Towards Holistic Modeling of Microservice Architectures Using LEMMA

Florian Rademacher[1], Jonas Sorgalla[1], Philip Wizenty[1] and Simon Trebbau[1]

[1]*IDiAL Institute, University of Applied Sciences and Arts Dortmund, Otto-Hahn-Straße 27, 44227 Dortmund, Germany*

Abstract

Microservice Architecture (MSA) is an approach for the realization of scalable and maintainable software systems. However, MSA adoption also increases architecture complexity significantly when compared to monolithic applications. This paper investigates MSA as an object of study for the development and application of architecture modeling languages (AMLs) to facilitate MSA engineering through Model-driven Engineering and lifted abstraction. To this end, we present a case study microservice architecture from the Electromobility domain and identify modeling dimensions to employ AMLs in MSA engineering. Next, we illustrate AML adoption for certain dimensions using the AMLs from our Language Ecosystem for Modeling Microservice Architecture (LEMMA). With these contributions, we aim to provide insights on MSA as a driver for research on holistic AML adoption throughout architecture design, development, and operation.

Keywords

microservice architecture, model-driven engineering, architecture modeling languages

1. Introduction

Microservice Architecture (MSA) is a novel approach for the realization of service-based software architectures [1]. MSA emerged from Service-oriented Architecture (SOA) [2, 3] and promotes to decompose software architectures into *microservices*. A microservice is a *service* [2] that puts particular emphasis on (i) cohesion by fulfilling a single, distinct task; (ii) independence in terms of its implementation, data management, testing, deployment, and operation; and (iii) responsibility w.r.t. its interaction with other components and ownership by exactly one team [1, 4].

Based on these characteristics, MSA adoption is expected to benefit a software architecture's (i) performance efficiency because microservices are independently scalable; (ii) maintainability by allowing targeted modification or replacement of functionality given microservices' high cohesion and loose coupling; and (iii) reliability due to microservices' constrained functional scope and their self-responsibility for fault handling [5, 1, 6, 7].

On the other hand, MSA tends to increase the complexity of a software architecture because it poses significant challenges concerning architecture design, development, and operation [8]. For example, regarding design, MSA requires microservice identification and afterwards careful balancing of microservices' granularity. Too fine-grained microservices increase network load, thereby decreasing performance [9], whereas too coarse-grained services counteract scalability. In addition, MSA fosters *technology heterogeneity* [1] by enabling teams to independently decide for implementation technologies, e.g., frameworks and databases, which may result in additional maintainability cost and steeper learning curves for new team members [10]. Moreover, MSA assumes a sophisticated deployment infrastructure including specialized components, e.g., for service discovery, API provisioning, load balancing, and monitoring [11].

This paper investigates MSA as an object of study for the design, implementation, and application of *architecture modeling languages* (AMLs), i.e., architecture description languages that constitute modeling languages in the sense of Model-driven Engineering (MDE) [12, 13]. Our hypothesis is that AMLs can reduce the complexity in MSA engineering by introducing abstraction to, e.g., (i) facilitate reasoning about granularity by reifying service boundaries; (ii) make technology choices explicit; and (iii) support the specification of operation infrastructure. Our contribution is twofold. First, we present a case study microservice architecture from the Electromobility domain and identify *modeling dimensions* [13] to employ AMLs in the context of MSA. Second, we illustrate AML adoption for certain modeling dimensions of the case study using the Language Ecosystem for Modeling Microservice Architecture (LEMMA), which is a set of AMLs for MSA, that we developed in our previous work [14, 15]. With both contributions, we aim to provide insights on MSA as a driver for AML research w.r.t. *holistic* architecture modeling, i.e., the usage of AMLs throughout architecture design, development, and operation.

The remainder of the paper is organized as follows.

MDE4SA 2021: Second International Workshop on Model-driven Engineering for Software Architecture, September 13, 2021, Virtual

florian.rademacher@fh-dortmund.de (F. Rademacher); jonas.sorgalla@fh-dortmund.de (J. Sorgalla); philip.wizenty@fh-dortmund.de (P. Wizenty); simon.trebbau@fh-dortmund.de (S. Trebbau)
0000-0003-0784-9245 (F. Rademacher); 0000-0002-7532-7767 (J. Sorgalla); 0000-0002-3588-5174 (P. Wizenty)

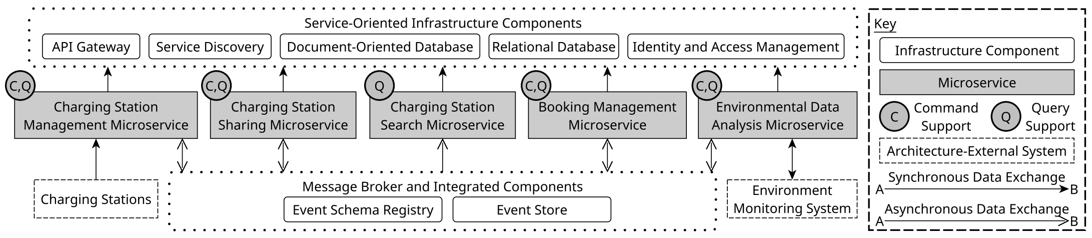
Service-Oriented Infrastructure Components
API Gateway
Service Discovery
Document-Oriented Database
Relational Database
Identity and Access Management
C,Q
Charging Station Management Microservice
C,Q
Charging Station Sharing Microservice
Q
Charging Station Search Microservice
C,Q
Booking Management Microservice
C,Q
Environmental Data Analysis Microservice
Charging Stations
Message Broker and Integrated Components
Event Schema Registry
Event Store
Environment Monitoring System
Key
Infrastructure Component
Microservice
C Command Support
Q Query Support
Architecture-External System
Synchronous Data Exchange
A B
Asynchronous Data Exchange
A B

Figure 1: Service-based design of the PACP including shared infrastructure components.

Section 2 introduces the case study microservice architecture. Section 3 derives modeling dimensions from the case study. In Section 4, we apply LEMMA to certain modeling dimensions by using its AMLs to express various parts of the case study architecture. Section 5 discusses the resulting insights on holistic architecture modeling for MSA. Sections 6 and 7 present related work and conclude the paper, respectively.

2. Park and Charge Platform Case Study

This section introduces the microservice architecture of the Park and Charge Platform (PACP), which we will use throughout the paper as a case study to illustrate and discuss holistic architecture modeling in the context of MSA. The PACP constitutes one of the deliverables of the PuLS research project[1]. PuLS aims to increase the accessibility of charging stations for electric vehicles by enabling citizens to offer spare stations on private ground for use by other owners of electric vehicles. In addition, charging stations are equipped with sensors to contribute in urban air quality monitoring.

In PuLS, we design, develop, and operate the PACP to handle charging station offering and booking, and the storage and analysis of air quality indicators. The PACP is a microservice architecture to ensure (i) scalability of the solution across city quarters; (ii) modifiability to foster innovation through quick functionality integration; and (iii) technology heterogeneity, in particular of programming languages used by project partners. Figure 1 shows the PACP's design.

The following paragraphs describe the microservices and infrastructure components of the PACP.

Microservices The PACP consists of five microservices, whose design permits runtime replication of service instances. In detail, the services provide the architecture with the following capabilities:

- `Charging Station Management Microservice`: Manages charging station information like location, charging type, and plug type, and receives data from charging stations.
- `Charging Station Sharing MS`: Realizes functionality for citizens to offer spare charging stations on private ground for use by others under certain conditions and for a given time period.
- `Charging Station Search MS`: This service implements functionality to search for spare charging stations.
- `Booking Management MS`: Enables owners of electric vehicles to book spare charging stations. For this purpose, the service maintains a Blockchain [16] to prevent manipulation during booking, and subsequent charging and billing processes.
- `Environmental Data Analysis MS`: Supports handling of air quality indicators, e.g., CO2 pollution, temperature, and humidity, and therefore interacts with a municipal Environment Monitoring System.

The listed PACP microservices constitute *logical microservices* in the sense of the Command Query Responsibility Segregation (CQRS) pattern [17]. CQRS decomposes a microservice into a *command part* and *query parts.* The command part handles incoming requests that result in state changes, e.g., database updates, and the query parts execute state queries, e.g., database reads. To this end, the command part sends state changes to query parts, which then incorporate the changes into their data models. For the PACP, the usage of CQRS has two benefits. First, query operations are much more frequent than write operations, and CQRS permits separate scaling of command and query parts because we realize them as *physically segregated microservices.* Second, we can optimize storage for query operations and thus, e.g., enable time series processing of sensor data.

Infrastructure Components Figure 1 considers two kinds of infrastructure components.

Service-oriented infrastructure components provide a single microservice with capabilities like API provision-

[1]Funding by the German Federal Ministry of Transport and Digital Infrastructure (grant number 03EMF0203C).

Table 1
Initial modeling dimensions according to Combemale et al. [13] with relevance to MSA.

#	Modeling Dimension	Stage	Associated Pains According to Soldani et al. [8]
D.1	Exploration	Design	Service Dimensioning, Size/Complexity (S/C)
D.2	Communication	Design	Service Contracts, S/C
D.3	Construction	Design	API Versioning, Communication Heterogeneity, Service Contracts, Microservice Separation, S/C
D.4	Implementation	Development Operation	Microservice Separation, Overhead, Human Errors Operational Complexity, Service Coordination, S/C
D.5	Testing	Development	Integration Testing, Performance Testing, S/C
D.6	Documentation	Design	S/C

ing (`API Gateway`), discovery of other PACP microservices (`Service Discovery`), and data management (`Document-Oriented Database` and `Relational Database`). Furthermore, the PACP secures accesses via the `Identity and Access Management` (IAM) component, which realizes user management, authentication, and authorization of microservices' command and query functionalities.

On the other hand, PACP microservices interact with each other by sending asynchronous events to the centralized `Message Broker` component (cf. Fig. 1). Following the Domain Event pattern [17], we conceive exchanged events *domain events* that belong to the portion of the application domain for which a microservice is responsible. To this end, the message broker integrates an *event schema registry* and behaves as an *event store*. The registry allows centralized management of event structures and their sharing across project partners, whereas the event store permits access to events in their order of appearance and thus auditing, e.g., of charging station booking processes.

3. Modeling Dimensions for Microservice Architecture

Based on our experience in realizing the PACP (cf. Sect. 2), we identify initial modeling dimensions according to Combemale et al. [13] and with relevance to MSA. We consider AMLs central means to construct models for these dimensions and later enable their processing. Table 1 lists the identified modeling dimensions together with the related stage and *pains* [8] in MSA engineering.

The following paragraphs summarize per stage the rationale for each dimension.

Design Stage In microservice design, models can make a microservice's granularity explicit by reifying the structures and relationships of domain concepts in the service's responsibility (cf. Dimension D.1 in Table 1). This modeling purpose aligns to the construction of microservices' *domain models* using Domain-driven Design (DDD) [18, 1]. DDD is a model-based methodology, which we used in the PACP's design to capture the structures and relationships of the relevant concepts from the application domain.

Moreover, models are a means to communicate and document, e.g., domain concepts or service contracts, across teams (D.2 and D.6). In MSA, efficient communication and a common architectural understanding is crucial since teams and their communication should be decomposed along service boundaries [4]. For the PACP, we rely on LEMMA models and derived artifacts to share microservice APIs and event schemas across teams (cf. Sect. 4).

Next to domain concept definition, models can be constructed in MSA engineering, e.g., to specify APIs and their versions, or reason about communication heterogeneity (D.3).

Development Stage Code generators may support the development of microservices (D.4) by producing boilerplate code from models [13, 19]. We use this approach for the PACP to reduce manual overhead and human errors in recurring coding tasks like connecting a service to the message broker (cf. Fig. 1). Furthermore, it enables us to keep the model-based architecture design consistent with the implementation of microservices' domain data, APIs, and deployment specifications (cf. Sect. 4). In addition, it is possible to run early integration tests with the generated code or manually extend it, e.g., for subsequent performance testing (D.5).

Operation Stage For the PACP, the use of models is also beneficial in the operation stage of MSA engineering. More precisely, we use models to harmonize the description of service deployment, infrastructure operation and usage across heterogeneous technologies. The resulting model-based description of the PACP's operation specifics then facilitates reasoning about the architecture's operational complexity (D.4).

4. Modeling Park and Charge Case Study Microservices with LEMMA

This section illustrates the modeling of PACP microservices (cf. Sect. 2) along the dimensions from Sect. 3 using LEMMA. LEMMA specifies textual AMLs for the modeling of MSA-based software systems from various *architecture viewpoints* [20]. Each LEMMA viewpoint clusters one or more AMLs, whose *metamodels* [13] formalize MSA concepts and enable stakeholders to state their concerns towards a microservice architecture [15]. By using LEMMA as a concrete modeling approach for MSA, we aim to provide insights on MSA as a driver for AML research w.r.t. the holistic usage of AMLs in microservice design, development, and operation.

Each of the following subsections presents an excerpt of one or more LEMMA models for a certain viewpoint on the PACP's `Charging Station Management Microservice` (`CSMM`; cf. Sect. 2). The complete code of all PACP models can be found on GitHub[2].

4.1. Modeling Microservices' Domain Data

As described in Sect. 3, a microservice's granularity shall result from its responsibility in the application domain. LEMMA specifies the Domain Viewpoint and its Domain Data Modeling Language (DDML) [15] to allow domain experts and microservice developers the model-based identification and clustering of services' domain concepts. The DDML covers modeling dimensions D.1 and D.3 (cf. Table 1) as it enables DDD-based organization of domain concepts in *bounded contexts* [18] and the separation of microservice data based on these contexts [1].

Listing 1 shows an excerpt of the `CSMM` domain model in LEMMA's DDML.

Listing 1: Excerpt of the `CSMM` domain model in LEMMA's DDML (file "domain.data").

```
1  context ChargingStationManagement {
2   structure ElectrifiedParkingSpace<entity, aggregate> {
3    string id<identifier>,
4    string name,
5    string plugType,
6    ChargingType chargingType<part>,
7    ParkingSpaceSize parkingSpaceSize<part>,
8    ...
9   }
10  enum ChargingType{
11   FAST,
12   NORMAL
13  }
14  structure ElectrifiedParkingSpaceCreated<valueObject,
15   domainEvent> {
16   immutable string name,
17   ...
18 }}
```

[2]https://www.github.com/SeelabFhdo/mde4sa-2021

The `CSMM` team constructed the domain model in collaboration with domain experts using DDD. Line 1 defines the bounded context `ChargingStationManagement`, which comprises three domain concepts.

The `ElectrifiedParkingSpace` concept in Lines 2 to 9 represents a parking space with a charging station. The concept is a DDD `entity` and thus has a domain-specific identity [18] determined by the `id` field, which therefore exhibits the DDML's `identifier` keyword [15] (cf. Line 3). In addition, the concept clusters the fields `name` and `plugType` (cf. Lines 4 and 5), which, like `id`, are of LEMMA's built-in `string` type.

Moreover, the `ElectrifiedParkingSpace` concept is a DDD `aggregate`. As such, it embeds other domain concepts like `ChargingType` and `ParkingSpaceSize` in the form of `parts` (cf. Lines 6 and 7), and defines a transactional boundary, e.g., for database access [18]. Consequently, instances of embedded domain concepts must not exist without a corresponding `ElectrifiedParkingSpace` instance.

Lines 10 to 13 illustrate the DDML's support for modeling enumerated domain concepts. Since all concepts defined in LEMMA domain models constitute *custom types* [15], it is possible to use the `ChargingType` enumeration as a field type and embed it in the `ElectrifiedParkingSpace` concept.

Lines 14 to 18 model the `ElectrifiedParkingSpaceCreated` concept as a DDD `valueObject` and `domainEvent` [18, 21]. The `CSMM`'s command microservice (cf. Sect. 2) uses the concept to inform query microservices about new parking spaces (cf. Sect. 4.4). LEMMA's DDML provides the `immutable` keyword to protect value objects against state changes (cf. Line 16). As opposed to entities, DDD recognizes value objects to model domain concepts that do not have domain-specific identities [18]. Instead, the identities of their instances result from the values of all fields and changes to the states of value object instances should require new value object instances.

4.2. Modeling Technology and Pattern Metadata

LEMMA specifies the Technology Viewpoint to support coping with MSA's technology heterogeneity [1]. For this purpose, the viewpoint comprises the Technology Modeling Language (TML) [14]. It covers modeling dimensions D.4 and D.5 in the Development stage of MSA engineering (cf. Table 1) by enabling the construction of technology models. These models can capture a variety of MSA-related technology information concerning, e.g., microservice programming languages, communication protocols, and operation technologies. However, it also allows the definition of arbitrary metadata using *technology aspects* [14]. Such aspects may augment elements in other LEMMA models with additional semantics regarding, e.g., technology-specific configuration options, but also pattern concepts.

The following paragraphs describe the usage of LEMMA's TML for the construction of technology models that reify microservice technologies and pattern concepts, respectively.

Modeling Technology Metadata with the TML In Listing 2, we present an excerpt of a technology model for the Spring framework[3] and thus the technology on which the majority of PACP microservices rely for their implementation.

Listing 2: Excerpt of the technology model for the Spring framework in LEMMA's TML (file "Spring.technology").

```
1  technology Spring {
2   protocols {
3    sync rest data formats "application/json"
4     default with format "application/json";
5   }
6   service aspects {
7    aspect PutMapping<singleval> for operations {
8     selector(protocol = rest);
9    }
10   ...
11 }}
```

Line 1 introduces the `Spring` technology. Next, the `protocols` section in Lines 2 to 5 specifies the synchronous `rest` protocol, which represents REST-based interaction [22] between PACP microservices and external consumers (cf. Sect. 2). Protocol definitions in the TML must also provide information about supported data formats. Thus, in Line 3 we express the support of the `rest` protocol for the JSON data format[4] and also select it as the protocol's default format in Line 4.

Lines 6 to 11 cluster the `service aspects` section of the Spring technology model. The section illustrates the definition of the `PutMapping` aspect to reify the eponymous Spring annotation[5]. LEMMA distinguishes between service-related and operation-related technology aspects [14]. Service-related technology aspects are applicable to elements of modeled microservices (cf. Sects. 4.3 and 4.4), whereas operation-related technology aspects target elements of modeled operation nodes (cf. Sect. 4.5).

Furthermore, LEMMA allows constraining aspects' applicability depending on the peculiarities of target elements. For example, the `PutMapping` aspect is applicable at most once to modeled microservice operations (cf. the `singleval` keyword and the `for operations` directive in Line 7). Additionally, the target operation must make use of the `rest` protocol (cf. the protocol `selector` in Line 8). These three constraints map to the semantics of Spring's `PutMapping` annotation, which elevates Java methods to handlers for HTTP `PUT` requests [23].

[3]https://www.spring.io
[4]https://www.json.org
[5]https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/PutMapping.html

Modeling Pattern Metadata with the TML Since the TML itself does not constrain the semantics of technology aspects and supports their usage on a variety of modeled elements, e.g., microservices, their interfaces, operations and containers [14], we also use them to reify information about concepts from design and architecture patterns relevant to MSA. More specifically, LEMMA's aspect mechanism enables to capture pattern metadata and augment modeled elements with them so that the elements become semantically recognizable as being involved in the realization of a design or architecture pattern. Listing 3 shows the technology model for the CQRS pattern as used by the PACP (cf. Sect. 2).

Listing 3: Excerpt of the technology model for the CQRS pattern in LEMMA's TML (file "Cqrs.technology").

```
technology CQRS {                              1
 service aspects {                             2
  aspect CommandSide for microservices {       3
   string logicalService;                      4
  }                                            5
  aspect QuerySide for microservices {         6
   string logicalService;                      7
  }                                            8
  ...                                          9
}}                                             10
```

The model specifies the `CQRS` technology (cf. Line 1) and defines two service-related technology aspects (cf. Lines 2 to 10). The `CommandSide` aspect in Lines 3 to 5 enables the augmentation of modeled microservices that constitute the physical command microservices in adoptions of the CQRS pattern (cf. Sect. 2). The `QuerySide` aspect in Lines 6 to 8, on the other hand, supports the semantic enrichment of modeled microservices, which represent physical query microservices in a CQRS scenario. Both aspects declare the `logicalService` property (cf. Lines 4 and 7). It can store the name of the logical microservice to which a set of physical CQRS microservices belongs.

The `CommandSide` and `QuerySide` aspects provide, e.g., model analyzers with a reliable means to recognize the command and query services of a logical CQRS microservice in order to verify that all query microservices provide operations to consume update events from the command microservice (cf. Sect. 4.4).

4.3. Modeling Microservices' Application Programming Interfaces

LEMMA defines the Service Viewpoint for developer concerns in microservice implementation [15]. The viewpoint specifies the Service Modeling Language (SML) for the model-based expression of microservices, their interfaces, operations and endpoints. The SML supports microservice separation and the design of APIs as *implicit service contracts* [24], and thus covers modeling dimensions D.2, D.3, and D.4 (cf. Table 1).

Listing 4 shows an excerpt of the service model for the CSMM's physical command microservice in LEMMA's SML. The model also reifies technology choices based on LEMMA's TML (cf. Sect. 4.2).

Listing 4: Excerpt of the service model for the CSMM's command microservice in LEMMA's SML including technology choices based on the TML (file "chargingStationManagement.services").

```
1  import datatypes from ”domain.data” as Domain
2  import technology from ”Spring.technology” as Spring
3  @technology(Spring)
4  @Spring::_aspects.Application(
5   name=”ChargingStationManagementCommand”,
6   port=8071
7  )
8  public functional microservice
9   de.fhdo.puls.ChargingStationManagementCommand {
10  @endpoints(
11   Spring::_protocols.rest: ”/resources/v1”;
12  )
13  interface Commands {
14   @endpoints(
15    Spring::_protocols.rest: ”/electrifiedParkingSpace”;
16   )
17   @Spring::_aspects.PutMapping
18   public createElectrifiedParkingSpace(
19    @Spring::_aspects.RequestBody
20    sync in command : Domain::ChargingStationManagement.
21     CreateElectrifiedParkingSpaceCommand);
22  }
23  ...
24 }
```

LEMMA's AMLs provide an import mechanism to integrate models from different viewpoints [15]. A model import must specify (i) the kind of the imported model elements (after the `import` keyword); (ii) the path to the imported model (after the `from` keyword); and (iii) an import alias (after the `as` keyword). Line 1 relies on LEMMA's import mechanism to import the CSMM domain model (cf. Listing 1) under the alias `Domain`. The import of domain models by service models determines the portion of the application domain, for which a modeled microservice is responsible. Line 2 imports the Spring technology model (cf. Listing 2) under the alias `Spring`. As described in Sect. 4.2, technology model imports integrate LEMMA's Technology Viewpoint with the Service Viewpoint so that modeled microservices can be augmented with information that reflect technology choices.

Lines 3 to 24 model the CSMM's command microservice. Line 3 uses the SML's `@technology` annotation to assign the microservice the imported `Spring` technology model. In order to configure the name and port of the Spring application that realizes the microservice[6], Lines 4 to 7 apply the `Application` aspect from the `Spring` technology model to the modeled microservice.

Line 8 introduces the microservice's definition. LEMMA provides modifiers to constrain a microservice's visibility to, e.g., architecture-internal components or the owning team [15]. The CSMM's command microservice, however, exhibits the `public` modifier so that it will be reachable by architecture-external components like charging stations (cf. Sect. 2). In addition, the service is of a `functional` nature. Hence, it provides a business-related capability to the architecture, and does not serve infrastructure or generic utility purposes [15].

Lines 10 to 13 introduce the microservice's commands API as the `Commands` interface with a REST endpoint. The SML integrates the `@endpoints` annotation for endpoint specifications that constitute combinations of a technology-specific protocol like the `rest` protocol from the imported `Spring` technology model (cf. Sect. 4.2), and one or more addresses like the URI segment "/resources/v1".

Lines 14 to 21 define the `createElectrifiedParkingSpace` operation as part of the `Commands` interface. Callers invoke the operation to trigger the creation of electrified parking spaces managed by the CSMM (cf. Sect. 4.1). Lines 14 to 17 determine the URI segment and HTTP request method [23] for the REST-based invocation of the operation. While the URI segment is again configured as an endpoint address for the imported `rest` protocol, the operation receives the request method via the `PutMapping` technology aspect from the Spring technology model (cf. Listing 2). Line 18 introduces the `createElectrifiedParkingSpace` operation and Lines 19 to 21 define its synchronous incoming parameter `command` [15]. The type of the parameter corresponds to a concept from the CSMM's domain model that constitutes a structured command for the creation of a newly managed parking space. Hence, the parameter also receives an application of the `RequestBody` aspect from the `Spring` technology model. This aspect maps to the eponymous Spring annotation[7], which leads to the extraction of parameter values from the request bodies of inbound HTTP requests.

4.4. Modeling Asynchronous Microservice Interaction

As described in Sect. 2, PACP microservices interact asynchronously using a message broker and events. We decided for Kafka[8] as broker technology and most events originate from command microservices informing query microservices about state changes.

To model Kafka-based sending of such CQRS update events by command services, we again rely on LEMMA's

[6]https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html

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

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

SML as it supports the (i) specification of event production and receipt by service operations; and (ii) augmentation of model elements with broker and pattern information from technology models (cf. Sect. 4.2). Listing 5 shows an extended version of the service model for the CSMM's command microservice (cf. Listing 4) with elements for asynchronous event sending.

Listing 5: Extended version of the service model for the CSMM's command microservice (cf. Listing 4) with elements for asynchronous microservice interaction.

```
1  ...
2  import technology from "Kafka.technology" as Kafka
3  import technology from "Cqrs.technology" as CQRS
4  ...
5  @technology(Kafka)
6  @technology(CQRS)
7  @endpoints(Kafka::_protocols.kafka: "kafka-server1:9092";)
8  @CQRS::_aspects.CommandSide("ChargingStationManagement")
9  functional microservice
10  de.fhdo.puls.ChargingStationManagementCommand {
11  ...
12  interface Commands {
13   ...
14   @Kafka::_aspects.Participant(
15    topic="parkingSpaceCreatedEvents"
16   )
17   sendParkingSpaceCreatedEvent(
18    async out event : Domain::ChargingStationManagement
19     .ElectrifiedParkingSpaceCreated);
20 }}
```

Lines 2 and 3 add imports for the Kafka technology model[9] and the CQRS technology model (cf. Listing 3), respectively. Next, Lines 5 and 6 apply both models to the command microservice. Consequently, we can configure an endpoint for the kafka protocol from the Kafka technology model to specify the location of the Kafka broker (cf. Line 7). Moreover, we apply the CommandSide aspect from the CQRS technology model to the microservice (cf. Line 8) so that it is semantically recognizable as the physical command microservice of the logical "ChargingStationManagement" microservice (cf. Sect. 4.2).

Lines 14 to 19 define the sendParkingSpaceCreatedEvent operation for sending Kafka events after the creation of newly managed electrified parking spaces. To this end, the Participant aspect configures the event's topic. In addition, the operation specifies the asynchronous outgoing parameter event [15], which is typed by the ElectrifiedParkingSpaceCreated domain event from the imported CSMM domain model (cf. Listing 1).

4.5. Modeling Microservice Deployment

In the following, we accompany the domain and service model of the CSMM's command microservice (cf. Sects. 4.1, 4.3, and 4.4) with an operation model for LEMMA's Operation Viewpoint [15]. The viewpoint specifies the Operation Modeling Language (OML), which covers the operation-related modeling dimension D.4 (cf. Table 1) and makes the operation infrastructure of a microservice architecture explicit. Listing 6 shows an excerpt of the operation model for the CSMM's command microservice.

Listing 6: Excerpt of the operation model for the CSMM's command microservice in the OML (file "chargingStationManagement.operation").

```
import microservices                                                1
 from "chargingStationManagement.services" as Services              2
import technology                                                   3
 from "container_base.technology" as ContainerBase                  4
import nodes from "eureka.operation" as ServiceDiscovery            5
import nodes from "keycloak.operation" as IAM                       6
import nodes from "mongodb.operation" as Database                   7
@technology(ContainerBase)                                          8
container CommandContainer                                          9
deployment technology ContainerBase::_deployment.Kubernetes         10
with operation environment "openjdk:11-jdk-slim"                    11
deploys Services::de.fhdo.puls.                                     12
 ChargingStationManagementCommand                                   13
depends on nodes ServiceDiscovery::Eureka, Database::MongoDB,       14
 IAM::Keycloak {                                                    15
 default values {                                                   16
  eurekaUri="http://discovery-service:8761/eureka"                  17
  ...                                                               18
 }                                                                  19
}                                                                   20
```

Lines 1 to 4 import the service model of the CSMM's command microservice (cf. Listing 4) and a technology model for container technology[10]. Lines 5 to 7 import three other LEMMA operation models that specify the PACP's service discovery and IAM component as well as the database of the CSMM's command microservice (cf. Sect. 2). LEMMA supports the decomposition of operation models to separate definitions of centralized infrastructure components, e.g., service discoveries, from those of microservice-specific components, e.g., containers.

Lines 8 to 20 model the CommandContainer to deploy the CSMM's command microservice. For this purpose, the container applies the imported ContainerBase technology model and leverages the provided Kubernetes deployment technology[11] with a Java image for its execution (cf. Lines 10 and 11). LEMMA's support for container-based deployment also determines microservices' technical replicability [4]. More precisely, modeled microservices (cf. Sect. 4.3) act as templates for runtime service instances, whose replicability characteristics are to be determined by modeled containers.

Lines 12 and 13 specify that the modeled container deploys the imported CSMM command microservice.

Lines 14 and 15 configure the container to depend on the PACP's Eureka-based service discovery[12] and Keycloak-based IAM provider[13] as well as the MongoDB database technology[14] for document-oriented storage of

[9]https://www.github.com/SeelabFhdo/mde4sa-2021/blob/master/technology/Kafka.technology

[10]https://www.github.com/SeelabFhdo/mde4sa-2021/blob/master/technology/container_base.technology

[11]https://www.kubernetes.io

[12]https://www.github.com/Netflix/eureka

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

[14]https://www.mongodb.com

charging station information (cf. Sect. 2).

Finally, the container uses the `default values` section of LEMMA's OML [15] in Lines 16 to 19 to determine values for technology-specific configuration options that account for all deployed microservices. More precisely, the `eurekaUri` option receives the URI to the PACP's service discovery so that the deployed `CSMM` command microservice can leverage its capabilities.

5. Discussion

This section provides insights on MSA as a driver for AML research w.r.t. holistic architecture modeling. Therefore, we discuss experiences from adopting LEMMA's AMLs to the PACP (cf. Sects. 2 and 4) in the different stages of MSA engineering (cf. Sect. 3).

5.1. AMLs in MSA Design

Concerning MSA design, LEMMA's DDML focuses on *tactical DDD*, i.e., the modeling of domain concepts *within* bounded contexts [18] (cf. Sect. 4.1). While tactical DDD allows determination of a microservice's granularity in terms of domain concept structures and relationships, it does not provide means to express domain-driven service interaction. For this purpose, *strategic DDD* is applicable as it classifies the relationships *between* bounded contexts [18]. Next to DDD, there also exist alternative approaches like Event Storming [25], which partition microservices and their interactions based on domain events. However, all these approaches use models to abstract from technical details, and foster the collaboration between domain experts and developers. For instance, strategic DDD relies on graphical *context maps* [18], while Event Storming combines a textual notation with box-and-line diagrams [25]. Thus, we perceive potential for AML research to study the effectiveness of these approaches and formalize them to allow automated reasoning of resulting models [13].

Furthermore, MSA enables teams to employ different approaches with varying degrees of autonomy in service design and realization [4]. For example, a team may own microservices, which incorporate *shared artifacts* [1] owned by other teams or which do not rely on such artifacts at all. Thus, AMLs for MSA must support distributed modeling including model evolution and integration. To this end, LEMMA allows, e.g., model decomposition within or across team boundaries using imports [26], and versioning of evolvable model elements [15].

5.2. AMLs in MSA Development

Technology abstraction is a key benefit of MDE and thus AMLs [13]. LEMMA enables technology-agnostic modeling in the DDML and SML (cf. Sects. 4.1, 4.3, and 4.4), and as-needed technology augmentation of models with the TML (cf. Sect. 4.2). This approach copes with MSA's technology heterogeneity [1] and facilitates the reconstruction of technology information from microservice implementations. However, it requires upfront technology model construction, and balancing of semantic-oriented and technology-oriented modeling. For example, the `PutMapping` aspect in Sect. 4.2 reifies the eponymous Spring annotation. Yet, it targets HTTP `PUT` requests so that the name `Put` fits better to the intended semantics. AML research could study trade-offs between semantic-oriented and technology-oriented modeling, e.g., by comparing the effectiveness of flexible AMLs like those of LEMMA with modeling languages that integrate keywords for MSA patterns or technologies (cf. Sect. 6).

Behavior modeling is another area for MSA-inspired AML research. Currently, none of LEMMA's AMLs supports behavior modeling w.r.t. service logic or data exchange as we initially perceived it technology-specific. In this respect, MSA represents an interesting field to study the integration of technology-specific behavior languages, e.g., programming languages, with modeling languages. However, AML research might also focus on adopting technology-agnostic behavior modeling languages, e.g., UML sequence diagrams, to MSA engineering for a facilitated reasoning about service interactions.

Due to MSA's technology heterogeneity, teams are free to employ AMLs in MSA engineering. Hence, AML research could study the collaboration of modeling and non-modeling teams. For the PACP, we implemented a set of model transformations to integrate LEMMA-based microservices with other teams' components. For synchronous service interactions, we support the transformation/derivation of LEMMA models to/from OpenAPI specifications[15]. For asynchronous service interactions, we transform/derive LEMMA models to/from Avro event specifications[16]. While this approach is sufficient for the PACP, it requires dedicated transformations as well as additional specification management.

5.3. AMLs in MSA Operation

In MSA operation, the usage of markup languages like YAML[17] is frequent for the textual specification of operation nodes and LEMMA's OML (cf. Sect. 4.5) aims to allow harmonization of heterogeneous textual specification approaches through models. As a result, AML research could next focus on model processing in the context of MSA operation. For instance, static analyzers may support in the reconstruction of MSA operation models from textual specifications. That is, because approaches like Kubernetes enable holistic operation specification

[15]https://www.openapis.org
[16]https://avro.apache.org
[17]https://www.yaml.org

ranging from service deployment to infrastructure configuration and usage.

6. Related Work

To the best of our knowledge, there currently exist no studies that investigate holistic AML adoption in MSA design, development, and operation. Hence, we present work related to AMLs for MSA, thereby focusing on the modeling of heterogeneous parts of microservice architectures to support holistic MDE adoption.

Le et al. [27] present the DcSL modeling language to bridge the gap between domain experts and software developers. LEMMA's DDML (cf. Sect. 4.1) follows a similar notion in addressing the concerns of domain experts and microservice developers. However, DcSL focuses on UML to capture domain information in models, and neither supports DDD patterns nor addresses distributed domain models as required in MSA engineering [1]. Moreover, LEMMA's DDML is part of an ecosystem dedicated to microservice architecture modeling, and permits domain concept referencing across models, e.g., to integrate the Domain Viewpoint with the Service Viewpoint (cf. Sects. 4.3 and 4.4). Additionally, Le et al. do not evaluate DcSL in the context of a cohesive case study (cf. Sect. 2).

Terzić et al. [28] present MicroBuilder to facilitate MSA engineering by MDE. The tool entails the MicroDSL language, which provides modeling concepts for data structures, service endpoints, and REST APIs. MicroDSL is evaluated by modeling the domain data and synchronous APIs of a web shop application. However, this evaluation omits the application of AMLs for the specification of asynchronous interaction and microservice operation as, unlike LEMMA (cf. Sects. 4.4 and 4.5), MicroDSL does not provide corresponding modeling concepts.

MDSL [29] is a modeling language with means to define microservice contracts including required and provided data. MDSL focuses on the expression of API providers and consumers with logical endpoint types. By contrast, LEMMA's SML considers microservice APIs to constitute collections of operations, which are organized in service-specific interfaces (cf. Sect. 4.3). Hence, MDSL focuses on a higher level of abstraction than our SML so that an integration of both languages seems beneficial. For instance, MDSL models may cluster information about microservice contracts in a technology-agnostic manner. These models could then be transformed to SML models, whose technology-specific extension would allow, e.g., subsequent microservice code generation.

CloudML [30] is a modeling language for the deployment of multi-cloud applications. It considers two levels of abstraction, i.e., the Cloud Provider-Independent Model (CPIM) and the Cloud Provider-Specific Model (CPSM). The CPIM relies on generic concepts like `NodeType` and `ArtefactType` to capture operation nodes and deployed artifacts provider-independently. A CPIM is then transformed to a provider-specific CPSM. Similarly to LEMMA's OML (cf. Sect. 4.5), CloudML focuses on operation aspects of cloud-native applications that may incorporate microservices. However, in the OML modeled nodes and the artifacts deployed to them always require technology information, which could involve cloud-provider-specific configuration profiles. On the other hand, CloudML ships with a definitive set of properties, e.g., `memory` for node types, to describe deployment. Thus, when compared to LEMMA's OML and its integration with the TML, CloudML lacks flexibility in adding new configuration properties. Furthermore, due to LEMMA's design as a modeling ecosystem, operation models in the OML can directly refer to artifact models, i.e., microservices in LEMMA's SML, thereby enabling holistic architecture modeling that combines design, development, and operation information.

7. Conclusion and Future Work

This paper investigated Microservice Architecture (MSA) [1] as an object of study for the research on architecture modeling languages (AMLs) [12] with a special focus on holistic AML adoption throughout MSA design, development, and operation. To this end, we first presented a case study microservice architecture (cf. Sect. 2). From the case study, we derived an initial set of modeling dimensions [13], and identified related stages and pains [8] in MSA engineering (cf. Sect. 3). Section 4 applied our Language Ecosystem for Modeling Microservice Architecture (LEMMA) [15] to construct models for the case study's domain data, technology choices, service APIs, and operation. The usage of LEMMA illustrated MSA's potential to stimulate AML research w.r.t. the model-based organization and integration of architecture concerns (cf. Sect. 5).

In the future, we plan to strengthen the presented insights on holistic AML adoption for architecture design, development, and operation by an empirical investigation of LEMMA's applicability for MSA practitioners. Given MSA's current popularity, such an investigation could particularly contribute to the clarification of benefits and challenges concerning industrial AML usage.

References

[1] S. Newman, Building Microservices: Designing Fine-Grained Systems, O'Reilly, 2015.

[2] T. Erl, Service-Oriented Architecture (SOA): Concepts, Technology and Design, Prentice Hall, 2005.

[3] P. Di Francesco, I. Malavolta, P. Lago, Research on architecting microservices: Trends, focus, and

potential for industrial adoption, in: 2017 IEEE International Conference on Software Architecture (ICSA), IEEE, 2017, pp. 21–30.

[4] I. Nadareishvili, R. Mitra, M. McLarty, M. Amundsen, Microservice Architecture: Aligning Principles, Practices, and Culture, O'Reilly, 2016.

[5] D. Taibi, V. Lenarduzzi, C. Pahl, Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation, IEEE Cloud Computing 4 (2017) 22–32. IEEE.

[6] J. Bogner, J. Fritzsch, S. Wagner, A. Zimmermann, Microservices in industry: Insights into technologies, characteristics, and software quality, in: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), IEEE, 2019, pp. 187–195.

[7] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina, Microservices: Yesterday, today, and tomorrow, in: Present and Ulterior Software Engineering, Springer, 2017, pp. 195–216.

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

[9] N. Kratzke, P.-C. Quint, Investigation of impacts on network performance in the advance of a microservice design, in: Cloud Computing and Services Science, Springer, Cham, 2017, pp. 187–208.

[10] D. Taibi, V. Lenarduzzi, On the definition of microservice bad smells, IEEE Software 35 (2018) 56–62. IEEE.

[11] A. Balalaie, A. Heydarnoori, P. Jamshidi, Migrating to cloud-native architectures using microservices: An experience report, in: Advances in Service-Oriented and Cloud Computing, Springer, Cham, 2016, pp. 201–215.

[12] D. D. Ruscio, I. Malavolta, H. Muccini, P. Pelliccione, A. Pierantonio, Developing next generation ADLs through MDE techniques, in: 2010 ACM/IEEE 32nd International Conference on Software Engineering, volume 1, IEEE, 2010, pp. 85–94.

[13] B. Combemale, R. B. France, J.-M. Jézéquel, B. Rumpe, J. Steel, D. Vojtisek, Engineering Modeling Languages: Turning Domain Knowledge into Tools, CRC Press, 2017.

[14] F. Rademacher, S. Sachweh, A. Zündorf, Aspect-oriented modeling of technology heterogeneity in Microservice Architecture, in: 2019 IEEE International Conference on Software Architecture (ICSA), IEEE, 2019, pp. 21–30.

[15] F. Rademacher, J. Sorgalla, P. Wizenty, S. Sachweh, A. Zündorf, Graphical and textual model-driven microservice development, in: Microservices: Science and Engineering, Springer, 2020, pp. 147–179.

[16] M. Nofer, P. Gomber, O. Hinz, D. Schiereck, Blockchain, Business & Information Systems Engineering 59 (2017) 183–187.

[17] C. Richardson, Microservices Patterns, Manning Publications, 2019.

[18] E. Evans, Domain-Driven Design, Addison-Wesley, 2004.

[19] F. Rademacher, S. Sachweh, A. Zündorf, 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), IEEE, 2020, pp. 229–236.

[20] ISO/IEC/IEEE, Systems and software engineering — Architecture description, Standard ISO/IEC/IEEE 42010:2011(E), 2011.

[21] E. Evans, Domain-Driven Design Reference, Dog Ear Publishing, 2015.

[22] R. T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, Ph.D. thesis, 2000.

[23] R. T. Fielding, J. F. Reschke, Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, RFC 7231, RFC Editor, 2014.

[24] O. Zimmermann, Microservices tenets, Computer Science - Research and Development 32 (2017) 301–310. Springer.

[25] M. Keeling, Design It!, Pragmatic Bookshelf, 2017.

[26] J. Sorgalla, P. Wizenty, F. Rademacher, S. Sachweh, A. Zündorf, Applying model-driven engineering to stimulate the adoption of devops processes in small and medium-sized development organizations (2021). arXiv:2107.12425.

[27] Duc Minh Le, Duc-Hanh Dang, Viet-Ha Nguyen, Domain-driven design using meta-attributes: A DSL-based approach, in: 2016 Eighth International Conference on Knowledge and Systems Engineering (KSE), IEEE, 2016, pp. 67–72.

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

[29] S. Kapferer, O. Zimmermann, Domain-driven service design, in: Service-Oriented Computing, Springer, Cham, 2020, pp. 189–208.

[30] N. Ferry, A. Rossini, F. Chauvel, B. Morin, A. Solberg, Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems, in: 2013 IEEE Sixth International Conference on Cloud Computing, IEEE, 2013, pp. 887–894.