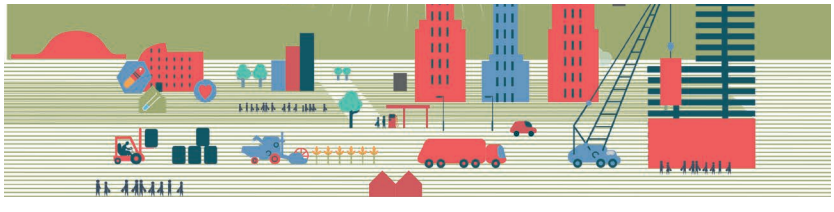
Challenges of Domain-Driven Microservice Design

A Model-Driven Perspective

Florian Rademacher, Jonas Sorgalla, and Sabine Sachweh,
Dortmund University of Applied Sciences and Arts

// This article explores several major challenges of domain-driven microservice design and presents ways to cope with them based on model-driven development. The article also provides an overview of supportive tools to practically address the challenges. //



DOMAIN-DRIVEN DESIGN (DDD) is a popular model-driven methodology for capturing domain knowledge relevant to software design.[1] To foster domain understanding and correctness of an emerging design, DDD emphasizes agile, collaborative modeling of domain experts and software engineers.

Currently, microservice architecture (MSA) is maturing as an architectural style for distributed software systems with high requirements for scalability and adaptability.[2] DDD gains additional relevance because it provides the means for decomposing domains into contexts, each clustering coherent domain concepts. These contexts correspond to functional microservices[3] that provide distinct business capabilities.

However, applying DDD to MSA poses several challenges concerning service deduction from domain models, modeling of infrastructure components, and domain modeling in autonomous teams. This article discusses these challenges from a model-driven perspective and presents ways to cope with them based on model-driven development (MDD).[4]

Running Example: The Cargo Domain Model

To illustrate the challenges of domain-driven microservice design, we use a cargo-related domain model[5] as a running example. It is depicted in Figure 1a as a UML class diagram (see the sidebar "Domain-Driven Design as a Modeling Means"). The diagram also includes the DDD patterns in the form listed in Table 1.

The domain model comprises three Bounded Contexts as UML packages, which denote starting points for three corresponding microservices. It is centered around the **Cargo** concept in the eponymous context. A **Cargo** holds various **Customers**, distinguished by roles like "shipper" or "receiver." This distinction is modeled as the qualified UML association **role** on **Cargo**. **Delivery History** keeps track of cargoes' **Handling Events**, possibly involving a **Carrier Movement** from a source **Location** to a target **Location**. A **Cargo** has a **goal**—i.e., a **Delivery Specification** with a destination **Location**.

0740-7459/18/$33.00 © 2018 IEEE

Challenges of Domain-Driven Microservice Design

This section elaborates on three major challenges of domain-driven microservice design, which impact the deduction of microservices from underspecified domain models, the modeling of MSA infrastructure components starting from domain models, and domain modeling across distributed autonomous microservice teams.

Deducing Microservices from Domain Models

To foster focusing on relevant concepts and efficient modeling, domain models typically omit information mandatory for deducing microservices and microservice characteristics like

- interfaces and operations;
- operation parameters and return types; and
- endpoints, protocols, and message formats.

Specific information about these characteristics is mandatory for service implementation. For example, the model in Figure 1a contains the Customer context without service interfaces and endpoints. However, these characteristics are necessary for implementing the corresponding microservice.

In domain models, associations between concepts of different contexts define exchange relationships for concept instances. When a context is implemented as a microservice, these relationships correspond to service interactions realizing instance exchange. In Figure 1a, a Cargo is associated with Customers. A Customer microservice thus needs to provide a Cargo microservice with Customer instances. However, the model does not specify which operations the service interface contains and how they may be invoked for instance retrieval. Indeed, the Customer Repository provides two operations that could be exposed to the Cargo microservice, but neither is clear if both operations are needed, nor do they specify return types. Concrete interface operations cannot be unambiguously deduced from the model.

An additional problem occurs when a cross-context association between two domain concepts is binary without explicit navigability specification—e.g., as for Delivery Specification and Location in Figure 1a. Without explicit navigability, instances may or may not be permitted to access instances at the other end of a UML association. Hence, the microservice responsible for requesting concept instances to establish the association—i.e., Cargo or Location—cannot be unambiguously determined.

Missing Infrastructure Components in Domain Models

Domain models intentionally do not comprise MSA infrastructure components.

The model in Figure 1a, for example, does not contain components for

- infrastructure provisioning—e.g., service discoveries or API gateways with stable interfaces for external requests—and
- service deployment—e.g., containers.

Additionally, domain expert requirements affecting technical aspects of the system must be documented separately from the domain model. Such a requirement could be that the Customer microservice needs to be externally accessible—e.g., for invoicing purposes. This requirement technically impacts the service: It needs to be externally discoverable and accessible, probably through an API gateway, and deployed to a container that forwards external requests.

Autonomous Domain Modeling

Due to high service cohesion and low coupling, microservice ownership is typically assigned to exactly one team.[2] The team is responsible for its services' implementation, operation, and design, including maintenance of corresponding Bounded

DOMAIN-DRIVEN DESIGN AS A MODELING MEANS

Domain-driven design abstracts domain concepts in models for software design. The domain models are typically sketched as UML class diagrams.[1] Classes, attributes, and methods describe domain concepts. Associations express concept relationships.

Domain models are usually underspecified; e.g., attributes possibly have no types, and methods may lack return types.[5] Underspecification fosters quick notation learning, model sketching, and focusing on relevant domain information. From a model-driven-design perspective, however, it is a major challenging factor of domain-driven microservice design as it introduces model informality.

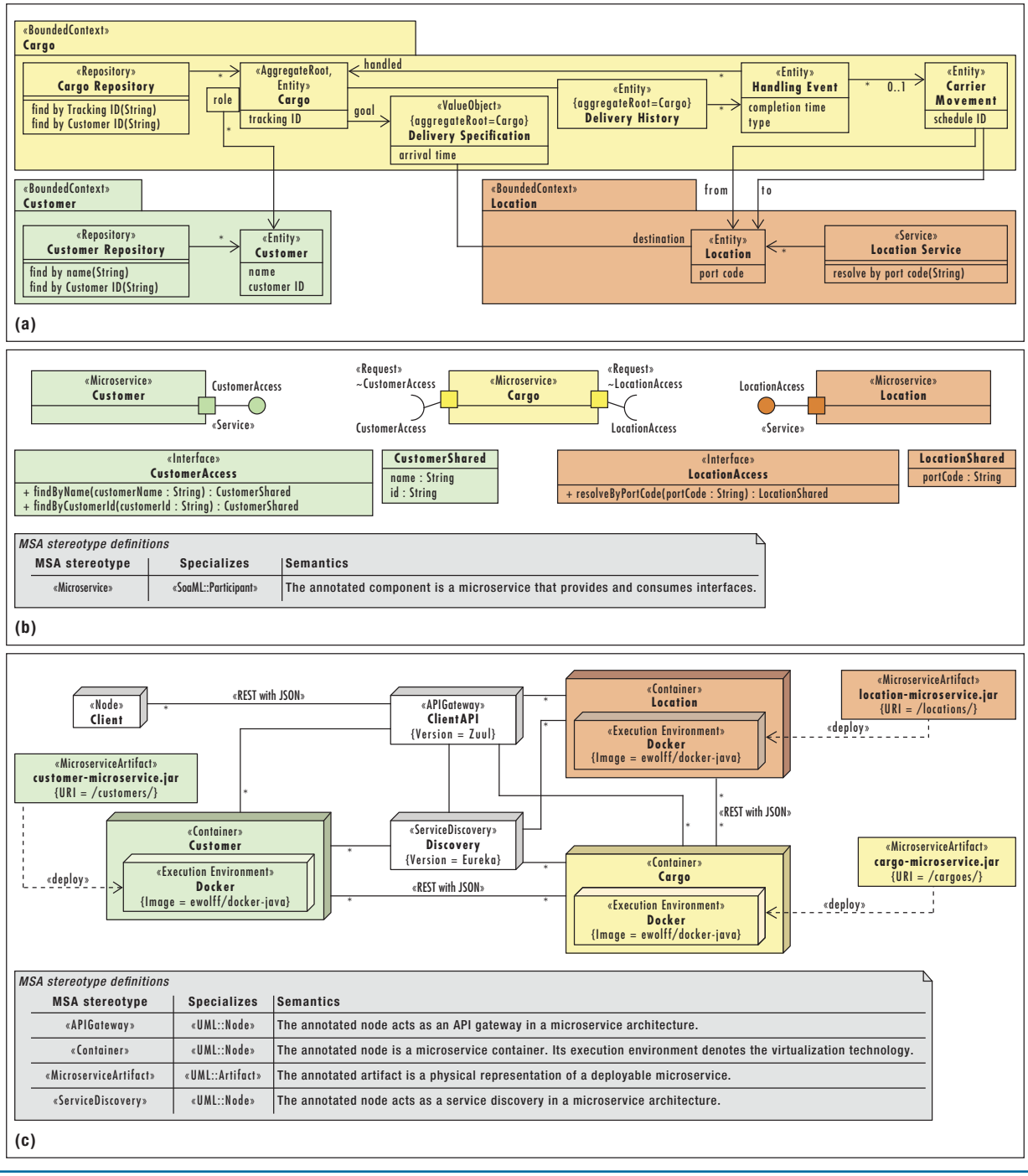
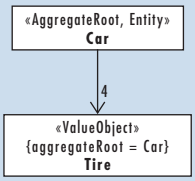
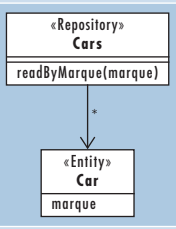
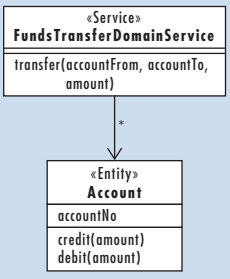
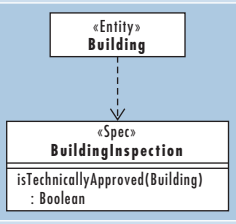
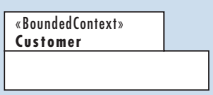


FIGURE 1. Applying domain-driven design (DDD) to microservice architecture (MSA). (a) The **Cargo** domain model (depicted as a UML class diagram with DDD-specific stereotypes).[1] (b) An intermediate model for the specification of microservice interfaces deduced from the domain model (modeled with the SoaML [Service Oriented Architecture Modeling Language] UML profile[7] and MSA-specific stereotypes). (c) An intermediate model, also deduced from the domain model, for the specification of microservice deployment (a UML deployment diagram with MSA-specific stereotypes).

Table 1. Domain-driven-design patterns for domain-driven microservice design, in UML notation.[5]

Pattern name	Example	Description
Entity		Instances of domain concepts modeled as Entities are distinguishable from others by a domain-specific identity.
Value Object		Instances of domain concepts modeled as Value Objects are typically immutable and lack a domain-specific identity. They might act as value containers when exchanging information between Bounded Contexts (see below).
Aggregate		An Aggregate is a cluster of associated Entity and Value Object instances. It is treated as a whole that can be accessed only by referencing its root Entity instance. Next to Bounded Contexts, Aggregates might also denote a primary driver for domain decomposition.[6]
Repository		A Repository permits access to persistent domain concept instances via operations that perform instance selection based on given search criteria.
Service		Services provide capabilities that are not the responsibility of Entities or Value Objects—e.g., control of business processes or domain concept instance transformations. A special type of domain-driven-design service is domain services. They interact with domain concept instances to realize business capabilities. For example, a funds transfer domain service might be responsible for coordinating credits and debits between banking accounts.[1] Services are parts of Bounded Contexts and are not to be confused with microservices that are derived from Bounded Contexts.
Specification		Specifications may be employed to determine if a domain concept instance fulfills a certain specification. Specification classes contain a set of Boolean operations to perform specification checks.
Bounded Context		Bounded Contexts define scopes for enclosed domain concepts—i.e., boundaries for concept validity and applicability. A functional microservice[3] should be aligned to a Bounded Context because, like a microservice, a context bundles and isolates coherent domain concepts and thus • defines the scope of an isolated business capability, • needs to explicitly define access to domain concept instances via exchange relationships to other contexts, and • has exactly one responsible team assigned.[2]

Contexts. So, challenges arise regarding domain model access and change management.

First, it has to be decided which domain model parts are visible to teams. One approach is to make the domain model completely accessible. Alternatively, teams might have insight into only their Bounded Contexts and associated concepts. Depending on the approach, organizational efforts and possible model changes differ.

Teams might be permitted to change other teams' Bounded Contexts. They would have to take into account that resolving and merging changes will be more difficult and that new service dependencies will be more likely.

Means of Model-Driven Development to Cope with Challenges

In the following, we present three distinct means of MDD, each of which addresses one of the challenges of domain-driven microservice design described in the section "Challenges of Domain-Driven Microservice Design."

Intermediate Models

Intermediate models show technical microservice characteristics from relevant viewpoints. A domain model should be transferred into intermediate models specifying microservice interfaces and deployment.

Figure 1b shows an interface model of the cargo system deduced from the domain model in Figure 1a. Simple Interfaces of the Service Oriented Architecture Modeling Language (SoaML) are used to concisely model provided and required service interfaces.[7] The «Microservice» stereotype identifies microservices. It specializes SoaML's «Participant» stereotype for generic service components. For each context in a domain model, the interface model contains an eponymous microservice.

SoaML expresses provided service interfaces as ports with outgoing UML lollipops—i.e., solid lines starting from the port and ended by a circle. Each lollipop exhibits the «Service» stereotype and interface name. A microservice provides an interface when at least one of the underlying Bounded Context's domain concepts is associated and navigable from another context. For example, the Customer microservice in Figure 1b provides a CustomerAccess interface because the Customer Entity in Figure 1a is associated with and navigable from the Cargo Entity in the Cargo context.

Required service interfaces are modeled as requesting ports with the «Request» stereotype and interface name in SoaML. A microservice requires an interface when its context's concepts reference concepts of other contexts in a navigable way. For example, the Cargo microservice in Figure 1b has a requesting port CustomerAccess to consume the eponymous interface of the Customer microservice. It enables the Cargo service to associate Cargo Entity instances with Customers as modeled in Figure 1a.

Figure 1c shows an intermediate UML deployment diagram deduced from the Cargo domain model (see Figure 1a). «MicroserviceArtifact» specializes UML's «Artifact» stereotype. «MicroserviceArtifact» allows explicit modeling of deployable microservice artifacts. The URI attribute specifies a deployed service's location. Each Bounded Context of a domain model is transferred to a «MicroserviceArtifact» in the deployment model. Each artifact is deployed to a container identified by the «Container» stereotype. Container execution environments specify the employed container technologies and images—e.g., Docker in Figure 1c.

Containers have a communication path when an association exists between concepts of the deployed microservices' Bounded Contexts. Communication paths specify protocols and message formats for container communication and correspond to service interactions in the interface model, which also differentiates service provider and requester roles. In Figure 1c, a communication path is modeled between the Cargo and Customer containers because the Cargo and Customer Entities are associated in Figure 1a. So, deployed microservices communicate via REST (Representational State Transfer) and JSON (JavaScript Object Notation). Message-based communication could be modeled by specifying a protocol like AMQP (Advanced Message Queuing Protocol) on a communication path.

Additionally, the deployment model contains infrastructure components—i.e., service discovery and an API gateway.

The intermediate models introduce technical characteristics that intentionally are not in the domain model but substantiate MSA implementation. The interface model shows deduced microservices, interfaces and operations with types, and service dependencies. The deployment model comprises technical information regarding service endpoints, protocols, and message formats; deployment technologies; and infrastructure components. The model enables deduction of container configurations and deployment descriptors.

Reducing Domain Model Informality

Domain model informality can significantly be reduced with modeling conventions.

One convention might be the prohibition of binary associations[1]—e.g., in Figure 1a, between **Delivery Specification** and **Location**. This convention encourages modelers to clarify which microservice is responsible for requesting concept instances from another microservice leveraging navigability specifications (see the section "Deducing Microservices from Domain Models"). If this convention is too strict, modelers should consider labeling the providing side of a binary association. For example, the modeling decision in Figure 1b that the **Cargo** requests the **Location** microservice is based on the named end of the **Delivery Specification** and **Location** association in Figure 1a.

In addition, modelers should agree on the DDD patterns that lead to microservice interfaces in advance of the modeling process. A convention might be that Repositories and Services result in interfaces when their maintained Entities are referenced from external contexts. For referenced concepts not maintained by Repositories or Services, another convention should be established—e.g., deducing default interface operations. Furthermore, the return types of methods should be explicitly modeled if they cannot be deduced from the domain model. For example, if a method is not part of a Repository or Service in which its return type is typically the maintained Entity type, unambiguous return type deduction might be impossible. An additional convention could determine that domain model methods with public UML visibility result in interface operations.

Model informality might additionally be reduced by defining shared models.[2] As a convention, the structures of shared models in Figure 1b correspond directly to the **Customer** and **Location** Entities in Figure 1a. However, in certain situations it is sensible to model shared models with reduced structures—e.g., to lower network traffic by transmitting fewer attributes. Shared models can be modeled as classes referenced from requesting contexts and dependent on concepts in providing contexts.[5] They might be directly added to interface models as operation result types.

Policies for Autonomous Domain Modeling

Modeling policies for regulating domain model access and permissible changes might be employed for distributed, autonomous service teams.

Giving all teams complete insight into the domain model—i.e., by providing access to the file that physically holds the model via shared storage—increases the set of domain concepts that may be affected by changes. A more restrictive policy is to make only coherent parts of the model accessible. First, each team needs to have insight into its services' Bounded Contexts. Second, it needs access to associated concepts from other contexts. For example, the team that implements the **Cargo** microservice of Figure 1a needs to know the structures of the **Customer** and **Location** Entities. In contrast, the **Customer** team needs access to only its Bounded Context as it does not reference other contexts' concepts in a navigable way (see the section "Intermediate Models").

Partial model access fosters low service coupling as each team has insight into only its contexts and a minimum of context-external concepts. However, it requires additional effort to split the model file into several files containing only team-relevant contexts and concepts. Independent of the applied model access approach, explicit model management is necessary; i.e., a central instance needs to keep track of changes from distributed teams and incorporate them into a unified domain model.

A policy regarding permissible changes of Bounded Contexts or domain concepts of other teams should be employed. Allowing such changes is sensible in situations where a domain model is incomplete, to enable its collaborative completion. Considering Figure 1a, a sensible addition of the **Cargo** team could be **latitude** and **longitude** fields in the **Location** Entity to extend a **Carrier Movement** with geolocation information.

However, allowing teams to change other teams' contexts needs careful consideration. First, changes might need to be resolved by involving those teams that changed the model, are responsible for the affected context and microservice, and realize dependent services. Second, changes may introduce new service dependencies—e.g., by associating concepts from different contexts.

For example, in Figure 1a the **Cargo** team might associate the **Customer** and **Location** Entities from the eponymous contexts to add address information to **Customers**. The **Customer** microservice then would newly require information from the **Location** microservice. Hence, the **Customer** and **Location** teams would be affected by a change introduced by the **Cargo** team.

Tools for Practical Domain-Driven Microservice Design

Tool support is crucial for practical MDD.[4] Currently, few MDD tools exist for domain-driven microservice design.

ABOUT THE AUTHORS

FLORIAN RADEMACHER is a PhD student and research associate at the Institute for the Digital Transformation of Application and Living Domains (IDiAL) at the Dortmund University of Applied Sciences and Arts. His research interests include model-driven development, specifically the design and implementation of domain-specific modeling languages, and service-based architectures, with a focus on microservice architecture. Rademacher received an MS in applied computer science from the Dortmund University of Applied Sciences and Arts. Contact him at florian.rademacher@fh-dortmund.de.

JONAS SORGALLA is a research associate at the Institute for the Digital Transformation of Application and Living Domains (IDiAL) at the Dortmund University of Applied Sciences and Arts. His primary research interest is model-driven development, with a focus on participatory design of domain-specific modeling languages. Sorgalla received an MS in applied computer science from the Dortmund University of Applied Sciences and Arts. Contact him at jonas.sorgalla@fh-dortmund.de.

SABINE SACHWEH is professor of software engineering at the Dortmund University of Applied Sciences and Arts. She's also a spokesperson for the university's Institute for the Digital Transformation of Application and Living Domains (IDiAL) and the head of the institute's Smart Environments Engineering Laboratory. Her research interests include social platforms and computer-supported communication for groups of socially underprivileged persons, advanced interaction techniques, and the challenges of digital transformation in industrial contexts. Sachweh received her PhD in computer science from Paderborn University. Contact her at sabine.sachweh@fh-dortmund.de.

For creating DDD-based domain models like Figure 1a, a UML profile exists. It comprises stereotypes for DDD patterns (see Table 1). Modelers can turn UML class diagrams into domain models by applying the stereotypes to existing elements. The profile's constraints for stereotype application following DDD semantics[5] aid in reducing domain model informality (see the section "Reducing Domain Model Informality"). The profile is developed with the Eclipse Papyrus modeling environment and can be applied to existing Papyrus-based UML class diagrams. The resulting domain model file may be distributed across microservice teams as described in the section "Policies for Autonomous Domain Modeling." The DDD profile, together with concrete usage instructions and examples, can be found at github.com/SeelabFhdo/ddmm-uml-profile.

AjiL is an Eclipse-based tool for graphical MSA modeling. It features a diagram editor for modeling rudimentary domain models, functional and infrastructure microservices, and provided and required interfaces. Thus, AjiL facilitates creating intermediate models for service interfaces (see the section "Intermediate Models"). AjiL ships with a code generator that produces microservice implementations from AjiL models. The generated Java code is based on Spring Boot and Spring Cloud. It is directly runnable and comprises interface implementations, stubs for implementing services' business logic, and configuration files for infrastructure components. However, AjiL currently does not support round-trip engineering.[4] The tool is available at github.com/SeelabFhdo/AjiL.

The profile and AjiL are actively used in industry-related research projects. Current development efforts focus on integrating both tools with the aim to automate the deduction of intermediate models from domain models. Therefore, profile-based domain models are to be automatically transformed into AjiL models for subsequent technology-related refinements and code generation.

Domain-driven design provides, among other things, the modeling means for software design to decompose domains into bounded contexts. Thereby, each context clusters coherent domain concepts and denotes a

candidate for a microservice. However, domain models are typically underspecified, which poses several challenges when applying DDD to MSA. In this article we discussed challenges in deducing microservices and related infrastructure components from domain models, as well as domain modeling in microservice teams. We then presented selected means and tools of MDD to cope with such challenges.

References

1. E. Evans, *Domain-Driven Design*, Addison-Wesley, 2004.
2. S. Newman, *Building Microservices: Designing Fine-Grained Systems*, O'Reilly Media, 2015.
3. M. Richards, *Microservices vs. Service-Oriented Architecture*, O'Reilly Media, 2015.
4. B. Selic, "The Pragmatics of Model-Driven Development," *IEEE Software*, vol. 20, no. 5, 2003, pp. 19–25.
5. F. Rademacher, S. Sachweh, and A. Zündorf, "Towards a UML Profile for Domain-Driven Design of Microservice Architectures," *Software Engineering and Formal Methods*, 2018, Springer, pp. 230–245.
6. C. Richardson, "Developing Transactional Microservices Using Aggregates, Event Sourcing and CQRS—Part 1," InfoQ, 3 Oct. 2016; www.infoq.com/articles/microservices-aggregates-events-cqrs-part-1-richardson.
7. *Service Oriented Architecture Modeling Language (SoaML) Specification Version 1.0.1*, Object Management Group, 2012; www.omg.org/spec/SoaML.