Correct-By-Construction Microservices with Model-Driven Engineering

The Case for Architectural Pattern Conformance Checking and Pattern-Conform Code Generation

Florian Rademacher[1]
Software Engineering, RWTH Aachen University, Germany

Abstract

Patterns are a common metaphor in software engineering that denotes reusable solutions for recurring software engineering problems. Architectural patterns focus on the interplay or organization of two or more components of a software system, and are particularly helpful in the design of complex software architectures such as those produced by Microservice Architecture (MSA).

This paper presents an approach for the language-based reification of architectural MSA patterns. To this end, we introduce a method to flexibly retrofit LEMMA (Language Ecosystem for Modeling Microservices) with support for modeling and implementing architectural MSA patterns. The method relies on the (i) specification of aspects to reify pattern applications in MSA models; (ii) validation of pattern applications; and (iii) code generation from correct pattern applications. Consequently, it contributes to correct-by-construction microservices by abstracting from the complexity of pattern implementations, yet still enabling their automated production with Model-Driven Engineering. We validate our method with the popular Domain Event and Command Query Responsibility Segregation patterns, and assess its applicability for 28 additional patterns. Our results show that LEMMA's expressivity covers the model-based expression of complex architectural MSA patterns and that its model processing facilities support pattern-specific extensions such that conformance checking and pattern-conform code generation can be modularized into reusable model processors.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Software and its engineering → Software design engineering; Software and its engineering → Source code generation; Software and its engineering → Model-driven software engineering; Software and its engineering → Software architectures; Software and its engineering → Cloud computing

Keywords and phrases Microservice Architecture, Architectural Patterns, Model-Driven Engineering, Static Model Analysis, Model Validation, Code Generation, Architectural Pattern Conformance Checking

Digital Object Identifier 10.4230/OASIcs.Microservices.2020-2022.8

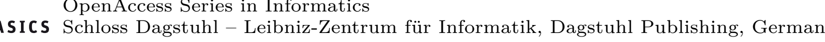
Related Version *Previous Version*: `Microservices 2020 Extended Abstract`

Supplementary Material *Software (Language Ecosystem for Modeling Microservice Architecture (LEMMA))*: `https://github.com/SeelabFhdo/lemma`
archived at `swh:1:dir:4ac248661825a16a18a88b976734455f601e0d85`

1 Introduction

Patterns are a common metaphor in software engineering that emerged from urban design [1] and denotes reusable solutions for recurring software engineering problems [20]. Patterns are particularly useful to codify and communicate knowledge gained from software engineering experience [69, 68].

[1] This work was conducted while working at IDiAL Institute, University of Applied Sciences and Arts Dortmund, Germany.

© Florian Rademacher;
licensed under Creative Commons License CC-BY 4.0
Joint Post-proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022).
Editors: Gokila Dorai, Maurizio Gabbrielli, Giulio Manzonetto, Aomar Osmani, Marco Prandini, Gianluigi Zavattaro, and Olaf Zimmerman; Article No. 8; pp. 8:1–8:25

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Considering their area of impact, patterns can be divided into two major categories. *Design patterns* [20] focus on software component internals whereas *architectural patterns* [8] recognize the interplay or organization of two or more components [68]. This paper focuses on the latter category in the context of Microservice Architecture (MSA) [36]. The inception of catalogs of architectural patterns for MSA engineering [55, 70] and the study of patterns' practical adoption [35, 64] show their relevance in dealing with complexity in microservice architecture design, implementation, and operation [59]. *Language-based approaches to MSA engineering* [21] constitute a paradigm that promotes the use of MSA-oriented software languages to conceive and deploy microservice architectures, thereby coping with MSA's complexity by specialized language primitives, e.g., for microservice interfaces or containers.

This paper presents an approach that combines both means for handling complexity in MSA engineering, i.e., architectural patterns and MSA-oriented software languages. More precisely, we show how to retrofit LEMMA – an ecosystem of modeling languages and model processing facilities for MSA [49] following Model-Driven Engineering (MDE) [10] – with architectural pattern support with the goal to enable *correct-by-construction microservices* via pattern conformance checking and pattern-conform code generation. This paper revises our abstract [48] accepted at the Third International Conference on Microservices as follows:

- Introduction of a reusable method including all steps to systematically extend LEMMA with support for architectural MSA patterns.
- Consideration of code generation as a step to map LEMMA models with architectural patterns to pattern-conform, executable microservices.
- Approach validation with a case study beyond the scope of the abstract's running example.

With these contributions, we continue our line of work on the intersection of MDE and MSA by applying LEMMA's means for the model-based capturing of stakeholder-oriented concerns in MSA engineering [49], enrichment of MSA models with metadata [51], and model validation and code generation [21] to (i) integrate pattern concepts in architecture models; and (ii) ensure pattern conformance on the architecture design and implementation level. The paper specifically goes beyond previous publications [53, 54] which neither considered pattern conformance checking nor pattern-conform code generation.

The remainder of the paper is structured as follows. Section 2 provides an overview of architectural MSA patterns. Section 3 describes our method to systematically extend LEMMA with architectural pattern support. Section 4 validates our approach. Sections 5 and 6 compare our approach to related work and conclude the paper, respectively.

2 Overview of Microservice Architecture Patterns

To make the paper self-contained, we give a non-exhaustive overview of architectural MSA patterns, thereby relying on corresponding pattern catalogs [55, 70] and empirical studies [35, 64]. We structure the patterns by the categories introduced by Márquez and Astudillo [35].

Communication Patterns

Communication patterns deal with issues in microservice interaction. The API Gateway pattern [35, 55] proposes the provisioning of a component that acts as a façade to microservices' functionality, i.e., it exposes only relevant functionality to architecture-external clients. With this characteristic, the pattern gives rise to API Composition [55, 64], Gateway Routing [64], and Gateway Offloading [64]. While API Gateway requests are often synchronous, patterns like Domain Event [55] and Event Sourcing [55] concern

asynchronous, event-based service interaction. The former pattern identifies domain concepts as events and the latter records event instances, e.g., for auditing purposes. Similarly, the Log Aggregator pattern [35, 55] centralizes the collection of additional log data.

Deployment Patterns

Deployment patterns focus on microservice deployment and deployment pipeline maintenance. The Sidecar pattern [55, 64] isolates and reuses functionality that crosscuts microservices, e.g., logging or configuration. Sidecar services cluster such functionality and provide business-oriented microservices with access. A more intrusive variant of this pattern is Microservice Chassis [55] which expects microservices' business logic to delegate handling of crosscutting concerns to programming frameworks. The Backend for Frontend [35, 55, 64] and Command Query Responsibility Segregation (CQRS) [55, 64] patterns provide microservice clients with interfaces for special needs. The former suggests several API Gateways, each dedicated to certain client needs or types. The CQRS pattern, on the other hand, considers a *logical microservice* to consist of (i) a physical *command-side microservice* that enables clients to change data; and (ii) one or more physical *query-side microservices* that enable clients to access data in client-specific representations. Asynchronous messaging ensures the synchronization between command-side and query-side microservices. The Database is the Service pattern [35] lets each microservice store its data in an isolated database to foster scalability.

Design Patterns

This category clusters patterns that focus on the architectural interplay of microservices, thereby regarding microservices as black boxes and allow pattern adoption at design time. We consider all *API design patterns* from the catalog of Zimmermann et al. [70] to belong to this category. The catalog organizes patterns along different dimensions in API design, e.g., Responsibility, Structure, or Quality. For example, the Processing Resource pattern (Responsibility) supports clustered exposure of application-level functionality in the form of activities or commands. The Parameter Tree pattern (Structure) concerns the tree-based design of request and response data structures that comprise containment relationships. On the Quality level, the API Key pattern [55, 70] secures API access by requiring eligible clients to provide a unique and valid *access token* for subsequent API access.

DevOps Patterns

DevOps patterns bridge between developer and operator concerns in MSA. Using the Externalized Configuration pattern [55, 64], microservices receive configuration values such as credentials or network location at runtime from specialized configuration services or stores. The Monitor patterns [35], and derivatives like Application Metrics [55], Distributed Tracing [55], Exception Tracking [55], and Health Check [35, 55], cover monitoring as a central DevOps practice [6]. With these patterns, microservices report events or status updates to a centralized server for visualization and analysis.

Migration Patterns

These patterns guide the decomposition of monoliths into microservice architectures [35]. The Strangler pattern [55, 64] suggests continuous decomposition by (i) steady migration of existing functionality to microservices; and (ii) direct realization of new functionality

in microservices. The ANTI-CORRUPTION LAYER pattern [64, 55] defines façades between monolith and microservice architecture for translating between deviating data representations. Patterns like DECOMPOSE BY BUSINESS CAPABILITY (a microservice covers a business capability) and DECOMPOSE BY SUBDOMAIN (a microservice covers a part of the application domain) prescribe strategies to assign functionalities to microservices [55].

Orchestration Patterns

These patterns foster the orchestration of microservices as business process participants. The CONTAINER pattern [35, 55], which suggests to run microservices in lightweight virtualized environments [60], is a key enabler for horizontal scaling in MSA [12]. Patterns like SERVICE REGISTRY [35, 55] and SERVICE DISCOVERY [35, 55] abstract from service instances' network locations and support service interaction via logical names. It is possible to combine the SERVICE DISCOVERY pattern with the LOAD BALANCER pattern [35] to optimize request routing to services with spare processing resources. In such scenarios, the CIRCUIT BREAKER pattern [35, 55] increases reliability as it caps the number of faulty interaction attempts to prevent failure cascades. The SAGA pattern [55] increases data consistency in distributed transactions by dividing them into *local transactions* executed by dedicated microservices. If a local transaction fails, the responsible microservice invokes a *compensating transaction* for rollback and informs its predecessor in the transaction chain to also rollback.

3 A LEMMA-Based Method for Pattern-Conform Microservice Design and Implementation

This section presents our method to retrofit LEMMA with architectural MSA pattern support – including pattern conformance checking and pattern-conform code generation for correct-by-construction microservices[2]. Figure 1 shows the specification of the method.

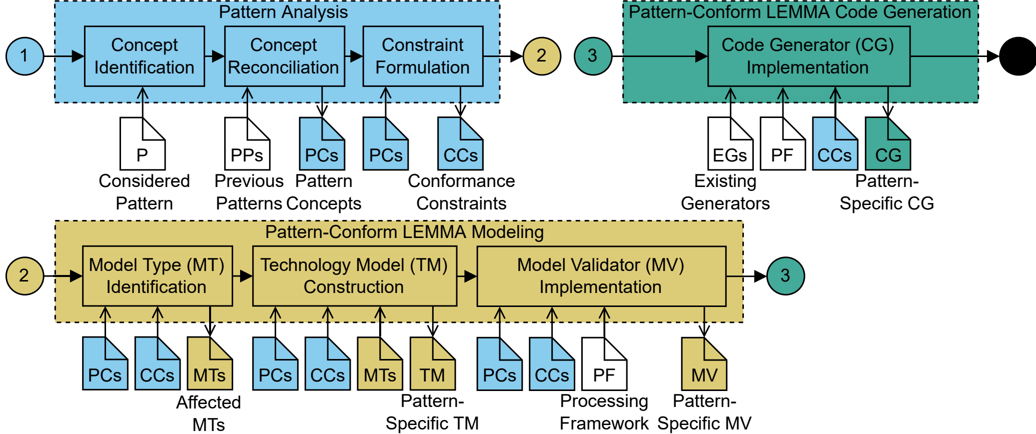


Figure 1 Overview of our method to retrofit LEMMA with architectural MSA pattern support. Method phases and activities are depicted as rectangles with dashed and solid lines, respectively. Required or produced artifacts are depicted as note sheets colored by their producing phase, if any.

[2] While we aim for the section to be understandable in a self-contained fashion, Appendices A and B yet provide additional background information on MDE and LEMMA.

Sections 3.1 to 3.3 describe the phases of the method and illustrate its instantiation on the example of the Domain Event pattern (Sect. 2).

3.1 Phase 1: Pattern Analysis

The first phase of the method consists of the three activities Concept Identification, Concept Reconciliation, and Constraint Formulation.

Concept Identification

This activity expects as input the pattern targeted by the method instance including its definition. For the Domain Event pattern, we refer to Richardson [55] who defines a domain event as a domain concept that conveys asynchronous messages between microservices.

Concept Reconciliation

The Concept Reconciliation activity expects the definitions and LEMMA-based specifications (Sect. 3.2) of all patterns covered by previous method instances as they might impact handling of the input pattern and its concepts. Suppose that the method instance for Domain Event is followed by an instance for the Event Sourcing pattern (Sect. 2). Since the latter leverages domain events, the integration of the former with LEMMA likely impacts the latter's integration. For example, it will not be necessary to treat a domain event as a genuine concept of the Event Sourcing pattern. On the other hand, the Concept Reconciliation activity helps to identify dependencies between pattern-specific LEMMA model validators (Sect. 3.2) and code generators (Sect. 3.3). For instance, the Event Sourcing pattern could require that domain events conforming to the Domain Event pattern are valid and manifested in source code so that an Event Sourcing code generator can refer to them.

The Concept Reconciliation phase yields a concept catalog for the input pattern. For Domain Event, this catalog defines the Domain Event concept as a structured domain concept representing asynchronously exchanged messages. Additionally, following the pattern's definition [55], the catalog has an entry for the Producer and Consumer concepts which denote microservices that send and receive domain events, respectively.

Constraint Formulation

The Constraint Formulation activity specifies conformance constraints for the concepts of the input pattern w.r.t. its definition. Table 1 lists constraints for the Domain Event pattern.

Table 1 Conformance constraints for the Domain Event pattern.

#	Constraint	Severity
C.1	Producer operations must be able to send asynchronous messages.	Error
C.2	Consumer operations must be able to receive asynchronous messages.	Error

Constraint C.1 prescribes that microservice operations which produce domain events must actually be able to send asynchronous messages. Constraint C.2 demands that operations of event consumers must actually be able to receive asynchronous messages. Both constraints exhibit the severity level "Error" so that their violation prevents further steps like code

generation. By contrast, the severity level "Warning" would identify constraint violations that still permit subsequent processing. An example for such a violation could be a microservice operation that uses a domain event in a synchronous interaction.

3.2 Phase 2: Pattern-Conform LEMMA Modeling

This phase extends LEMMA for modeling and applying the input pattern. It involves the activities Model Type Identification, Technology Model Construction, and Model Validator Implementation.

Model Type Identification

This activity requires the pattern concepts and conformance constraints from the previous activities (Sect. 3.1) to determine the LEMMA model types (Appendix B) for pattern application. In some cases, the LEMMA modeling language for the construction of models of a certain type already comprises native constructs for pattern application. For the Domain Event pattern, LEMMA's Domain Data Modeling Language supports direct modeling of structured domain events with the `domainEvent` keyword as illustrated in Listing 1.

Listing 1 Excerpt of a LEMMA domain model illustrating domain event definition.

```
1 // Model name: ChargingStationManagement.data
2 structure ParkingAreaCreatedEvent<domainEvent> {
3   immutable long commonId,
4   immutable ParkingAreaInformation info
5 }
```

In LEMMA's Domain Data Modeling Language, a domain event is a structured domain concept that typically consists of one or more immutable fields [16]. Instances of `ParkingAreaCreatedEvent` gather a value of the primitive type `long`[3] in the field `commonId` and an instance of the structured domain concept `ParkingAreaInformation`[4] in the field `info`.

Hence, a LEMMA model type affected by extending LEMMA with Domain Event support is the domain model type because LEMMA's Domain Data Modeling Language provides native support for domain event modeling and thus the expression of the Domain Event concept from the pattern's concept catalog (Sect. 3.1). By contrast to other MDE-for-MSA approaches [2, 63, 61, 25], LEMMA does not integrate pattern-specific keywords to foster language learnability and model comprehension. As a result, LEMMA does not provide modeling constructs for Domain Event Producers and Consumers (Sect. 3.1). To retrofit these concepts, the following Technology Model Construction activity relies on the *technology aspect* mechanism [51] of LEMMA's Technology Modeling Language (Appendix B), making the technology model type also affected by LEMMA Domain Event support. Similarly, the service and mapping model types (Appendix B) are affected as both enable assignment of aspects to microservices, e.g., to mark operations as Domain Event Producers.

Technology Model Construction

LEMMA's technology aspects can be exploited as a flexible mechanism to augment model elements with metadata. We rely on such metadata to reify pattern concepts and applications in LEMMA models. Listing 2 shows LEMMA's Domain Event technology model.

[3] LEMMA's type system integrates all primitive types of Java [22].
[4] For the sake of brevity, we omit the specification of this domain concept and refer to the complete domain model on Software Heritage [46].

Listing 2 Domain Event technology model in LEMMA's Technology Modeling Language.

```
1 // Model name: DomainEvents.technology
2 technology DomainEvents {
3   service aspects {
4     aspect Producer<singleval> for operations { string handlerName<mandatory>; }
5     aspect Consumer<singleval> for operations { string handlerName<mandatory>; }
6   }
7 }
```

LEMMA supports service-related and operation-related technology aspects. The latter are applicable to operation nodes expressed in LEMMA's Operation Modeling Language (Appendix B). For the Domain Event pattern, however, we rely on service-related technology aspects which allow augmentation of LEMMA domain and service model elements. The above technology model for the Domain Event pattern specifies the `service aspects` `Producer` and `Consumer`. Both can occur at most once (`singleval`) on modeled microservice `operations` (Table 1) and require the configuration of a `handlerName`. According to the pattern definition [55], this latter property identifies the program unit, e.g., the Java class, being responsible for handling domain event production or consumption.

Listing 3 illustrates the application of the Domain Event aspects in a LEMMA service model and mapping model, respectively.

Listing 3 Excerpts of (a) a service model in LEMMA's Service Modeling Language; and (b) a mapping model in LEMMA's Service Technology Mapping Modeling Language. Both models apply the Producer concept of the Domain Event pattern based on the technology model in Listing 2.

```
1  // Model name:
2  //  ChargingStationManagementCommandSide.services
3  import datatypes from "ChargingStationManagement.data"
4    as Domain
5  import technology from "DomainEvents.technology"
6    as DomainEvents
7  @technology(DomainEvents)
8  public functional microservice
9    org.example.ChargingStationManagementCommandSide {
10   interface CommandSide {
11     @DomainEvents::_aspects.Producer
12       ("ParkingAreaProducerService")
13     sendParkingAreaCreatedEvent(
14       async out event
15         : Domain::ParkingAreaCreatedEvent
16     );
17   }
18 }
```

(a)

```
1  // Model name:
2  //  ChargingStationManagementCommandSide.mapping
3  import microservices from
4    "ChargingStationManagementCommandSide.services"
5      as CommandSideServices
6  import technology from
7    "DomainEvents.technology"
8    as DomainEvents
9  @technology(DomainEvents)
10 CommandSideServices
11   ::org.example.ChargingStationManagementCommandSide {
12   operation CommandSide.sendParkingAreaCreatedEvent {
13     aspects {
14       DomainEvents::_aspects.Producer
15         ("ParkingAreaProducerService");
16     }
17   }
18 }
```

(b)

Listing 3a shows the excerpt of a LEMMA service model that specifies a Domain Event Producer by applying the eponymous aspect from the Domain Event technology model (Listing 2). Lines 3 to 4 import the domain model that defines the `ParkingAreaCreated-Event` domain event (Listing 1) and Lines 5 to 6 import the Domain Event technology model. Line 7 uses the built-in `@technology` annotation of LEMMA's Service Modeling Language (Appendix B) to assign the Domain Event technology model to the `Charg-ingStationManagementCommandSide` microservice whose definition starts in Lines 8 to 9. The service consists of the `CommandSide` interface (Line 10) that clusters the operation `sendParkingAreaCreatedEvent` (Lines 13 to 16). This operation is marked as a Domain Event Producer in Lines 11 to 12. More precisely, the assignment of the Domain Event technology model to the `ChargingStationManagementCommandSide` microservice enabled us to apply the `Producer` aspect to the operation, thereby semantically identifying it as a Domain Event Producer (Sect. 3.1).

Listing 3b also performs this semantic enrichment but on the basis of LEMMA's Service Technology Mapping Modeling Language (Appendix B). The primary difference to Listing 3a is the aspect application within the `aspects` section in Lines 13 to 16. The application of technology aspects in mapping models is an alternative to aspect application in service models.

Specifically, it externalizes technology aspect application which allows keeping service models technology-agnostic and thus reusable across different technology stacks, thereby facilitating technology migration on the model-level to deal with MSA's technology heterogeneity [36].

Model Validator Implementation

LEMMA's Technology Modeling Language already integrates certain constructs like `mandatory`, `singleval`, and the `for`-selector to constrain aspect application [51] (Listing 2). However, these constructs are not sufficient to express pattern-related constraints such as those in Table 1, e.g., because the constraints require traversal of other LEMMA models, the simultaneous application of aspects from other technology models, or certain aspect property values. We decided against the extension of the Technology Modeling Language with modeling support for such complex constraints to keep the language concise.

Instead, we employ LEMMA's Model Processing Framework (MPF; Appendix B) to accompany pattern-specific technology models with required conformance constraints. The result is a LEMMA model processor whose Model Validation phase performs constraint checking and violation reporting, and is extensible by code generation capabilities (Sect. 3.3). Listing 4 shows an excerpt of the MPF-based model validator for the Domain Event pattern written in Kotlin[5]. The complete source code can be found on Software Heritage [44].

Listing 4 Excerpt of the model validator for the Domain Event pattern in Kotlin.

```
1  $$@SourceModelValidator$$
2  internal class ServiceModelSourceValidator : AbstractXtextModelValidator() {
3    $$@Check$$
4    private fun checkProducer(operation: Operation) {
5      if (operation.hasServiceAspect("DomainEvents", "Producer") &&
6        !operation.hasResultParameters(CommunicationType.ASYNCHRONOUS))
7        error("The Producer aspect may only be applied to operations with a result " +
8          "parameter", ServicePackage.Literals.OPERATION__NAME)
9    }
10 }
```

A model validator based on the MPF must exhibit the `@SourceModelValidator` annotation [47] (Line 1) and extend the `AbstractXtextModelValidator` class [45] (Line 2). Furthermore, a validation method (Lines 4 to 9) must be preceded by the `@Check` annotation. From this annotation and the annotated method's signature, the MPF recognizes validation methods and when to invoke them. More specifically, the MPF will traverse the abstract syntax tree (AST) of a parsed LEMMA input model, identify the type of the current AST node, and, during model validation, invoke all validation methods whose signature matches this type. Consequently, it executes `checkProducer` for every modeled microservice `Operation`. In case the operation applies the `Producer` aspect from the `DomainEvents` technology model (cf. Line 5 and Listing 2) and does not have an asynchronous result parameter (Line 6), the validator yields an error using the `error` method from LEMMA's MPF (Lines 7 to 8). `checkProducer` thus implements Constraint C.1 of the Domain Event pattern (Table 1).

Model processors based on LEMMA's MPF are commandline tools that can readily be integrated into continuous integration pipelines [27]. However, the MPF also supports Live Validation for interactive model validation and error reporting. It relies on the Language Server Protocol (LSP) [11] to connect to the modeling IDE and display validation errors during model construction so that modelers need not invoke commandline validation separately from the IDE and trace errors messages to the erroneous model elements manually. Figure 2 shows the IDE manifestation of violating the check for Constraint C.1 in Listing 4.

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

Figure 2 Model validation error resulting from an unintended application of the `Producer` aspect for the Domain Event pattern and reported by LEMMA's MPF to the modeling IDE via the LSP.

3.3 Phase 3: Pattern-Conform LEMMA Code Generation

This phase consists solely of the Code Generator Implementation activity which has to consider possibly existing generators (Fig. 1). For example, the realization of the Domain Event pattern requires a message broker to transmit domain events [55]. As there exist several alternative broker technologies, e.g., RabbitMQ[6] or Kafka[7], a Domain Event code generator has to take into account the possible existence of generators for certain broker technologies. For Java-based microservices, LEMMA already bundles a Java Base Generator (JBG) that provides convenience mechanisms to map LEMMA model elements to Java code and a plugin infrastructure for MPF-based Java code generators.

In the following, we suppose that pattern-specific code generators produce Java code. Consequently, they can directly be integrated with LEMMA's JBG. Listing 5 shows an excerpt of the Domain Event code generator plugin for the JBG. Since the JBG is based on LEMMA's MPF, this generator is part of the same model processor as the pattern's model validator (Listing 4) and its complete source code is also available on Software Heritage [43].

Listing 5 Excerpt of the Domain Event code generator plugin for the JBG written in Kotlin.

```
1  $$@CodeGenerationHandler$$
2  internal class DataStructureHandler : GenletCodeGenerationHandlerI {
3    override fun handlesEObjectsOfInstance() = IntermediateDataStructure::class.java
4    override fun generatesNodesOfInstance() = ClassOrInterfaceDeclaration::class.java
5    override fun execute(structure: IntermediateDataStructure,
6      clazz: ClassOrInterfaceDeclaration) {
7      val eventGroup = clazz.getAspectProperty("EventGroup", "name") ?: return null
8      val groupInterface = EventGroups.addOrGetGroupInterface(eventGroup)
9      node.addImplementedType(groupInterface.nameAsString)
10     return node
11   }
12 }
```

The JBG structures the MPF's Code Generation phase (Appendix B) into code generation handlers each of which maps a particular LEMMA model element type to a Java AST node type [22]. A code generation handler is a class augmented with the `@CodeGeneration-Handler` annotation (Line 1) and implementing the `GenletCodeGenerationHandlerI` interface (Line 2) such that the handler must override the methods `handlesEObjectsOfInstance` and `generatesNodesOfInstance` (Lines 3 and 4). The methods respectively inform the JBG which LEMMA model element type the handler can process and to which Java AST node type the element type corresponds. The handler in Listing 5 maps data structures in LEMMA's Domain Data Modeling Language (Appendix B) to Java class declarations.

The `execute` method (Lines 5 to 11) clusters the handler's logic. It concerns domain event grouping, and, following the pattern's definition [55], produces a Java interface for each event group which is then assigned as an implemented type to the domain event Java class.

[6] `https://www.rabbitmq.com`
[7] `https://kafka.apache.org`

4 Validation

We validate our method (Sect. 3) with CQRS, i.e., one of the most complex patterns in Sect. 2 that (i) spans several LEMMA viewpoints (Appendix B); (ii) requires more than one model for the same viewpoint and thus ad hoc model parsing; (iii) combines technology-agnostic and technology-specific validation; and (iv) impacts service configuration and source code.

We present the research questions (RQs; Sect. 4.1), case study (Sect. 4.2), and results (Sect. 4.3) of our method's validation, and discuss the latter (Sect. 4.4).

4.1 Research Questions

Our validation focuses on the following RQs:

RQ 1 To what extent do LEMMA and our method for its extension with pattern support cover patterns whose complexity exceeds that of Domain Event significantly?

RQ 2 How much effort is required to provide such complex extensions?

4.2 Case Study

We validate our method with a case study microservice architecture called Park and Charge Platform (PACP). The PACP originates from a research project that aims to enable the offering of private charging stations for electric vehicles for use by other owners of electric vehicles. We described the PACP's architecture in more detail in a previous publication [53]. Almost all PACP microservices apply CQRS. Figure 3 provides an overview of the pattern.

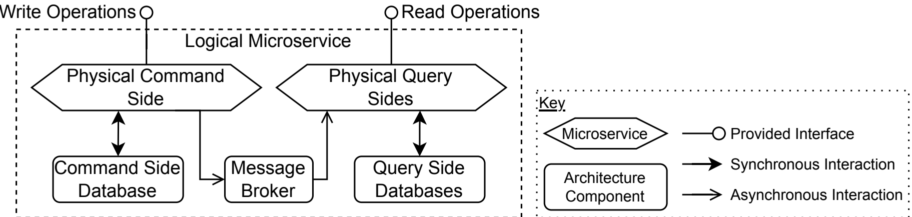


Figure 3 Overview of the CQRS pattern.

As mentioned in Sect. 2, a CQRS application consists of a logical microservice that provides clients with write and read operations. Depending on an operation's kind, its actual implementation resides in a physical command side microservice (write) or one of potentially several query side microservices (read). Command side microservices allow the alteration of data in command side databases [55] and asynchronously inform query side microservices about alterations via message brokers. Query sides then update their databases accordingly.

The benefits of CQRS for the PACP are twofold. First, we can scale read operations independently of write operations and for the PACP the former are much more frequent. Second, the pattern allows storage optimization for read operations so that we can preprocess sensor data from charging stations for time series processing as well as for relational queries.

In the following, we focus on the PACP's Charging Station Management Microservice. It is responsible for managing charging station information like location, charging and plug type, and also receives data from charging stations. While we consider only a single PACP microservice, our results are directly transferable to the CQRS-conform modeling and code generation of all other PACP microservices.

4.3 Results

We structure the presentation of the validation results by the phases of our method (Sect. 3).

Phase 1: Pattern Analysis

Figure 3 is a direct outcome of the Concept Identification activity. Structured by LEMMA's viewpoints (Appendix B), a CQRS application consists of the following concepts:

- **Domain Viewpoint:** Domain concepts to be stored in databases and used to convey asynchronous messages between physical microservices.
- **Service Viewpoint:** Logical CQRS microservice consisting of exactly one command side and at least one query side microservice. Physical microservices provide synchronous operations to consumers and interact with each other asynchronously.
- **Operation Viewpoint:** Infrastructure nodes like databases and a message broker.

In the following, we focus on the Domain and Service Viewpoint because they have the most impact on the upcoming activities.

During the Concept Reconciliation activity (Fig. 1), we not only discovered the above concepts but also the close relationship between the DOMAIN EVENT and CQRS patterns. Consequently, we decided to rely on domain events to define the structures for the asynchronous messages sent from command side to query side microservices.

Table 2 lists the constraints resulting from the Constraint Formulation activity.

Table 2 Conformance constraints for the CQRS pattern.

#	Constraint	Severity
C.3	Logical microservice consists of command and at least one query side microservice.	Error
C.4	Command side microservice should be able to send domain events.	Warning
C.5	Query side microservice should be able to receive domain events.	Warning
C.6	Incoming domain event parameters of query side operations should be type-compatible with outgoing domain event parameters of command side operations.	Warning

Violations of Constraints C.4 to C.6 yield warnings to permit iterative CQRS modeling.

Phase 2: Pattern-Conform LEMMA Modeling

From the concerned LEMMA viewpoints, the Model Type Identification activity identified the domain, service, and operation model types (Appendix B) to be affected by CQRS.

Technology Model Construction resulted in the LEMMA technology model in Listing 6.

Listing 6 CQRS technology model in LEMMA's Technology Modeling Language.

```
1 // Model name: Cqrs.technology
2 technology CQRS {
3   service aspects {
4     aspect CommandSide for microservices { string logicalService; }
5     aspect QuerySide for microservices { string logicalService; }
6   }
7 }
```

The `CommandSide` and `QuerySide` aspects allow determination of command side and query side microservices. Two microservices that apply these aspects belong to the same logical CQRS microservice when the values for the `logicalService` property are equal.

In the Model Validator Implementation activity, we developed a LEMMA model validator for the above technology model. Its Kotlin-based implementation consists of 180 lines of code (LOC), excluding empty lines, and is available on Software Heritage [42]. The validator implements Constraints C.4 and C.5 (Table 2).

To realize Constraint C.3, we decided to rely on a built-in construct of LEMMA's Service Modeling Language (Appendix B), i.e., the `required microservices` directive. It allows modeling of relationships between microservices in the same or distinct service models – in the latter case based on LEMMA's import mechanism and inter-model references. Hence, our CQRS validator checks the constraints in Table 2 only when a query side requires a command side microservice of the same logical microservice as identified by the `logicalService` property of the corresponding aspect applications (Listing 6). The specification of the dependency of a query side on a command side microservice is in line with CQRS because the former relies on events received by the latter to update its database accordingly (Fig. 3).

Concerning Constraint C.6, we decided against its inclusion in the CQRS validator. Instead, we extended an implementation for the type-checking of sending and receiving microservice operations in an existing JBG plugin (Sect. 3.3) for the Kafka message broker.

Phase 3: Pattern-Conform LEMMA Code Generation

LEMMA bundles a set of JBG plugins for popular MSA technologies like Kafka and Spring[8]. Among others, the Kafka plugin [40] supports the extension of Java codebases produced from LEMMA models by the JBG, e.g., with configuration code for the connection to a Kafka broker and with methods for sending events via this broker. For the extension of LEMMA with CQRS, we were able to reuse the plugin for the most part. As described above, we only had to extend its validator with a check for Constraint C.6 (Table 2). This extension concerned 141 LOC between Lines 185 and 343 of the validator's Kotlin implementation [41].

4.4 Discussion

We discuss the validation of our method w.r.t. the framed RQs (Sect. 4.1).

RQ 1 Method Applicability for Complex Architectural Patterns

Based on the successful application of the method on the CQRS pattern (Sect. 4.3) we conclude that it is applicable not only to comparatively simple patterns like Domain Event (Sect. 3) but also to patterns that involve more concepts, validations, and code to be generated. However, an in-depth evaluation of the method's applicability would require its usage on ideally all patterns from Sect. 2. In order to anticipate the structure of such an evaluation, we assessed the complexity of extending LEMMA with pattern support using our method for all of these patterns, except for Domain Event and CQRS. Table 3 shows an excerpt of this assessment for one pattern from each category in Sect. 2, besides Migration. That is, because Migration is mostly out of LEMMA's scope. We refer to Appendix C for the complete list of assessed pattern extension complexity.

API Gateway is the example of a pattern that requires comparatively low effort to be integrated with LEMMA. That is because it concerns only the Operation viewpoint (Appendix B) and requires the modeling of one infrastructure node that needs to be augmented with the technology for an API gateway. The amount of generated code is probably also

[8] `https://www.spring.io`

Table 3 Assessed complexity of extending LEMMA with support for selected architectural MSA patterns (see Appendix C for the complete list).

Pattern	LEMMA Viewpoints			Quantities[a)]			Extension Complexity[b)]
	Dom.	Serv.	Op.	Concepts	Constraints	LOC[c)]	
API Gateway			✓	○	○	○	○
Backend for Frontend	✓	✓	✓	○	○	●	◐
API Key	✓	✓	✓	○	◐	●	●
Distributed Tracing		✓	✓	●	●	●	●
Container			✓	○	○	●	N/A

Symbol key: ○ = Low; ◐ = Middle; ● = High.
a) Assessed relative across all patterns.
b) Assessed based on quantities, and experiences with DOMAIN EVENT (Sect. 3) and CQRS (Sect. 4).
c) Assessed quantity of generated code.

low as there exist readily usable implementations of the pattern, e.g., Zuul[9]. Similarly, the assumed integration effort for the BACKED FOR FRONTEND pattern is rather low. In fact, LEMMA already provides all required concepts for modeling this pattern and, as for CQRS, only a technology model to link a backend with a frontend service will be needed. Nonetheless, the amount of generated code will likely be high since backend services are full-fledged microservices. The API KEY pattern, on the other hand, calls for additional constraints that allow checking whether the additional security infrastructure was configured correctly. Moreover, we expect the necessity to generate this infrastructure to further increase the effort for API KEY integration. The integration of the DISTRIBUTED TRACING pattern will be nearly infeasible as it requires concepts and corresponding constraints for service interactions. However, LEMMA does currently not exhibit such concepts and the technology aspect mechanism is likely not sophisticated enough to fully retrofit them. Concerning the CONTAINER pattern, no LEMMA integration is needed as it is already fully supported by LEMMA's Operation Modeling Language (Appendix B).

RQ 2 Method Effort for Complex Architectural Patterns

Given our experience with the conceptual and implementation subtleties of LEMMA and the CQRS pattern, the execution of the method was straightforward. At least for the Pattern Analysis phase (Sect. 3), we expect the method to not impose significant overhead, even for stakeholders that are not familiar with LEMMA. That is because the phase does not require LEMMA knowledge and a basic knowledge of the targeted pattern can be assumed as otherwise the objective decision to extend LEMMA with support for it would not have been possible at all. For the first two activities of the Pattern-Conform LEMMA Modeling phase, basic LEMMA knowledge is necessary to identify the model types affected by pattern extension and construct the corresponding technology model. However, the amount of LEMMA knowledge increases significantly for subsequent model validator and code generator

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

development. Moreover, both these activities require additional implementation effort. LEMMA's extension with the CQRS pattern comprises 327 LOC including the technology model, validator, and extension of the Kafka JBG plugin (Sect. 4.2). For complex patterns like Gateway Offloading or Microservice Chassis (Appendix C) the LOC count is likely to increase, especially when no JBG plugin or model processor for extension exists. We consider the objective quantification of the effort required by our method of high interest for future research works.

5 Related Work

We present work related to MDE-for-MSA and architectural pattern conformance assurance.

MDE-for-MSA

There exists a plethora of MDE approaches with support for the MSA viewpoints Domain [13, 63, 29], Service [23, 2, 61, 29, 25], and Operation [18, 2, 61, 25] (Appendix B). By contrast to LEMMA and the presented method (Sect. 3), the majority of these approaches does not allow pattern integration on the language-level. Only DCSL [13] can be considered to have basic support for language-level extensibility leveraging meta-attributes. However, it only covers domain modeling and only a subset of architectural MSA patterns concerns the Domain viewpoint (Appendix C). Instead, most of the patterns are rooted in the Service and Operation viewpoints. As opposed to LEMMA, MDE-for-MSA approaches for these viewpoints integrate pattern-specific concepts like `Circuit Breaker` [2], `APIGatewayService` [61], or `serviceDiscoveryType` [25] into provided modeling languages. Hence, these approaches aggravate learnability by extensive language syntaxes and prevent retrofitting, thus requiring new releases each time the modeling of a new pattern shall be supported.

Architectural Pattern Conformance Assurance

Conformance checking between the intended design of a software and its actual implementation is an important activity in software architecting [5] that may also reveal deviations from pattern specifications. Kim and Shen [30] leverage a divide-and-conquer strategy to assess the syntactic conformance of UML class diagrams [39] with design patterns that are specified via a UML extension for role-based metamodeling. Roles in pattern models prescribe contributions to pattern solutions and can be played by more than one element in assessed class diagrams. Consequently, the approach is able to also capture pattern variations. Chihada et al. [9] present an approach to pattern conformance checking based on Support Vector Machines. To this end, classifiers for design patterns are trained based on the peculiarities of object-oriented metrics in pattern specifications. In the next step, the classifiers are applied to source code that has been partitioned into smaller chunks of possible design pattern manifestations. From the confidence values returned by classifiers, Chihada et al. assess the likeliness for pattern occurrence. This approach is also able to capture pattern variations. However, other than Kim and Shen, Chihada et al. operate on source code. Díaz-Pace et al. [14] suggest a heuristic approach to detect source code deviations from behavioral architecture scenarios expressed as use-case maps (UCMs) which are graph-based descriptions of expected system executions. Deviations between UCM specifications and evolved source code are then detected by exercising the architecture implementation against test cases derived from UCMs. Similar to Kim and Shen, Díaz-Pace et al. identify conformance by means of an abstract view on component responsibilities. By contrast to the aforementioned works, our method

(Sect. 3) aims to ensure pattern conformance by model validation and code generation. While this approach allows pattern expression in the same constructive source model and on a level of abstraction that facilitates pattern recognition, it also expects the production of code that is actually pattern-conform, thereby bundling pattern conformance checking and pattern-conform code generation in model processors. Furthermore, we focus on architectural patterns instead of design patterns and do not explicitly consider pattern variations. They may however be codified as aspect properties in pattern technology models so that code generators produce varied pattern instances w.r.t. property values. Similar to Chihada et al., we also consider pattern analysis a crucial step prior to pattern handling.

There also exist works concerning the assessment and restoration of architectural pattern conformance in MSA engineering [38, 37]. These works are of particular interest to us because they define metrics and refactorings on the model-level to determine and resolve pattern deviations. We regard the integration of these approaches with our method as a sensible future work as it would permit (i) testing the actual conformance of generated pattern code, thereby enabling model processor developers to evaluate processor correctness; and (ii) resolution of pattern deviations from reconstructed architecture models [52].

Similar to us, Falkenthal et al. [17] focus on linking pattern specifications with solution implementations which may be source code that reifies a pattern realization. Next to pattern variation, the work by Falkenthal et al. raises an important concern regarding the modularization and composition of pattern solutions. Our method recognizes this concern in Phase 3 (Fig. 1), where existing code generators are examined prior to the implementation of a novel pattern-specific code generator in order to identify generator dependencies and reusability. In its current form, this examination has to be conducted manually, and thus requires deep knowledge about existing generators and their implementations. It would therefore be beneficial to reason about means to describe code generator composability on a more abstract level, e.g., by extending LEMMA's Technology Modeling Language with capabilities to express composition relationships between pattern-specific technology aspects.

6 Conclusion and Future Work

In this paper, we presented an approach that combines two orthogonal means for complexity reduction in Microservice Architecture (MSA) engineering, i.e., architectural MSA patterns (Sect. 2) and language-based approaches to MSA. More precisely, we introduced a method (Sect. 3) for the systematic retrofitting of LEMMA (Language Ecosystem for Modeling Microservices) with support for modeling and implementing architectural MSA patterns. The presented method relies on the (i) specification of aspects to reify pattern applications in MSA models; (ii) validation of pattern applications for correctness; and (iii) code generation from MSA models with correct pattern applications. We validated the feasibility of our method for two popular architectural MSA patterns, i.e., Domain Event and Command Query Responsibility Segregation (CQRS), and provide a complexity assessment for the integration of 28 other patterns with LEMMA (Sect. 4). Our approach aims at enabling correct-by-construction microservices by abstracting from the complexity of pattern implementations, yet still enabling their automated production with techniques from Model-Driven Engineering (MDE). In particular, our results show that LEMMA's expressivity is versatile enough to even allow the model-based expression of comparatively complex architectural MSA patterns such as CQRS and that the provided model processing facilities support pattern-specific LEMMA extensions such that pattern conformance checking and pattern-conform code generation can be modularized into reusable model processors.

While we are confident that from the 28 remaining architectural MSA patterns, 19 exhibit a low to middle complexity concerning their integration with LEMMA, an objective quantification is still necessary. We therefore plan to extend LEMMA with support for all of these patterns and identify even further applicable patterns, e.g., from Cloud Computing [62]. We are also interested in applying our approach with other works targeting the model-based assessment and restoration of architectural pattern conformance in microservice implementations. As a result, it would become possible to detect pattern applications and deviations from source code, and exploit the presented approach for the correct resolution of such deviations. We are also interested in extending LEMMA with means to anticipate the composability of pattern-specific code generators, e.g., by expressing composition relationships between pattern-specific technology aspects on the model-level.

References

1. Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, 1977.
2. Nuha Alshuqayran, Nour Ali, and Roger Evans. Towards micro service architecture recovery: An empirical study. In *2018 Int. Conf. on Softw. Architecture (ICSA)*, pages 47–56. IEEE, 2018. doi:10.1109/ICSA.2018.00014.
3. David Ameller, Xavier Burgués, Oriol Collell, Dolors Costal, Xavier Franch, and Mike P. Papazoglou. Development of service-oriented architectures using model-driven development: A mapping study. *Information and Software Technology*, 62:42–66, 2015. Elsevier. doi:10.1016/J.INFSOF.2015.02.006.
4. Paul Baker, Shiou Loh, and Frank Weil. Model-Driven Engineering in a large industrial context—Motorola case study. In *Model Driven Engineering Languages and Systems*, pages 476–491, Berlin, Heidelberg, 2005. Springer. doi:10.1007/11557432_36.
5. Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, third edition, 2013.
6. Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. Pearson, 2015. URL: http://bookshop.pearson.de/devops.html?productid=208463.
7. Justus Bogner, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann. Microservices in industry: Insights into technologies, characteristics, and software quality. In *2019 Int. Conf. on Softw. Architecture Companion (ICSA-C)*, pages 187–195. IEEE, 2019. doi:10.1109/ICSA-C.2019.00041.
8. Frank Buschmann, Kelvin Henney, and Douglas Schimdt. *Pattern-Oriented Software Architecture*, volume 5. Wiley, 2007. URL: https://www.worldcat.org/oclc/314792015.
9. Abdullah Chihada, Saeed Jalili, Seyed Mohammad Hossein Hasheminejad, and Mohammad Hossein Zangooei. Source code and design conformance, design pattern detection from source code by classification approach. *Applied Soft Computing*, 26:357–367, 2015. doi:10.1016/J.ASOC.2014.10.027.
10. Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, Bernhard Rumpe, Jim Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. CRC, 2017.
11. Microsoft Corporation. Language Server Protocol specification - 3.17, 2022. URL: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification.
12. Nicola Dragoni, Ivan Lanese, Stephan Thordal Larsen, Manuel Mazzara, Ruslan Mustafin, and Larisa Safina. Microservices: How to make your application scale. In *Perspectives of System Informatics*, pages 95–104. Springer, 2018. doi:10.1007/978-3-319-74313-4_8.
13. Duc Minh Le, Duc-Hanh Dang, and Viet-Ha Nguyen. Domain-driven design using meta-attributes: A DSL-based approach. In *2016 Eighth International Conference on Knowledge and Systems Engineering (KSE)*, pages 67–72. IEEE, 2016. doi:10.1109/KSE.2016.7758031.

14 J.A. Díaz-Pace, Álvaro Soria, Guillermo Rodríguez, and Marcelo R. Campo. Assisting conformance checks between architectural scenarios and implementation. *Information and Software Technology*, 54(5):448–466, 2012. doi:10.1016/J.INFSOF.2011.12.005.

15 Eric Evans. *Domain-Driven Design*. Addison-Wesley, 2004.

16 Eric Evans. *Domain-Driven Design Reference*. Dog Ear Publishing, 2015.

17 Michael Falkenthal, Johanna Barzen, Uwe Breitenbücher, Christoph Fehling, and Frank Leymann. From pattern languages to solution implementations. In *Proceedings of the 6th International Conference on Pervasive Patterns and Applications (PATTERNS 2014)*, pages 12–21. Xpert Publishing Services (XPS), 2014.

18 Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin, and Arnor Solberg. Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 887–894. IEEE, 2013. doi:10.1109/CLOUD.2013.133.

19 Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering (FOSE)*, pages 37–54. IEEE, 2007. doi:10.1109/FOSE.2007.14.

20 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

21 Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, and Florian Rademacher. Model-driven generation of microservice interfaces: From LEMMA domain models to Jolie APIs. In *Coordination Models and Languages*, pages 223–240. Springer, 2022. doi:10.1007/978-3-031-08143-9_13.

22 James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. The Java language specification: Java SE 17 edition. JSR-392, Oracle Inc., 2021.

23 Giona Granchelli, Mario Cardarelli, Paolo Di Francesco, Ivano Malavolta, Ludovico Iovino, and Amleto Di Salle. Towards recovering the software architecture of microservice-based systems. In *2017 Int. Conf. on Softw. Arch. Workshops (ICSAW)*, pages 46–53. IEEE, 2017. doi:10.1109/ICSAW.2017.48.

24 ISO/IEC/IEEE. Systems and software engineering — Architecture description, 2011.

25 JHipster. Jhipster domain language (jdl), 2022-14-02. URL: https://www.jhipster.tech/jdl.

26 Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988. SIGS Publications.

27 Robbert Jongeling, Jan Carlson, and Antonio Cicchetti. Impediments to introducing continuous integration for model-based development in industry. In *45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 434–441. IEEE, 2019. doi:10.1109/SEAA.2019.00071.

28 Stefan Kapferer and Olaf Zimmermann. Domain-driven service design. In *Service-Oriented Computing*, pages 189–208. Springer, 2020. doi:10.1007/978-3-030-64846-6_11.

29 Stefan Kapferer and Olaf Zimmermann. 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*, pages 299–306. INSTICC, SciTePress, 2020. doi:10.5220/0008910502990306.

30 Dae-Kyoo Kim and Wuwei Shen. Evaluating pattern conformance of uml models: a divide-and-conquer approach and case studies. *Software Quality Journal*, 16(3):329–359, sep 2008. doi:10.1007/S11219-008-9048-5.

31 Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. Montithings: Model-driven development and deployment of reliable iot applications. *Journal of Systems and Software*, 183:111087, 2022. doi:10.1016/J.JSS.2021.111087.

32 Kevin Lano and Shekoufeh Kolahdouz-Rahimi. Model-transformation design patterns. *IEEE Transactions on Software Engineering*, 40(12):1224–1259, 2014. IEEE. doi:10.1109/TSE.2014.2354344.

33 Parastoo Mohagheghi and Vegard Dehlen. Where is the proof? - A review of experiences from applying MDE in industry. In *Model Driven Architecture – Foundations and Applications*, pages 432–443. Springer, 2008. doi:10.1007/978-3-540-69100-6_31.

34 Mustafa Abshir Mohamed, Moharram Challenger, and Geylani Kardas. Applications of model-driven engineering in cyber-physical systems: A systematic mapping study. *Journal of Computer Languages*, 59:1–54, 2020. doi:10.1016/J.COLA.2020.100972.

35 Gastón Márquez and Hernán Astudillo. Actual use of architectural patterns in microservices-based open source projects. In *25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 31–40. IEEE, 2018. doi:10.1109/APSEC.2018.00017.

36 Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly, 2015. URL: https://www.worldcat.org/oclc/904463848.

37 Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, and Sebastian Geiger. Evaluating and improving microservice architecture conformance to architectural design decisions. In Hakim Hacid, Odej Kao, Massimo Mecella, Naouel Moha, and Hye-young Paik, editors, *Service-Oriented Computing*, pages 188–203, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-91431-8_12.

38 Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, Sebastian Meixner, and Sebastian Geiger. Metrics for assessing architecture conformance to microservice architecture patterns and practices. In Eleanna Kafeza, Boualem Benatallah, Fabio Martinelli, Hakim Hacid, Athman Bouguettaya, and Hamid Motahari, editors, *Service-Oriented Computing*, pages 580–596, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-65310-1_42.

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

40 Florian Rademacher. Kafka plugin for LEMMA's Java Base Generator on Software Heritage. URL: https://archive.softwareheritage.org/swh:1:dir:98fdbe5eb42ef33cf9cc7895ef47f6ca8f3791b0;origin=https://github.com/SeelabFhdo/lemma;visit=swh:1:snp:a668854675d50d3f3cad56eb5e3961b683d0f70a;anchor=swh:1:rev:2e9ccc882352116b253a7700b5ecf2c9316a5829;path=/code%252520generators/de.fhdo.lemma.model_processing.code_generation.springcloud.kafka/.

41 Florian Rademacher. Service model validator in the kafka plugin for LEMMA's Java Base Generator on Software Heritage. URL: https://archive.softwareheritage.org/swh:1:cnt:3b1e8bb4a5da879b05b06812c855e50572bbdc96;origin=https://github.com/SeelabFhdo/lemma;visit=swh:1:snp:a668854675d50d3f3cad56eb5e3961b683d0f70a;anchor=swh:1:rev:2e9ccc882352116b253a7700b5ecf2c9316a5829;path=/code%20generators/de.fhdo.lemma.model_processing.code_generation.springcloud.kafka/src/main/kotlin/de/fhdo/lemma/model_processing/code_generation/springcloud/kafka/validators/ServiceModelSourceValidator.kt.

42 Florian Rademacher. CQRS model validator on Software Heritage. URL: https://archive.softwareheritage.org/swh:1:cnt:9558df1409cbac539118ff3ed8eab013e8d84a44;origin=https://github.com/SeelabFhdo/lemma;visit=swh:1:snp:a668854675d50d3f3cad56eb5e3961b683d0f70a;anchor=swh:1:rev:2e9ccc882352116b253a7700b5ecf2c9316a5829;path=/code%20generators/de.fhdo.lemma.model_processing.code_generation.springcloud.cqrs/src/main/kotlin/de/fhdo/lemma/model_processing/code_generation/springcloud/cqrs/validators/ServiceModelSourceValidator.kt.

43 Florian Rademacher. DOMAIN EVENT code generation handler on Software Heritage. URL: https://archive.softwareheritage.org/swh:1:cnt:dab6b250ef32a4da914ffdcae0b8f7f5c79e09a1;origin=https://github.com/SeelabFhdo/lemma;visit=swh:1:snp:a668854675d50d3f3cad56eb5e3961b683d0f70a;anchor=swh:1:rev:2e9ccc882352116b253a7700b5ecf2c9316a5829;path=/code%20generators/de.fhdo.lemma.model_processing.code_generation.springcloud.domain_events/src/

main/kotlin/de/fhdo/lemma/model_processing/code_generation/springcloud/domain_events/handlers/DataStructureHandler.kt.

44 Florian Rademacher. Domain Event model validator on Software Heritage. URL: https://archive.softwareheritage.org/swh:1:cnt:46968c76c89ffe9cb23d480403eafcc837a4cd23;origin=https://github.com/SeelabFhdo/lemma;visit=swh:1:snp:a668854675d50d3f3cad56eb5e3961b683d0f70a;anchor=swh:1:rev:2e9ccc882352116b253a7700b5ecf2c9316a5829;path=/code%20generators/de.fhdo.lemma.model_processing.code_generation.springcloud.domain_events/src/main/kotlin/de/fhdo/lemma/model_processing/code_generation/springcloud/domain_events/validators/ServiceModelSourceValidator.kt.

45 Florian Rademacher. AbstractXtextModelValidator class of lemma's model processing framework on Software Heritage. URL: https://archive.softwareheritage.org/swh:1:cnt:cbc0b6283dc88cee0c747f0824cf697d05db8506;origin=https://github.com/SeelabFhdo/lemma;visit=swh:1:snp:a668854675d50d3f3cad56eb5e3961b683d0f70a;anchor=swh:1:rev:2e9ccc882352116b253a7700b5ecf2c9316a5829;path=/de.fhdo.lemma.model_processing/src/main/kotlin/de/fhdo/lemma/model_processing/phases/validation/AbstractXtextModelValidator.kt.

46 Florian Rademacher. ChargingStationManagement.data lemma domain model on Software Heritage. URL: https://archive.softwareheritage.org/swh:1:cnt:1578bbef5fb1b6463db33582c06d930236ed8653;origin=https://github.com/SeelabFhdo/lemma;visit=swh:1:snp:a668854675d50d3f3cad56eb5e3961b683d0f70a;anchor=swh:1:rev:2e9ccc882352116b253a7700b5ecf2c9316a5829;path=/examples/charging-station-management/models/domain/ChargingStationManagement.data.

47 Florian Rademacher. SourceModelValidator annotation of lemma's model processing framework on Software Heritage. URL: https://archive.softwareheritage.org/swh:1:cnt:2207b7f3f5ddbee2bda1ec9ae1968b9fed03947f;origin=https://github.com/SeelabFhdo/lemma;visit=swh:1:snp:a668854675d50d3f3cad56eb5e3961b683d0f70a;anchor=swh:1:rev:2e9ccc882352116b253a7700b5ecf2c9316a5829;path=/de.fhdo.lemma.model_processing/src/main/kotlin/de/fhdo/lemma/model_processing/annotations/SourceModelValidator.kt.

48 Florian Rademacher. A non-intrusive approach to extend microservice modeling languages with architecture pattern support. In *Third Int. Conf. on Microservices (Microservices 2020)*, 2020. URL: https://www.conf-micro.services/2020/papers/paper_3.pdf.

49 Florian Rademacher. *A Language Ecosystem for Modeling Microservice Architecture*. PhD thesis, University of Kassel, 2022. URL: https://kobra.uni-kassel.de/handle/123456789/14176.

50 Florian Rademacher, Martin Peters, and Sabine Sachweh. Design of a domain-specific language based on a technology-independent web service framework. In *Software Architecture*, pages 357–371. Springer, 2015. doi:10.1007/978-3-319-23727-5_29.

51 Florian Rademacher, Sabine Sachweh, and Albert Zündorf. Aspect-oriented modeling of technology heterogeneity in Microservice Architecture. In *2019 Int. Conf. on Softw. Architecture (ICSA)*, pages 21–30. IEEE, 2019. doi:10.1109/ICSA.2019.00011.

52 Florian Rademacher, Sabine Sachweh, and Albert Zündorf. A modeling method for systematic architecture reconstruction of microservice-based software systems. In *Enterprise, Business-Process and Information Systems Modeling*, pages 311–326. Springer, 2020. doi:10.1007/978-3-030-49418-6_21.

53 Florian Rademacher, Jonas Sorgalla, Philip Wizenty, and Simon Trebbau. Towards holistic modeling of microservice architectures using LEMMA. In *Companion Proc. of the 15th European Conf. on Software Architecture 2021*, pages 1–10. CEUR-WS, 2021. URL: http://ceur-ws.org/Vol-2978/mde4sa-paper2.pdf.

54 Florian Rademacher, Jonas Sorgalla, Philip Wizenty, and Simon Trebbau. Towards an extensible approach for generative microservice development and deployment using lemma. In

Patrizia Scandurra, Matthias Galster, Raffaela Mirandola, and Danny Weyns, editors, *Software Architecture*, pages 257–280. Springer, 2022. doi:10.1007/978-3-031-15116-3_12.

55 Chris Richardson. *Microservices Patterns*. Manning Publications, 2019.

56 Davide Di Ruscio, Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Alfonso Pierantonio. Developing next generation ADLs through MDE techniques. In *32nd Int. Conf. on Software Engineering (ICSE)*, volume 1, pages 85–94. IEEE, 2010. doi:10.1145/1806799.1806816.

57 Gerald Schermann, Jürgen Cito, and Philipp Leitner. All the services large and micro: Revisiting industrial practice in services computing. In *Service-Oriented Computing – ICSOC 2015 Workshops*, pages 36–47. Springer, 2016. doi:10.1007/978-3-662-50539-7_4.

58 Stefan Sobernig and Uwe Zdun. Inversion-of-Control layer. In *Proc. of the 15th European Conf. on Pattern Languages of Programs*, pages 1–22. ACM, 2010. doi:10.1145/2328909.2328935.

59 Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232, 2018. Elsevier. doi:10.1016/J.JSS.2018.09.082.

60 Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proc. of the 2nd Europ. Conf. on Computer Systems 2007*, pages 275–287. ACM, 2007. doi:10.1145/1272996.1273025.

61 Jonas Sorgalla, Philip Wizenty, Florian Rademacher, Sabine Sachweh, and Albert Zündorf. AjiL: Enabling model-driven microservice development. In *Proc. of the 12th Europ. Conf. on Softw. Architecture: Companion Proceedings*, pages 1:1–1:4. ACM, 2018. doi:10.1145/3241403.3241406.

62 Tiago Boldt Sousa, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. A survey on the adoption of patterns for engineering software for the cloud. *IEEE Transactions on Software Engineering*, 48(6):2128–2140, 2022. doi:10.1109/TSE.2021.3052177.

63 Branko Terzić, Vladimir Dimitrieski, Slavica Kordić, Gordana Milosavljević, and Ivan Luković. Development and evaluation of MicroBuilder: a model-driven tool for the specification of REST microservice software architectures. *Enterprise Information Systems*, 12:1034–1057, 2018. doi:10.1080/17517575.2018.1460766.

64 Guilherme Vale, Filipe Figueiredo Correia, Eduardo Martins Guerra, Thatiane de Oliveira Rosa, Jonas Fritzsch, and Justus Bogner. Designing microservice systems using patterns: An empirical study on quality trade-offs. In *19th Int. Conf. on Softw. Architecture (ICSA)*, pages 69–79, 2022. doi:10.1109/ICSA53651.2022.00015.

65 Matias Ezequiel Vara Larsen, Julien DeAntoni, Benoit Combemale, and Frédéric Mallet. A behavioral coordination operator language (bcool). In *18th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS)*, pages 186–195, 2015. doi:10.1109/MODELS.2015.7338249.

66 Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in Model-Driven Engineering. *IEEE Software*, 31(3):79–85, 2014. IEEE. doi:10.1109/MS.2013.65.

67 Andreas Wortmann, Olivier Barais, Benoit Combemale, and Manuel Wimmer. Modeling languages in industry 4.0: an extended systematic mapping study. *Software and Systems Modeling*, 19(1):67–94, jan 2020. doi:10.1007/S10270-019-00757-6.

68 Cheng Zhang and David Budgen. What do we know about the effectiveness of software design patterns? *IEEE Transactions on Software Engineering*, 38(5):1213–1231, 2012. doi:10.1109/TSE.2011.79.

69 Cheng Zhang and David Budgen. A survey of experienced user perceptions about software design patterns. *Information and Software Technology*, 55(5):822–835, 2013. doi:10.1016/J.INFSOF.2012.11.003.

70 Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. *Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges*. Addison-Wesley, 2022.

Appendix

A Model-Driven Engineering

MDE is a paradigm that promotes the use of *models* in the software engineering process [10]. In the sense of MDE, a model is a software artifact that describes selected aspects of a software system in an abstracted fashion and can replace more concrete artifacts for certain purposes. For instance, a model may be a partial representative for source code focusing on capturing component interfaces [50] or coordination [65]. A major goal of MDE is to increase models' usage beyond documentation and turn them into first-class citizens in software engineering, e.g., by deriving executable code from them or facilitate quality assessment [66].

MDE formalizes construction specifications for models as *modeling languages* [10]. A modeling language consists of (i) an abstract syntax defining available modeling concepts; (ii) one or more concrete syntaxes whose constructs allow concept instantiation by users; and (iii) semantics that assign meaning to concepts. MDE suggests the realization of *model processors* to elevate models from documentation to engineering artifacts. Examples of model processors comprise code generators, static analyzers, and interpreters [10].

Given its focus on abstraction, MDE is particularly applicable in the design, development, and operation of complex software systems [19], and their architectures [56]. Its adoption has benefited software engineering in heterogeneous domains such as cyber-physical systems [34], Industry 4.0 [67], Internet of Things [31], and Service-Oriented Architecture (SOA) [3].

B Model-Driven Microservice Engineering with LEMMA

The successful adoption of MDE for SOA (Appendix A) stimulated research on its application to MSA [23, 2, 63, 61, 28, 25]. LEMMA (Language Ecosystem for Modeling Microservice Architecture) [49] is an approach to *MDE-for-MSA* [21] that copes with MSA's complexity by concern-driven decomposition. To this end, it provides integrated modeling languages for the modeling of a microservice architecture from different *viewpoints* [24]. A viewpoint addresses selected stakeholder concerns, and prescribes notations and processing instructions for architecture models reifying such concerns. Figure 4a shows LEMMA's viewpoints, their stakeholders, modeling languages and model types, and Table 4 describes them.

LEMMA provides an import mechanism to let modeling languages prescribe reference relationships between modeling concepts. This mechanism supports model integration across viewpoints to increase models' information content. Figure 4b depicts the import relationships between LEMMA's model types. The following import relationships can be established:

- **Domain Model:** Domain models can import domain concepts from other domain models to use these external concepts as types for local concepts.
- **Service Model:** Service models can import microservices from other service models to capture service dependencies. Furthermore, service models can import domain concepts from domain models to use them as types of operation parameters. To configure a microservice to exploit a certain technology, service models can be augmented with information captured by imported technology models, e.g., protocols or aspects (Table 4).
- **Mapping Model:** Mapping models import microservices and, transitively, domain concepts from service models as well as technology information from technology models. They can then augment microservices and domain concepts with technology information.
- **Technology Model:** Technology models import other technology models to specify conversion directions between technology-specific types.

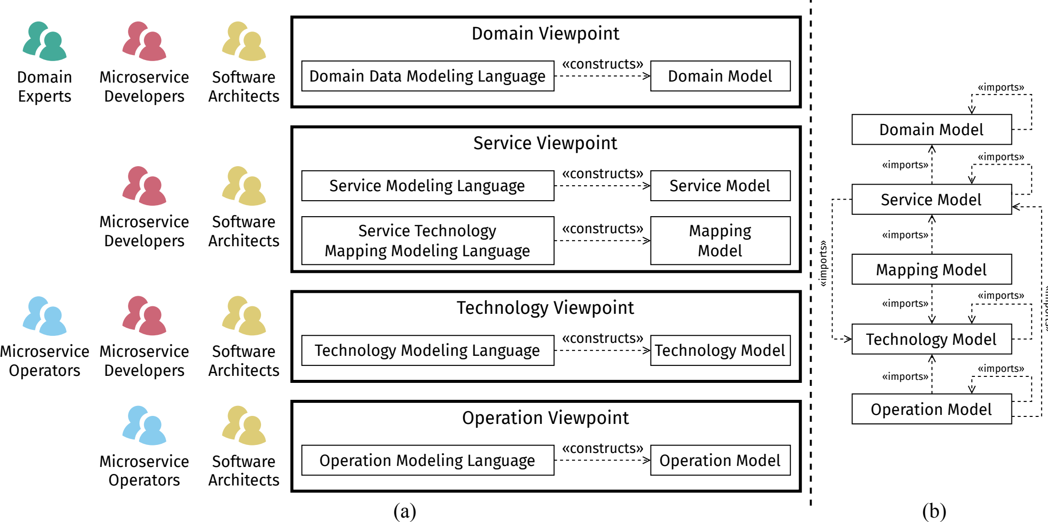


Figure 4 (a) LEMMA's viewpoints, their stakeholders, modeling languages and model types. (b) Import relationships between LEMMA model types.

- **Operation Model:** Operation models import other operation models to identify dependencies between operation nodes. Moreover, they can import technology and service models for node configuration and container-based microservice deployment, respectively.

Next to viewpoint-specific modeling languages, LEMMA also bundles its own MPF to facilitate the implementation of model processors by technology-savvy MSA stakeholders, e.g., microservice developers and operators (Table 4). The MPF applies the Phased Construction pattern [32] to structure model processing into the following phases:

1. **Model Parsing:** Parses input LEMMA models into object graphs to allow their efficient traversal. The phase is automated and does not require knowledge of MDE technologies.
2. **Model Validation:** Supports the implementation of validity checks as part of model processors. For instance, a code generator for event-based microservices might first ensure that modeled microservices exhibit operations that actually permit event handling.
3. **Code Generation:** Since code generation is among the key drivers for practical MDE adoption [66, 4, 33], LEMMA's MPF integrates a corresponding phase. However, since the MPF does not impose any requirements on generated code, the Code Generation phase may also be exploited to reify results from other model processing purposes, e.g., quality attribute values calculated during static model analysis [49].

LEMMA's MPF draws on popular MSA technologies and mechanisms such as the Java programming language [57, 7] and class-based Inversion of Control [26, 58]. Therefore, it provides specialized Java annotations for the Model Validation and Code Generation phases. The MPF detects these annotations at runtime on classes and methods, and handles annotated elements as intended by the phase, e.g., for signaling errors in validated models.

C Complexity Assessment for Extending LEMMA with Support for All Architectural Patterns in Sect. 2

Table 4 Description of LEMMA's viewpoints, stakeholders, modeling languages and model types.

Viewpoint	Stakeholders	Modeling Languages
Domain	Domain Experts, Microservice Developers, Software Architects	*Domain Data Modeling Language*: Enables to construct *domain models* with domain-specific data structures, collections, and enumerations via Domain-driven Design [15].
Service	Microservice Developers, Software Architects	*Service Modeling Language*: Supports the construction of *service models* for microservices, interfaces, and operations. *Service Technology Mapping Modeling Language*: Enables the construction of *mapping models* that assign technology-specific information to domain or service model elements. Models can therefore remain technology-agnostic and reusable across alternative technology choices [36].
Technology	Microservice Operators, Microservice Developers, Software Architects	*Technology Modeling Language*: Allows the construction of *technology models* that capture types of service programming languages, communication protocols, and operation technologies. In addition, generic *technology aspects* can be specified to augment elements in LEMMA models, e.g., data structures and microservices, with technology-specific information like database mappings or endpoint locations.
Operation	Microservice Operators, Software Architects	*Operation Modeling Language*: Constructs *operation models* for microservice deployment and infrastructure including its usage by services.

Table 5 Assessed complexity of extending LEMMA with support for the architectural MSA patterns described in Sect. 2.

Category	Pattern	LEMMA Viewpoints			Quantities[a)]			Complexity of Extension[b)]
		Dom.	Serv.	Op.	Concepts	Constraints	LOC[c)]	
Communication	**API Gateway**			✓	○	○	○	○
	Comment: Requires only one infrastructure component.							
	API Composition	✓	✓	✓	●	●	●	●
	Comment: No built-in modeling concepts for API composition.							
	Gateway Routing			✓	○	○	○	○
	Comment: Requires identification of API gateway as router.							
	Gateway Offloading		✓	✓	●	○	●	●
	Comment: Number of concepts likely increases with offloaded service functionality.							
	Event Sourcing	✓	✓	✓	○	◐	●	◐
	Comment: Requires persisting message broker component. Services interacting with broker must only send/receive domain events.							
	Log Aggregator			✓	○	○	○	○
	Comment: Requires only one infrastructure component.							

Category	Pattern	LEMMA Viewpoints			Quantities[a)]			Extension Complexity[b)]
		Dom.	Serv.	Op.	Concepts	Constraints	LOC[c)]	
Deployment	**Sidecar**		✓		◐	◐	●	◐
	Comment: Requires Sidecar identification and assignment to business-oriented service.							
	Microservice Chassis		✓		●	●	●	●
	Comment: By contrast to Sidecar: Requires delegating of all cross-cutting concerns.							
	Backend for Frontend	✓	✓	✓	○	○	●	◐
	Comment: Additional separate backends for highly diverse, client-specific frontends.							
	Database is the Service			✓	○	◐	○	○
	Comment: Requires validation that each service has its own database.							
Design	**Processing Resource**	✓	✓		○	○	●	N/A
	Comment: Fully supported by LEMMA's domain and service modeling concepts.							
	Parameter Tree	✓			◐	◐	○	◐
	Comment: Requires identification of domain concept containments.							
	API Key	✓	✓	✓	○	◐	●	●
	Comment: Requires identification of API key fields and mature security infrastructure.							
DevOps	**Externalized Configuration**		✓	✓	○	◐	◐	◐
	Comment: Requires configuration provider component to be used by services.							
	Monitor			✓	○	○	●	●
	Comment: Requires monitor component to be used by services.							
	Application Metrics			✓	○	○	●	●
	Comment: Requires metrics collector component to be used by services.							
	Distributed Tracing		✓	✓	●	●	●	●
	Comment: No built-in modeling concepts for service interaction.							
	Health Check			✓	○	○	●	●
	Comment: Requires health checker component to be used by services.							
Migration	**Strangler**	✓	✓	✓	○	○	●	N/A
	Comment: Fully supported by LEMMA's modeling concepts for the viewpoints.							
	Anti-Corr. Layer	✓	✓		○	○	●	N/A
	Comment: Fully supported by LEMMA's domain and service modeling concepts.							
	Decompose by Business Cap.	✓			N/A	N/A	N/A	N/A
	Comment: Pre-MSA systems are out of LEMMA's scope.							

Category	Pattern	LEMMA Viewpoints			Quantities[a)]			Extension Complexity[b)]
		Dom.	Serv.	Op.	Concepts	Constraints	LOC[c)]	
	Decompose by Subdomain	✓			N/A	N/A	N/A	N/A
	Comment: Pre-MSA systems are out of LEMMA's scope.							
Orchestration	**Container**			✓	○	○	●	N/A
	Comment: Fully supported by LEMMA's Operation Modeling Language.							
	Serv. Registry			✓	○	○	○	○
	Comment: Requires only one infrastructure component.							
	Serv. Discovery			✓	○	○	○	○
	Comment: Requires only one infrastructure component.							
	Load Balancer			✓	○	○	○	○
	Comment: Requires only one infrastructure component.							
	Circuit Breaker		✓		○	○	◐	○
	Comment: Requires identification of circuit breaker capability on services.							
	Saga	✓	✓	✓	●	●	●	●
	Comment: No built-in modeling concepts for service interactions.							

Symbol key: ○ = Low; ◐ = Middle; ● = High.
a) Assessed relative across all patterns.
b) Assessed based on quantities, and experiences with DOMAIN EVENT (Sect. 3) and CQRS (Sect. 4).
c) Assessed quantity of generated code.