo.lemma.operationdsl/src/de/fhdo/lemma/operationdsl/OperationDsl.xtext
78. Rademacher, F.: Grammar specification of LEMMA's Service Modeling Language on Software Heritage. https://archive.softwareheritage.org/browse/origin/content/?origin_url=https://github.com/SeelabFhdo/lemma&path=de.fhdo.lemma.servicedsl/src/de/fhdo/lemma/ServiceDsl.xtext
79. Rademacher, F.: Grammar specification of LEMMA's Service Technology Mapping Modeling Language on Software Heritage. https://archive.softwareheritage.org/browse/origin/content/?origin_url=https://github.com/SeelabFhdo/lemma&path=de.fhdo.lemma.technology.mappingdsl/src/de/fhdo/lemma/technology/mappingdsl/MappingDsl.xtext
80. Rademacher, F.: Grammar specification of LEMMA's Technology Modeling Language on Software Heritage. https://archive.softwareheritage.org/browse/origin/content/?origin_url=https://github.com/SeelabFhdo/lemma&path=de.fhdo.lemma.technology.technologydsl/src/de/fhdo/lemma/technology/TechnologyDsl.xtext
81. Rademacher, F.: LEMMA model processing example on Software Heritage. https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/SeelabFhdo/lemma&path=examples/model-processing
82. Rademacher, F.: LEMMA's Java Base Generator on Software Heritage. https://archive.softwareheritage.org/browse/directory/7dccb79b9804d8d9459c86ba9721e1197f59b865/?origin_url=https://github.com/SeelabFhdo/lemma&path=code%20generators/de.fhdo.lemma.model_processing.code_generation.java_base
83. Rademacher, F.: LEMMA's Model Processing Framework on Software Heritage. https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/SeelabFhdo/lemma&path=de.fhdo.lemma.model_processing
84. Rademacher, F.: LEMMA's static analysis library on Software Heritage. https://archive.softwareheritage.org/browse/directory/7dccb79b9804d8d9459c86ba9721e1197f59b865/?origin_url=https://github.com/SeelabFhdo/lemma&path=de.fhdo.lemma.analyzer.lib
85. Rademacher, F.: LEMMA's static analyzer on Software Heritage. https://archive.softwareheritage.org/browse/directory/7dccb79b9804d8d9459c86ba9721e1197f59b865/?origin_url=https://github.com/SeelabFhdo/lemma&path=de.fhdo.lemma.analyzer
86. Rademacher, F.: Metamodel implementation of LEMMA's Domain Data Modeling Language on Software Heritage. https://archive.softwareheritage.org/browse/origin/content/?origin_url=https://github.com/SeelabFhdo/lemma&path=de.fhdo.lemma.data.datadsl.metamodel/model/DataViewpointModel.xcore
87. Rademacher, F.: Metamodel implementation of LEMMA's Operation Modeling Language on Software Heritage. https://archive.softwareheritage.org/browse/origin/content/?origin_url=https://github.com/SeelabFhdo/lemma&path=de.fhdo.lemma.operationdsl.metamodel/model/OperationViewpointModel.xcore
88. Rademacher, F.: Metamodel implementation of LEMMA's Service Modeling Language on Software Heritage. https://archive.softwareheritage.org/browse/origin/content/?origin_url=https://github.com/SeelabFhdo/lemma&path=/de.fhdo.lemma.servicedsl.metamodel/model/ServiceViewpointModel.xcore
89. Rademacher, F.: Metamodel implementation of LEMMA's Service Technology Mapping Modeling Language on Software Heritage. https://archive.softwareheritage.org/browse/origin/content/?origin_url=https://github.com/SeelabFhdo/lemma&path=de.fhdo.lemma.technology.mappingdsl.metamodel/model/TechnologyMappingModel.xcore
90. Rademacher, F.: Metamodel implementation of LEMMA's Technology Modeling Language on Software Heritage. https://archive.softwareheritage.org/browse/origin/content/?origin_url=https://github.com/SeelabFhdo/lemma&path=de.fhdo.lemma.technology.technologydsl.metamodel/model/TechnologyDefinitionModel.xcore

91. Rademacher, F.: Service model for the `OrderService` on Software Heritage. https://archive.softwareheritage.org/browse/origin/content/?origin_url=https://github.com/frademacher/dissertation-supplemental-material&path=chapters-5-6-concrete-syntax-example-and-intermediate-models/Order/Order.services
92. Rademacher, F.: An overview of modeling concepts for service-based software architectures. In: Software Engineering Publications. Kasseler Online Bibliothek, Repository und Archiv (KOBRA) (2020). https://doi.org/10.17170/kobra-202008191601
93. Rademacher, F.: A language ecosystem for modeling microservice architecture. Ph.D. Thesis, University of Kassel (2022). https://doi.org/10.17170/kobra-202209306919. https://kobra.uni-kassel.de/handle/123456789/14176
94. Rademacher, F., Peters, M., Sachweh, S.: Design of a domain-specific language based on a technology-independent web service framework. In: Weyns, D., Mirandola, R., Crnkovic, I. (eds.) Software Architecture, pp. 357–371. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23727-5_29
95. Rademacher, F., Sachweh, S., Zündorf, A.: Aspect-oriented modeling of technology heterogeneity in microservice architecture. In: 2019 IEEE International Conference on Software Architecture (ICSA), pp. 21–30. IEEE, Piscataway (2019). https://doi.org/10.1109/ICSA.2019.00011
96. Rademacher, F., Sachweh, S., Zündorf, A.: 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), pp. 229–236. IEEE, Piscataway (2020). https://doi.org/10.1109/SEAA51224.2020.00047
97. Rademacher, F., Sachweh, S., Zündorf, A.: A modeling method for systematic architecture reconstruction of microservice-based software systems. In: Nurcan, S., Reinhartz-Berger, I., Soffer, P., Zdravkovic, J. (eds.) Enterprise, Business-Process and Information Systems Modeling, pp. 311–326. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-49418-6_21
98. Rademacher, F., Sorgalla, J., Wizenty, P., Sachweh, S., Zündorf, A.: Graphical and textual model-driven microservice development. In: Bucchiarone, A., Dragoni, N., Dustdar, S., Lago, P., Mazzara, M., Rivera, V., Sadovykh, A. (eds.) Microservices: Science and Engineering, pp. 147–179. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-31646-4_7
99. Rademacher, F., Sorgalla, J., Wizenty, P., Trebbau, S.: Towards an extensible approach for generative microservice development and deployment using LEMMA. In: Scandurra, P., Galster, M., Mirandola, R., Weyns, D. (eds.) Software Architecture, pp. 257–280. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-15116-3_12
100. Rahman, M.I., Panichella, S., Taibi, D.: A curated dataset of microservices-based systems. In: Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution, pp. 1–9. CEUR-WS (2019). http://ceur-ws.org/Vol-2520/paper1a.pdf
101. Richards, M.: Software Architecture Patterns, 1st edn. O'Reilly, Sebastopol (2015)
102. Richardson, C.: Microservices Patterns, 1st edn. Manning Publications, Shelter Island (2019)
103. Rumpe, B.: Agile Modeling with UML, 1st edn. Springer, Berlin (2017)
104. Ruscio, D.D., Malavolta, I., Muccini, H., Pelliccione, P., Pierantonio, A.: Developing next generation ADLs through MDE techniques. In: 2010 ACM/IEEE 32nd International Conference on Software Engineering, vol. 1, pp. 85–94. IEEE, Piscataway (2010). https://doi.org/10.1145/1806799.1806816
105. Schauerhuber, A., Schwinger, W., Kapsammer, E., Retschitzegger, W., Wimmer, M., Kappel, G.: A survey on aspect-oriented modeling approaches. Technical Report, Vienna University of Technology (2007)
106. Schermann, G., Cito, J., Leitner, P.: All the services large and micro: Revisiting industrial practice in services computing. In: Norta, A., Gaaloul, W., Gangadharan, G.R., Dam, H.K. (eds.) Service-Oriented Computing – ICSOC 2015 Workshops, pp. 36–47. Springer, Berlin (2016). https://doi.org/10.1007/978-3-662-50539-7_4

107. Sobernig, S., Zdun, U.: Inversion-of-control layer. In: Proceedings of the 15th European Conference on Pattern Languages of Programs, EuroPLoP '10, pp. 1–22. ACM, New York (2010). https://doi.org/10.1145/2328909.2328935
108. [Software] Florian Rademacher: Language Ecosystem for Modeling Microservice Architecture (LEMMA). VCS: https://www.github.com/SeelabFhdo/lemma, SWHID: <swh:1:dir:4ac248661825a16a18a88b976734455f601e0d85;origin=https://github.com/SeelabFhdo/lemma;visit=swh:1:snp:f57caa7209e46735adc66f1cb937a606b4466556;anchor=swh:1:rev:22fd04c6b8a4cb126334db40c331f90ca9730606> (2022)
109. Soldani, J., Tamburri, D.A., Heuvel, W.J.V.D.: The pains and gains of microservices: a systematic grey literature review. J. Syst. Softw. **146**, 215–232 (2018). Elsevier. https://doi.org/10.1016/j.jss.2018.09.082
110. Soliman, M., Riebisch, M., Zdun, U.: Enriching architecture knowledge with technology design decisions. In: 2015 12th Working IEEE/IFIP Conference on Software Architecture, pp. 135–144. IEEE, Piscataway (2015). https://doi.org/10.1109/WICSA.2015.14
111. Soltesz, S., Pötzl, H., Fiuczynski, M.E., Bavier, A., Peterson, L.: Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys '07, pp. 275–287. ACM, New York (2007). https://doi.org/10.1145/1272996.1273025
112. Sorgalla, J., Rademacher, F., Sachweh, S., Zündorf, A.: Modeling microservice architecture: A comparative experiment towards the effectiveness of two approaches. In: Proceedings of the 35th Annual ACM Symposium on Applied Computing, SAC '20, p. 1506–1509. ACM, New York (2020). https://doi.org/10.1145/3341105.3374065
113. Sorgalla, J., Sachweh, S., Zündorf, A.: Exploring the microservice development process in small and medium-sized organizations. In: Morisio, M., Torchiano, M., Jedlitschka, A. (eds.) Product-Focused Software Process Improvement, pp. 453–460. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64148-1_28
114. Sorgalla, J., Wizenty, P., Rademacher, F., Sachweh, S., Zündorf, A.: AjiL: Enabling model-driven microservice development. In: Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings, ECSA '18, pp. 1:1–1:4. ACM, New York (2018). https://doi.org/10.1145/3241403.3241406
115. Sorgalla, J., Wizenty, P., Rademacher, F., Sachweh, S., Zündorf, A.: Applying model-driven engineering to stimulate the adoption of devops processes in small and medium-sized development organizations. SN Comput. Sci. **2**(6), 459 (2021). https://doi.org/10.1007/s42979-021-00825-z
116. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Boston (2008)
117. Stoermer, C., Rowe, A., O'Brien, L., Verhoef, C.: Model-centric software architecture reconstruction. Softw. Practice Exper. **36**(4), 333–363 (2006). Wiley. https://doi.org/10.1002/spe.699
118. Taibi, D., Lenarduzzi, V.: On the definition of microservice bad smells. IEEE Softw. **35**(3), 56–62 (2018). IEEE. https://doi.org/10.1109/MS.2018.2141031
119. Taibi, D., Lenarduzzi, V., Pahl, C.: Processes, motivations, and issues for migrating to microservices architectures: an empirical investigation. IEEE Cloud Comput. **4**(5), 22–32 (2017). IEEE. https://doi.org/10.1109/MCC.2017.4250931
120. Terzić, B., Dimitrieski, V., Kordić, S., Milosavljević, G., Luković, I.: Development and evaluation of MicroBuilder: a model-driven tool for the specification of REST microservice software architectures. Enterprise Informat. Syst. **12**(8-9), 1034–1057 (2018). Taylor & Francis. https://doi.org/10.1080/17517575.2018.1460766
121. The Open Group: SOA reference architecture. C119 (2011)
122. Trebbau, S., Wizenty, P., Sachweh, S.: Towards integrating blockchains with microservice architecture using model-driven engineering. In: Gregory, P., Kruchten, P. (eds.) Agile Processes in Software Engineering and Extreme Programming – Workshops, pp. 167–175. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-88583-0_16

123. Whittle, J., Hutchinson, J., Rouncefield, M.: The state of practice in model-driven engineering. IEEE Softw. **31**(3), 79–85 (2014). IEEE. https://doi.org/10.1109/MS.2013.65
124. Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H., Heldal, R.: Industrial adoption of model-driven engineering: Are the tools really the problem? In: Model-Driven Engineering Languages and Systems: 16th International Conference, MODELS 2013, Miami, FL, USA, September 29–October 4, 2013. Proceedings 16, pp. 1–17. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-41533-3_1
125. Wizenty, P., Ponce, F., Rademacher, F., Soldani, J., Astudillo, H., Brogi, A., Sachweh, S.: Towards resolving security smells in microservices, model-driven. In: Proceedings of the 18th International Conference on Software Technologies (ICSOFT 2023). To appear
126. Wizenty, P., Rademacher, F.: Towards viewpoint-based microservice architecture reconstruction. In: Abstracts of the Fourth International Conference on Microservices (Microservices 2022). Microservices Community (2022). https://www.conf-micro.services/2022/papers/paper_16.pdf
127. Wortmann, A., Barais, O., Combemale, B., Wimmer, M.: Modeling languages in industry 4.0: an extended systematic mapping study. Softw. Syst. Model. **19**(1), 67–94 (2020). https://doi.org/10.1007/s10270-019-00757-6
128. Zimmermann, O.: Microservices tenets. Comput. Sci. Res. Develop. **32**(3–4), 301–310 (2017). Springer. https://doi.org/10.1007/s00450-016-0337-0
129. Zimmermann, O., Stocker, M., Lübke, D., Pautasso, C., Zdun, U.: Introduction to microservice API patterns (MAP). In: Cruz-Filipe, L., Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F., Sachweh, S. (eds.) Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019), OpenAccess Series in Informatics (OASIcs), vol. 78, pp. 4:1–4:17. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2020). https://doi.org/10.4230/OASIcs.Microservices.2017-2019.4
130. Zimmermann, O., Stocker, M., Lübke, D., Zdun, U., Pautasso, C.: Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges. Addison-Wesley Signature Series (Vernon). Addison-Wesley Professional, Boston (2022)