Pattern-based Modelling, Integration, and Deployment of Microservice Architectures

Vladimir Yussupov[1], Uwe Breitenbücher[1], Christoph Krieger[1],
Frank Leymann[1], Jacopo Soldani[2], and Michael Wurster[1]
[1]Institute of Architecture of Application Systems, University of Stuttgart, Germany
Email: [lastname]@iaas.uni-stuttgart.de
[2]Department of Computer Science, University of Pisa, Italy
Email: soldani@di.unipi.it

***Abstract*—Microservice-based architectures (MSAs) gained momentum in industrial and research communities since finer-grained and more independent components foster reuse and reduce time to market. However, to come from the design of MSAs to running applications, substantial knowledge and technology-specific expertise in the deployment and integration of microservices is needed. In this paper, we propose a model-driven and pattern-based approach for composing microservices, which facilitates the transition from architectural models to running deployments. Using a unified modelling for MSAs, including both their integration based on Enterprise Integration Patterns (EIPs) and deployment aspects, our approach enables automatically generating the artefacts for deploying microservice compositions. This helps abstracting away the underlying infrastructure including container orchestration platforms and middleware layer for service integration. To validate the feasibility of our approach, we illustrate its prototypical implementation, with Kubernetes used as container orchestration system and OpenFaaS used for managing integration logic, and we present a case study.**

***Index Terms*—Microservice Architecture, Service Composition, Enterprise Integration Pattern, Model-driven Engineering**

I. Introduction

Microservice architectures (MSA) gained a lot of attention from both industry and academia [1]. In this architectural style, a single application is composed of independent and fine-grained services, each running in its own process and communicating using lightweight protocols such as HTTP or MQTT [1], [2]. Microservices are intended to be simple and focus on accomplishing one task well. Additionally, due to a smaller scale, each service can be built using the most appropriate tool for the job, e.g., based on the respective team's programming language preferences. Furthermore, MSAs natively enable continuous delivery allowing frequent releases and fast feedback loops, which fits perfectly into an agile development setup where cross-functional teams are responsible for a certain service over its full lifetime [3].

At the same time, MSAs come with some drawbacks, and also inherit some common pitfalls of traditional distributed systems [4]. For example, the higher number of units to manage makes it more complex to compose and operate MSA-based applications. For integrating services, developers have to deal with various heterogeneous problems, e.g., asynchrony, cascades of failures, and incompatible data models [3]. In addition, developing services in various programming languages may lead to duplication of efforts for solving identical issues, e.g., implementation of common integration patterns for message routing and transformation. Moreover, services often have different load patterns and must be deployed and operated using diverse and specialised configurations [5].

Not surprisingly, transforming the design of a MSA into a running system requires substantial knowledge of concepts in the fields of service deployment and integration. Developers are indeed required to devise solutions for deploying the microservices constituting an application, e.g., specifying their actual deployment on a container orchestration system such as Kubernetes. They are also required to design and develop an integration actually enabling service-to-service communication, e.g., using *Enterprise Integration Patterns (EIP)* [6] and message-oriented middleware. This obviously results in a time-consuming, cumbersome and error-prone process, which requires developers to gain deep technical expertises on all involved technologies and on the languages for programming and configuring such technologies.

With the aim of easing the design and deployment of MSAs, in this paper we propose the *MIcroservices COmposition (MICO) approach*. MICO unifies and synergically combines the architectural, integration, and deployment aspects of MSA-based applications, enabling a "one-click" transition from modelled integration of microservices to running deployments. The main contributions of this paper are:

1) We propose a unified meta-model that synergically combines architecture, integration, and deployment of microservices, based on the pipes and filters architectural style, and with integration patterns as first-class citizens.
2) We present the MICO approach that enables transparent transition from integration models to running deployments by generating the required deployment artefacts.
3) We introduce a technology-agnostic system enabling our approach, and its prototypical implementation. We also show how we exploited such implementation to conduct a concrete case study based on a third-party application.

Notably, the overall MICO approach includes automatic import of user-provided source code repositories into the system as runnable building blocks to be used in a microservice composition model. Furthermore, reusable and configurable

2325-6362/20/$31.00 ©2020 IEEE
DOI 10.1109/EDOC49727.2020.00015

implementations of common enterprise integration patterns can be used for modelling desired microservice compositions. Resulting application models can then be automatically transformed by the approach-enabling system into a deployable set of source code and configuration files that are transparently handled by the underlying infrastructure layer.

The remainder of the paper is organised as follows. Sections II and III describe the background and motivating scenario. Sections IV and V introduce the pattern-based microservices composition meta-model and our approach. Section VI discusses the approach-enabling system architecture and its prototype, and Section VII presents a case study. Finally, Sections VIII and IX discuss related work and the approach generalisation directions, and Section X concludes the paper.

II. Background

This section provides the necessary background information on MSAs, container orchestration, and integration patterns.

A. Microservices and Container Management

The notion of a *microservice* [1], [2] serves as a core building block of MSAs in which applications are developed as a set of small, single-purpose and loosely-coupled components interacting with each other using lightweight communication mechanisms such as HTTP-based REST APIs. While microservices are closely-related to service-oriented architectures (SOA), the idea to introduce proper bounded contexts that share as little as possible makes this architectural style different from SOA where sharing among components is not discouraged [7]. As a result, having small bounded contexts size fosters the reusability and allows shipping microservices independently of each other [8]. One of the core enablers facilitating microservice-based application development is the lightweight, container-based virtualization, e.g., using Docker containers [9]. The core idea is to ship microservices in containers to remove the dependency on a particular technology or platform, hence simplifying the delivery process. Since the number of microservices in a typical component architecture might get large, integration and management of multiple running containers becomes a complex task to accomplish.

Container orchestration systems aim to simplify management of container-based applications in clustered environments [10]. For instance, managing multiple manually-deployed containers would require monitoring their availability and restarting faulted containers, applying scaling rules, and managing inter-container communication. Docker Swarm and Kubernetes are prominent examples of container orchestration systems. For example, Kubernetes simplifies container management features such as autoscaling, self-healing, resource monitoring, and deployment automation using declarative deployment models [11] describing the desired state for all containers in the given application. Furthermore, Kubernetes serves as a basis for a rich ecosystem of complementary software, e.g., service meshes such as Istio or Function-as-a-Service (FaaS) platforms like OpenFaaS and Kubeless.

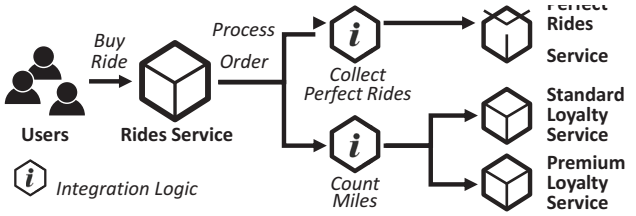

Fig. 1. Example microservice architecture with service integration points.

B. Enterprise Integration Patterns

A *pattern* describes a proven solution for a particular problem that reoccurs frequently in a certain context [12]. Typically, patterns are documented in an abstract way and follow a well-defined structure comprising a pattern's name, a problem description, details about the context in which they can be applied, and a proven solution. Enterprise Integration Patterns (EIPs) [6] describe well-known, proven solutions facilitating the integration of disparate enterprise systems using messaging. The documented patterns explain multiple integration concepts, from messaging systems' basics to specific aspects such as message construction or routing techniques.

One important example of the basic concept related to messaging systems is the *Pipes and Filters* pattern, which describes an architectural style in which a large task is subdivided into a set of small interconnected tasks (called *filters*) connected by channels (called *pipes*). Examples of specific patterns are grouped in categories such as messaging channel types, e.g., *Point-to-Point* or *Publish-Subscribe* channels relying on queues and topics respectively. Another large group of patterns describes message routing, e.g., *Content-based Router* or *Message Filter* patterns. The former describes how to route messages to different consumers based on their content type, whereas the latter describes the solution for filtering by message's type. Furthermore, when solving integration challenges, messages can be split as described by the *Splitter* pattern or aggregated using *Aggregator* pattern. Chaining aforementioned patterns results in composite solutions such as the *Scatter-Gather* pattern. Combining arbitrary number of patterns using pipes-and-filters provides a flexible way of defining integration of business logic components [6].

Another powerful property of integrating components using EIPs is the increased readability of complex combinations of patterns that represent desired integration solutions. Moreover, messaging patterns align well with the concept of event-driven computing, e.g., an *Event Message* pattern can be combined with other message routing and processing patterns to implement even-driven task pipelines.

III. Motivating Scenario

Consider the rides management application that is commonly used in Amazon's hands-on workshops [13]. Figure 1 illustrates an excerpt of such application, which consists of four microservices, namely *Rides Service*, *Perfect Rides Service*, *Standard Loyalty Service*, and *Premium Loyalty Service*. The

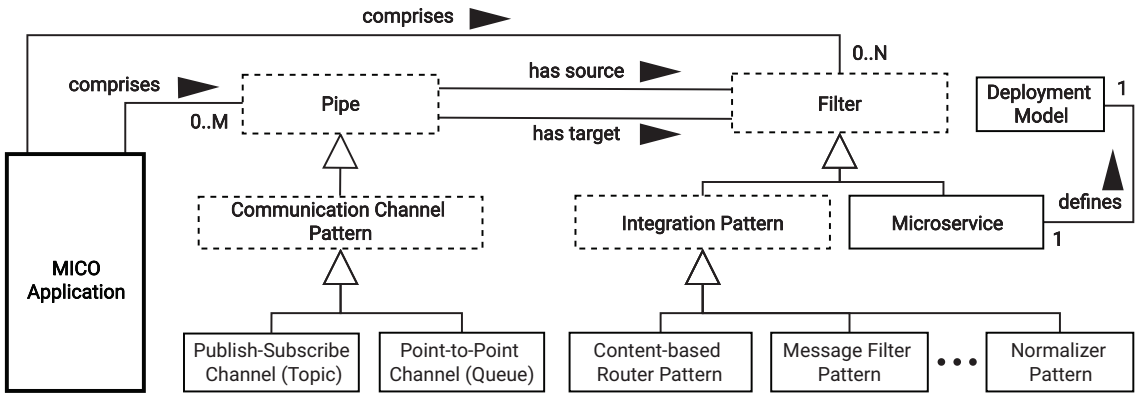

Fig. 2. The pattern-based microservice composition (MICO) meta-model (abstract types are drawn with the dashed line).

overall interaction flow requires solving two integration tasks relying on ride order notifications originating from the Rides Service. The first integration task consists in filtering order notifications and select only perfect rides for processing them using the Perfect Rides Service, e.g., rides with the user-given rating higher than usual. The second integration task is to check the customer's subscription type in the ride order and process the purchased travelling miles using the corresponding loyalty service, i.e., the Standard or Premium Loyalty Service. Even if simple, such an example illustrates how crucial service integration is for suitably enabling the composition and deployment of microservices as a whole application.

There already exist different technologies facilitating deployment and integration aspects. For example, container orchestration systems such as Kubernetes can be used to automatically deploy and manage microservices. Integration frameworks such as Apache Camel support the integration of services using well-known enterprise integration patterns. However, these solutions are either focused on the deployment or integration of services. Moreover, such technologies require deep technical expertise about the applied concepts and used programming or configuration languages. For example, to deploy our example using Kubernetes, one needs to understand Kubernetes concepts such as *Pods*, *Deployments*, and *Services*. Furthermore, developers need to understand the chosen messaging technology to integrate the services using implemented patterns, e.g., *Filter* pattern and *Content-based Router* pattern for the first and second integration tasks respectively.

The discussed issues, hence, raise the following research question: *How to bridge the gaps among architectural models, integration, and deployment of microservice compositions by also abstracting from technology-specific details to reduce the entry-level requirements?* To answer this question, we introduce a model-driven and pattern-based approach, which combines the architectural, integration, and deployment views of microservices. We also show how this enables automatically generating the source code, configuration files, and deployment files that are needed for the transparent deployment of the modelled microservice compositions.

IV. MICO Meta-Model: Composing Microservices using Enterprise Integration Patterns

In this section, we introduce the pattern-based composition meta-model our approach relies on, and briefly describe how the motivating scenario described in Section III can be expressed using this meta-model.

A. MICO Meta-model

The core part of the MICO approach is the microservice composition meta-model shown in Figure 2, which enables describing how microservices need to be integrated using well-known integration patterns, together with the abstracted away deployment information that enables transparent transition from the model to running deployments. The overall idea of the MICO meta-model derives from the pipes-and-filters architectural style, a well-documented enterprise integration pattern [6]. In this architectural style, processing components (i.e., the *filters*) are connected together by means of channels (i.e., the *pipes*). Modelled integrations of microservices, e.g., the motivating example shown in Figure 1, can also be expressed as a composition of filters interacting by means of different types of communication channels.

To combine such view on the integration with the deployment information, the first-class citizens of a *MICO Application* are the *Filters* that represent the application's building blocks, and the *Pipes* that specify required communication channels. The filters are further divided into two possible subtypes, namely *Microservices* and *Integration Patterns*. Microservices represent components that implement business logic, whereas Integration Patterns serve for tackling microservices integration tasks, e.g., the *Collect Perfect Rides* and *Count Miles* in Figure 1. In general, patterns represent custom integration logic that is bound to a particular integration problem, e.g., message routing. To facilitate the structuring and reuse of integration patterns, the MICO meta-model provides an explicit type system representing implementations of enterprise integration patterns [6], e.g., Message Filter, Content-based Router, or Normalizer patterns.

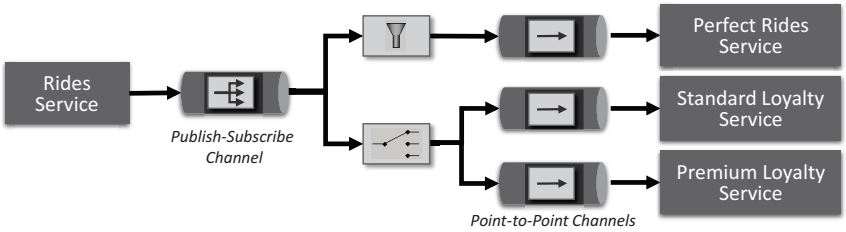

Fig. 3. An instance of the MICO meta-model depicting the integration for the motivating scenario discussed in Section III.

For integrating microservices, the MICO meta-model already supports modelling two kinds of pipes, i.e., topic-based and queue-based communication channels, which represent *Publish-Subscribe* and *Point-to-Point* communication patterns, respectively. These options provide a flexible way of composing available microservices. The aforementioned Integration Pattern entities are used specifically for modelling message-based communication in which available business logic components are interconnected via an Integration Pattern and matching output and input *Topics* or *Queues*. Additionally, Microservice entities define the deployment model that includes the information required for deploying the microservice with its underlying sub-components such as databases, and additional meta-data such as owner details. After specifying the required data, resulting integration models can be processed to derive a set of deployment artefacts for transparent deployment of the application instances to the used infrastructure.

B. Exploiting the Meta-Model in our Motivating Example

An instance of the MICO meta-model shown in Figure 3 depicts one possible solution for the integration tasks presented in the motivating scenario from Section III. As described in Section IV-A, the entire integration is modelled as a set of filters connected by means of pipes, i.e., the Microservices responsible for processing rides data integrated by means of two Integration Patterns using Point-to-Point and Publish-Subscribe channels. For example, the "Collect Perfect Rides" service integration task in Figure 1 is modelled by the following interconnection: the Rides Service's output topic needs to be connected with the Integration Pattern that processes produced messages by selecting only rides that have high rating, and passes the resulting messages to Perfect Rides Service's input queue. The Integration Pattern in this case is a specifically-configured instance of the Message Filter pattern. The "Count Miles" integration task in Figure 1 is instead modelled as an interconnection of the Rides Service's output topic with the specifically-configured Content-based Router as Integration Pattern which routes the rides data to the suitable Loyalty Service's input queue, i.e., Standard or Premium, depending on the message's data.

In the following sections, we elaborate on how such a high-level architectural view on integration of aforementioned microservices can be used to enable the automatic generation of all corresponding artefacts needed for enacting the deployment of modelled MICO applications.

V. MICO Approach: From Models to Deployments

In this section, we present the MICO approach that relies on the unified model described in Section IV, and which enables the transition from the modelled pattern-based microservice integrations to running deployments without the need to manually configure the underlying technologies. Under the hood, modelled microservice integrations are automatically transformed into running deployments using the dedicated MICO software layer hosted on top of the chosen infrastructure, e.g., container orchestration platforms such as Kubernetes.

A. Overview

Figure 4 shows a general view on the process of pattern-based composition of microservice applications using the MICO meta-model discussed in Section IV. The MICO approach consists of four steps, i.e., (1) importing microservices as application's building blocks, (2) modelling pattern-based integrations, (3) generating deployment artefacts, and (4) deploying application instances. For simplicity, the steps are presented in sequence, even if the transition between steps is intended to be agile, e.g., allowing to returning from the deployment phase back to importing components, then modifying the model and redeploying the updated application.

Before starting to compose microservices, modellers have to import the application's components into the system first, to be able to use them as building blocks in the desired integration model. Examples of sources for components import can be code hosting platforms such as GitHub or Bitbucket, or local code repositories versioned using Git or Subversion. After the import, the specified source code repository is converted in a format that can be deployed to the chosen infrastructure layer, e.g., a deployable container image, which can be used as a part of the MICO application.

After importing required microservices, modellers are able to represent desired integrations, which define how application components are connected together and provide the deployment details required for each involved component in a form of deployment models to support transparent transformation into running applications. Using topic- and queue-based communication channels, modellers also define desired integrations, which can then be transformed into a set of artefacts required by the underlying infrastructure such as container orchestration platform to instantiate the model as a running deployment.

The saved application models can easily be modified, e.g., by adding new components, updating existing components'

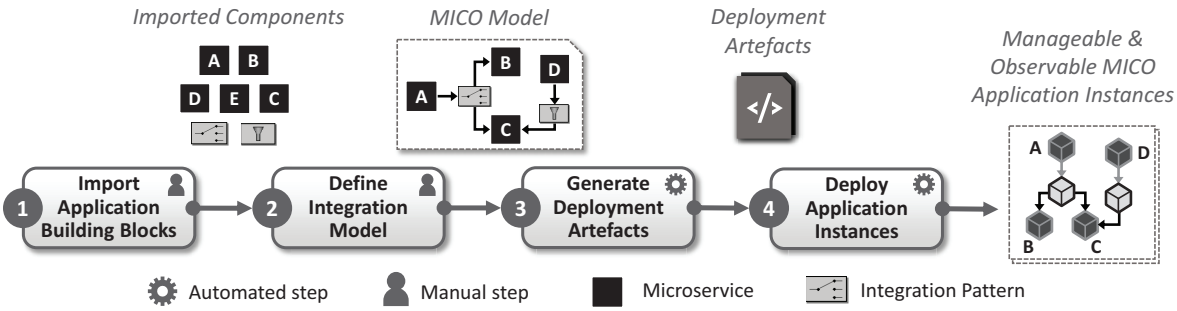

Fig. 4. Overview of the pattern-based microservice composition approach.

versions, or defining new compositions between components in existing models. Modified models can then be translated into deployments without requiring to manually specify any information on the underlying infrastructure.

B. Step 1: Import Application Building Blocks

As discussed in Section IV, the main building blocks for creating microservice compositions in MICO are *Filters* of different kinds. On the technical level, a Filter of type Microservice can represent, for example, a container image together with a deployment model providing details on how to deploy it. On the other hand, the implementation of the Integration Pattern filter type entirely depends on the MICO-enabling software layer. For example, the platform might provide a set of default, configurable implementations for common integration patterns, hence enabling transparent integration of chosen business logic components. On the other hand, a pattern implementation might also be a custom software component imported into the software layer in a similar way as business logic components. MICO's Filters can be imported and stored in a dedicated service repository maintained in a MICO software layer to be available for use within a MICO application. The reasons why such repository is needed are twofold, namely (i) to have a common format for storing application building blocks, i.e., local or published Docker images which have to be generated if the source code is provided, and (ii) to simplify reusing components.

For importing filters of type microservice in particular, we envision a *source-to-image* workflow. This means that the Microservice can simply be imported in a runnable format by specifying the URL to its code repository containing the source code of the microservice and a corresponding deployment model containing the instructions, e.g., for building the container image using the Dockerfile and deploying it with its related dependencies. Based on the provided sources, the respective container image is built and staged into an image repository. In addition, users can provide a service specification describing complementary details about the service such as the messages produced and consumed by the service, e.g., by defining the structure of a message and the type of content of each data element within the message.

To ensure a common message format, we envision MICO-enabling platforms to rely on CloudEvents[1], a vendor-neutral specification of event data. CloudEvents specification standardises a set of event's metadata to facilitate proper routing and processing of the messages, including such details as the (i) unique identifier, (ii) source, (iii) type, and (iv) schema the event data adheres to. The event data of the message is encapsulated within the data attribute and follow the defined schema. This results in two possible types of services to import, namely (i) *MICO-enabled microservices* that comply with MICO development requirements, i.e., CloudEvents-based message format, and (ii) *regular microservices* that can still be imported, but will need to be composed using Integration Patterns that implement required message transformation patterns such as Normalizer pattern [6]. Hence, Integration Patterns allow flexibly mixing existing business logic with MICO-enabled microservices.

C. Step 2: Define Integration Model

Microservices available in the service repository can then be used as part of *MICO Models*, representing the desired integration of microservices. As described in Section IV, to create microservice compositions in which services are loosely coupled and highly scalable, developers are able to define messaging-based communication channels using generic *Integration Patterns*, which are based on the Enterprise Integration Patterns [6] and implement functionalities such as splitting, aggregating, routing, and transformation of messages.

Essentially, an Integration Service reads messages from a specific topic or queue, modifies the message according to the provided integration logic and routes the resulting message to one or multiple output queues or topics. Examples of integration logic include enrichment, splitting, or filtering of messages. For example, to model an integration from *microservice A* to *microservice B*, one firstly needs to specify an output communication channel such as topic or queue for microservice A, which is then used as an input communication channel for microservice B. Afterwards, the modelled channels have to be associated with the desired Integration Pattern, e.g., routing or filtering messages produced by service A and

[1]https://cloudevents.io

sending them to the input channel of service B. This pipes-and-filters-based modelling style makes integration models very flexible, easily allowing to introduce new or reuse existing Integration Patterns in-between for accomplishing the given integration task. After connecting required Integration Patterns, modellers need to configure the actual integration logic, which may be implemented differently on the technical level.

Basically, the pattern implementation is coupled with the message formats produced or consumed by the connected microservices. Hence, the overall configuration process depends on the underlying MICO software layer implementation. For example, modellers can directly specify the integration logic on the level of the model, which will be transformed into the required deployment format, e.g., hosted on the Function-as-a-Service platform and bound to the required messaging events. Another option is to import Integration Pattern implementations similar to how microservices are imported. While the integration logic can be stored together with the microservices, it is also possible to have a dedicated integration logic management component, e.g., integration container images repository.

D. Steps 3: Generate Target Deployments

To achieve a "one-click" transition from the modelling of an application to its running deployment, a created MICO Model can be automatically deployed to an underlying infrastructure such as container orchestration platform. The deployment of a microservice composition using the MICO software layer consists of three main steps, i.e., (i) deployment of the messaging infrastructure by creating the message queues and topics needed for message-based communication, (ii) deployment of Integration Patterns implementing integration functionalities, and (iii) deployment of MICO Filters implementing the business logic. Typically, this process involves translation of the data in MICO Models into a required set of deployment actions for every component in the model. For example, this may involve executing the provided declarative deployment model and a specific sequence of API calls or complementary commands, e.g., establishing the modelled communication channels. Hence, the concrete execution of these steps is infrastructure-specific, e.g., with respect to the employed container orchestration platform, message-oriented middleware, and the chosen software for hosting and managing the integration pattern implementations. As a result, to generate the target deployment logic for underlying infrastructure requires having dedicated transformation solutions, e.g., generating Kubernetes deployments, creating Apache Kafka topics, and deploying integration logic to a dedicated container repository.

E. Step 4: Deploy Application Instances

After the deployment artefacts for the target infrastructure are generated, the deployment can be executed by the MICO-enabling platform. As additional features, the system should also provide capabilities for management, observability, and testing of modelled microservice compositions. Modellers must be able to visualise the service dependency graph,

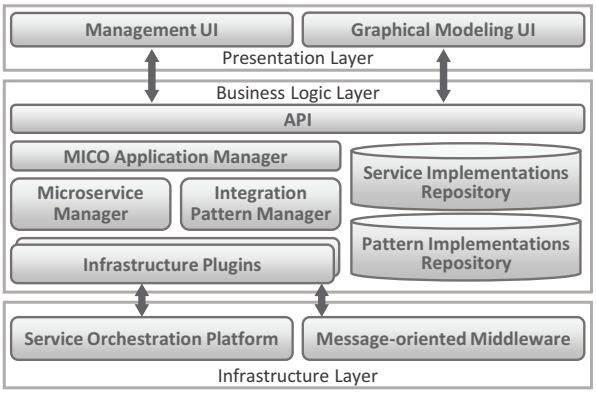

Fig. 5. A system architecture enabling the MICO approach.

which facilitates understanding the structure of a microservice composition and also provides some basic metrics for monitoring the health of the deployment, e.g., to quickly identify failed deployments. Moreover, to ease the burden of testing deployments, it should be possible to specify test messages that can be used to verify the correct functionality of message processing components, e.g., to verify the correct transformation and routing of messages. Another required management abstractions are related to transitioning between the steps of the approach, e.g., modification of versions and scaling configurations, updating already deployed components, or redeployment and undeployment of running applications.

VI. System Architecture and Prototype

We hereby introduce a system architecture enabling the MICO approach and discuss several architectural decisions with respect to certain components of the system. Furthermore, we present a prototypical implementation that enables modelling and deploying MICO applications on top of Kubernetes, with the aim of illustrating the feasibility of our approach. The implemented MICO system can be used to facilitate the transition from modelled microservice compositions to managing deployed instances on a Kubernetes cluster by abstracting away technology-specific details.

A. System Architecture

Figure 5 illustrates a system architecture enabling the MICO approach, which includes all main components needed to implement the described functionalities. Users can interact with the system by means of a *Graphical modelling and Management UIs* providing a graphical modelling tool and management interfaces for composing and deploying MICO applications as described in Sections V-C and V-D.

The business logic layer consists of several major components that implement the described functionalities of MICO for managing microservice compositions, exposed using an *API*, e.g., a REST API. The *Microservice Manager* provides functionalities for managing MICO Filters implementing business logic, e.g., importing new microservices into the system and storing them in the *Service Implementations Repository*.

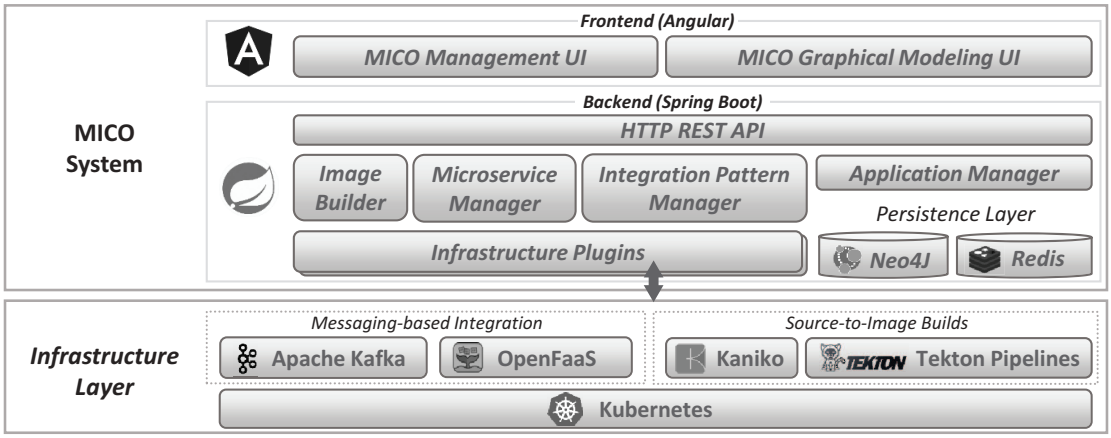

Fig. 6. A prototypical implementation of the MICO system.

The system coordinates the source-to-image workflow for generating the corresponding runnable building blocks based on provided source code repository links as discussed in Section V-B. Similarly, the *Integration Pattern Manager* provides functionalities for managing and storing the Integration Patterns in *Pattern Implementations Repository* as described in Section V-C, e.g., default implementations configured based on the user-provided integration logic, or custom implementations of entire patterns.

From the technical perspective, the event-driven nature of Integration Patterns makes Function-as-a-Service (FaaS) cloud service model a good candidate for hosting implementations of integration logic. In such case, FaaS-hosted functions can be triggered by messages originating from given queues or topics and scaled to zero after completing the integration task, helping to reduce the usage of available compute resources. Moreover, since FaaS hosts provided source code directly, it is easier for modellers to introduce custom integration logic by simply providing the required implementation to the MICO software layer, hence reducing the required management efforts. Hence, we envision the usage of a FaaS platform as a component for managing and processing integration tasks.

The *MICO Application Manager* provides the functionalities for composition, management, and monitoring of components within a microservice application by providing the functionalities described in Section V-C and Section V-E. For the deployment of created microservice compositions to a container-based orchestration platform and managing message-oriented middleware as discussed in Section V-D, technology specific *Infrastructure Plugins* are utilised, e.g., for the deployment to Kubernetes and configuring Apache Kafka-based integrations.

B. Prototype

We implemented a prototype of MICO-enabling platform[2], which follows the system architecture presented in Section VI-A, and targets the Kubernetes container orchestration platform. As shown in Figure 6, we use Apache Kafka[3] and OpenFaaS as means to enable the message-based integration of microservices. For the implementation of the MICO system architecture we use the Java-based framework Spring Boot[4]. In addition, Angular[5] is used for the development of the web interface that provides an extensible graph-based modelling component for creating microservice compositions and a set of management interfaces for managing MICO applications. The prototype provides an implementation of the main management components, i.e., Microservice Manager, Integration Pattern Manager, and MICO Application Manager, as described in Section VI-A. Furthermore, the backend provides Infrastructure Plugins required for supporting all the functionalities needed to deploy and manage MICO applications on Kubernetes with Apache Kafka and OpenFaaS used for messaging-based integration. In addition, the Image Builder component is implemented for coordinating source-to-image builds of Docker container images for imported MICO Services. Docker images are built directly in a Kubernetes cluster, using Tekton[6] and Kaniko[7], and staged into a DockerHub repository for further use as MICO application's building blocks. For interacting with Kubernetes and Tekton, the prototype relies on the corresponding clients maintained by fabric8[8]. The information about MICO services and designed MICO applications is stored using the graph database management system Neo4j[9], whereas the data about background jobs is stored using Redis[10], an in-memory data structure store.

We also implemented two default Integration Services representing the common enterprise integration patterns, namely *Content-Based Router* and *Message Filter*. The implementations of patterns use a generic Integration Service that handles

[2]https://github.com/UST-MICO/mico

[3]https://kafka.apache.org/
[4]https://spring.io/projects/spring-boot
[5]https://angular.io
[6]https://tekton.dev/
[7]https://github.com/GoogleContainerTools/kaniko
[8]https://github.com/fabric8io/kubernetes-client
[9]https://neo4j.com
[10]https://redis.io

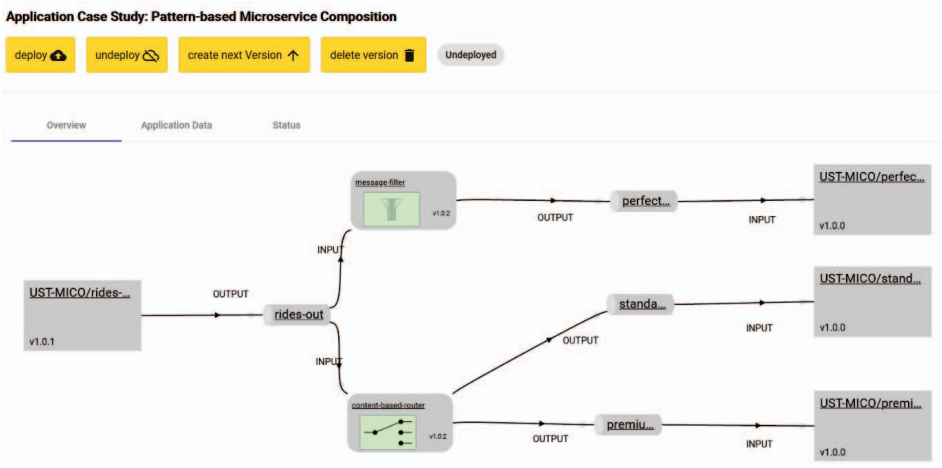

Fig. 7. An integration of microservices described in Section III, modelled using the graphical modelling UI in MICO prototype.

communication with Kafka, in combination with functions hosted on OpenFaaS that implement the actual integration logic, e.g., for message routing, splitting, and transformation. Functions are provided in a function store and users can adjust these functions based on custom requirements. Internally, a generic component communicating with Kafka read messages from specified topics, hands them over to the corresponding functions, and sends the resulting message to one or multiple output topics determined by the function.

VII. Case Study

In the following, we discuss how we modelled, deployed, and managed the microservice composition described in Section III on Kubernetes using the MICO system.

Step 1: Importing Application Building Blocks. As an initial step, we prototypically implemented[11] the four microservices depicted in Figure 1 as simple message producers and consumers, and imported them into the Service Implementations Repository of MICO by providing the URLs to their source code repository and a service specification with the metadata about the services. For example, in case of the *Rides Service*, the service specification describes that the service provides an interface that is accessible via port 80 on HTTP and can be used to communicate with the service via a REST-API. The specification also describes that the service uses messaging to communicate with other services and produces messages that contain information about completed rides that follow a given schema. As mentioned in Section V-B, the messages produced by MICO-enabled services follow the CloudEvents specification. Listing 1 shows an excerpt of an exemplary CloudEvents message produced by the Rides Service serialised as JSON. The event is of type completedRideEvent and originates from the event source RidesService. The schema, the event data of this message adheres to is provided in form of an URI in the dataschema attribute. For the sake of simplicity, in our example, the schema only contains a single data element with key customerRating. The event data itself is encapsulated in the data attribute following the defined schema.

Listing 1. Excerpt of a message following the CloudEvent schema

```
{
    "type" : "completedRideEvent",
    "source" : "RidesService",
    "id" : "A234-1234-1234",
    "dataschema": "someURI",
    "data" : { "customerRating" : "9.1", ... },
    ...
}
```

Step 2: Designing the MICO Model. Next, we used the imported services as building blocks of a MICO application that represents the microservice composition from Figure 1. To solve the given integration tasks, we implemented the Integration Patterns, and used them in the model to connect Business Logic Services together. Figure 7 displays the modelled composition of microservices using the graphical modelling component of the MICO system.

To solve the first integration task, i.e., to filter out messages sent to the Perfect Rides Service, we exploited the built-in Message Filter pattern (as Integration Pattern), which we configured to deal with the appropriate input and output topics. In our example, "perfect rides" is filtered based on the rating given by customers. A ride is considered to be perfect in case the rating is above 9.0. Hence, for configuring the Message Filter accordingly, we provided the data element that holds this information together with the defined threshold of 9.0.

The second integration task is solved by exploiting the built-in Content-based Router pattern (as Integration Pattern), which routes messages to Standard or Premium Loyalty Services depending on the value of message's respective field. After finalising the MICO Model of the considered application, we were able to deploy it on the underlying infrastructure layer using respective plugin for Kubernetes.

Steps 3 and 4: Generating and Enacting the Deployment. To deploy and test the created composition, we used a single

[11] https://github.com/UST-MICO/mico-case-study

node Kubernetes cluster that we run using Docker Desktop with installed Apache Kafka as a message-oriented middleware to enable message-based communication, and OpenFaaS for managing and handling the interaction with integration patterns. We initiated the deployment of the application via the MICO Management UI, which resulted in automatically generating the artefacts needed for deploying the application, and in their actual deployment on the Kubernetes cluster.

We also tested the deployment using the monitoring endpoints provided by implemented microservices. In particular, to verify the correct message passing of the deployed application, we observed the received messages to check if they comply with the integration requirements for filtering perfect rides, and routing to the suitable loyalty service.

VIII. Related Work

Microservices are on the rise, with several research efforts analysing their potentials and possible pitfalls, e.g., [4], [14], and [15], just to mention some. From all such efforts, it emerges a need for supporting developers and administrators of MSA-based applications, especially during the design, integration and deployment of their applications.

Since Newman's guide to building microservices [1], various approaches have been proposed to provide such support, typically focusing either on the high-level design of microservices, on their actual integration, or on their deployment. For instance, [16], [17], [18] and [19] all focus on supporting the *design* of MSA-based applications. [16] investigates the applicability of existing methodologies for the design of enterprise architectures to MSA, and show how these can be used to suitably design microservice compositions. [17] presents MicroART, a tool for automatically determining the architecture of a running MSA-based system, hence enabling to view and reason on its architecture to refine its design. [18] proposes an approach to model MSA by means of so-called "microservice ambients", which allow to analyse and refine (if needed) the granularity of the microservices forming an application. Finally, [19] propose a solution for representing MSA in TOSCA, and for identifying and resolving architectural smells in such MSA.

Other approaches worth mentioning are [20], [21], [22], [23], [24] and [25], all focusing on the actual *integration* of the microservices forming an application. [20], [21] and [22] indeed propose three different approaches for developing the integration among the microservices in an application, based on RESTish protocols, on aspect-oriented programming and on iterative refinements, respectively. [23] and [24] illustrate how integrate and engineer MSA-based application by exploiting Jolie, a programming language for microservice compositions. [25] instead proposes an approach for automatically generating the test cases needed for testing the developed integrations in a MSA-based application.

[26], [27], [28] and [29] instead focus on the *runtime support* for MSA-based application. [26] and [27] indeed illustrate two different cloud-based deployment platforms for continuously managing microservices, dynamically scaling them based on their load and capable of restoring failed microservice instances. [28] presents Beethoven, an event-driven lightweight middleware platform for orchestrating the microservices forming an application based on their event/-data flow. [29] instead proposes an integrated dashboard for deploying, managing and monitoring MSA-based application.

All above approaches however differ from ours, as they focus on only one among design, integration and runtime support of microservices, and relying on the technical expertise of application developers and administrators to carry out the other phases. The MICO approach we propose in this paper instead tries to provide a first "one-click" solution, by synergically combining the design, integration and runtime support of MSA-based application, and by abstracting away from all needed technology-specific expertise.

In this perspective, the closest to the MICO approach are the solutions proposed in [30] and [31]. [30] proposes a UML-based approach for designing the composition of microservices forming an application, by exploiting Enterprise Integration Patterns (EIP [6]) to design their integration. [30] also illustrate a semi-automated solution to generate the actual sources implementing the integration among the microservices in an application. This reduces the amount of technical efforts required to application developers and administrators, who are anyway still required to profound their technical expertise in completing the development of the integration and in specifying the deployment of their applications on container-based orchestrators. Similar considerations apply to the composition solution proposed by [31]. Even if the latter fully automates the generation of service-to-service integrators, it still relies on the technical expertise of application operators to set up the deployment of an application.

In summary, to the best of our knowledge, the MICO approach is the first trying to support application administrators throughout the design, integration, and deployment of their MSA-based application, by trying to synergically combine such phases while at the same time abstracting away from technology-specific details.

IX. Approach Generalisation Directions

Essentially, the main goal of the MICO approach is to facilitate the transition from microservice integration models to deployments on a desired target infrastructure. Similar to other model-driven approaches, the extent to which the respective MICO concepts can be adapted and generalised, e.g., to support different kinds of infrastructure layers or to employ additional service interaction types, is an interesting issue to investigate. In the following, we outline several possible directions related to generalisation of the MICO approach. In particular, we discuss (i) how generic the infrastructure layer can be and whether the approach is bound only to message-based communication, and (ii) to what extent the set of patterns used in the MICO meta-model can be extended.

Since our approach and the underlying meta-model discussed in Sections IV and V focus solely on the EIP patterns [6], the modelled communication channels are bound to

messaging-based interaction by design. Such design restrictions, essentially, limit the applicability of the approach to modelling data pipelines. Especially, considering the requirements imposed on message formats, the modelled MICO applications would benefit more from microservices implemented as reusable "data processors" that can be integrated into multiple different pipelines rather than microservices crafted for one specific use case. On the other hand, to introduce additional means of interaction between microservices the meta-model can easily be extended by adding new types of *Pipes* connecting *Microservices*. Such generalisation would allow modelling messaging-based integrations combined together with, e.g., service-to-service interactions via HTTP-based APIs. As a first step towards mixing different communication styles in modelled MICO applications, our prototype supports connecting microservices using the interface connections, which are implemented by establishing mappings among ports exposed by a microservice and Kubernetes *Services*. These Kubernetes-specific mappings are then used for service discovery at runtime[12]. Although as an early feature it is possible to deploy MSAs interacting by means of HTTP-based interfaces, the technical implications arising when combining two different kinds of communication channels require further assessment.

Another important question is how to support other kinds of infrastructure layers for deploying modelled MICO applications, e.g., deployments that also combine provider-managed service offerings such as AWS SQS or AWS Kinesis for microservice-specific communication. In such cases, deploying the integration middleware is not always straightforward, e.g., the integration can be implemented using functions hosted on AWS Lambda interacting via AWS SQS. Moreover, the ways microservices are implemented and deployed themselves vary too, from Container-as-a-Service and Platform-as-a-Service offerings to purely serverless, FaaS-based implementations. In our prototype, we focus solely on Kubernetes-based deployments due to a large popularity of container orchestration systems for deploying microservices. However, the underlying system architecture and meta-model are technology-agnostic, which allows enabling other kinds of infrastructure layers by implementing corresponding infrastructure plugins. For example, for enabling AWS-based deployments one might implement an infrastructure plugin generating AWS Cloud Formation templates to deploy modelled MICO applications. Similar approaches exist in the context of canonical deployment modelling [32].

Another interesting extension direction concerns how to employ other kinds of patterns in the MICO meta-model, and especially the microservice-centric patterns [33], e.g., how the model such patterns as API Gateway [33] or Microservice Chassis [33] as parts of the MICO application. Although the MICO meta-model is flexible enough to allow forming composite patterns such as Composed Message Processor [6], it is worth emphasising that in certain cases placing the pattern implementation outside the microservice's boundaries is unnecessary. For example, the Polling Consumer [6] pattern fits perfectly as an internal part of a microservice, hence being encapsulated into the respective Microservice entity in the MICO application model. The Microservice Chassis pattern has a similar effect as it is intended to be an internal part of a microservice, hence also being encapsulated in Microservice entities. On the other hand, the derivatives of the Gateway[13] pattern such as Messaging Gateway [6] or API Gateway patterns, essentially, are separate components that can be crafted manually or rely on well-established solutions like Kong[14]. This allows having such patterns as first-class citizens in the MICO application model, but only, if the meta-model is extended with other forms of communication channels as discussed previously. The possible need to refine the communication channels also applies to more sophisticated patterns such as Process Manager [6], where multiple microservices interact with each other via a dedicated orchestrator component. Depending on infrastructure-related technicalities, modelling of corresponding communication channels might require more information than regular message-based communication channels, e.g., to configure the orchestrator implementation. While the MICO approach focuses on message-based integration patterns, extending the underlying meta-model with other subtypes of the Filter and Pipe modelling constructs is a core requirement for generalising the applicability of our approach, which in turn requires further classification of pattern placement in the MICO meta-model.

X. Conclusion

We introduced MICO, a model-driven approach for deploying and managing microservice compositions and a system architecture enabling it. The main part of the approach is the MICO meta-model that blends together architectural and deployment aspects facilitating the transition from integration models to running deployments. Alongside with modelling interface-based service integration, the MICO Model enables using so-called integration services to model messaging-based service interaction. Integration services rely on implementations of common enterprise integration patterns, facilitating loosely-coupled integration of microservices while at the same time abstracting from the technical details related to the underlying infrastructure deployment requirements. To validate our approach, we implemented the presented system architecture prototypically using Kubernetes for container orchestration, Apache Kafka as a message-oriented middleware, and OpenFaaS for handling the service integration logic, and we run a concrete case study based on a third-party application.

In future work, we plan to focus on research directions discussed in Section IX and extend our approach to modelling and deploying composition of cloud-native applications also including multi-cloud use cases and serverless-only application topologies. Furthermore, we plan to investigate how existing

[12]https://mico-docs.readthedocs.io/en/latest/tutorials/02-manage-service.html#manage-service-interfaces

[13]https://martinfowler.com/eaaCatalog/gateway.html

[14]https://konghq.com/kong

architecture modelling languages such as ArchiMate and cloud modelling languages can be used to expand potential usage scenarios for the MICO meta-model. Additionally, there exist multiple pattern languages describing patterns refining other higher-level patterns [34]. For example, Strauch et al. [35] refine the higher-level EIPs such as Aggregator and Splitter patterns to tackle issues related to data confidentiality. In our work, we plan to combine these ideas for modelling and deployment of integration patterns related to particular problem domains in MICO applications, e.g., compliance with company's data confidentiality requirements.

ACKNOWLEDGMENT

This work is partially funded by the EU project *RADON* (825040), the German Research Foundation (DFG) project ADDCompliance (636503), and the project *DECLware* (University of Pisa, PRA_2018_66). We would also like to thank the anonymous reviewers for the constructive feedback which helped improving this paper.

REFERENCES

[1] S. Newman, *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[2] J. Lewis and M. Fowler, "Microservices: a definition of this new architectural term," mar 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html

[3] O. Zimmermann, "Microservices tenets," *Computer Science-Research and Development*, vol. 32, no. 3-4, pp. 301–310, 2017.

[4] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, "The pains and gains of microservices: A systematic grey literature review," *Journal of Systems and Software*, vol. 146, pp. 215–232, 2018.

[5] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural patterns for microservices: a systematic mapping study." SCITEPRESS, 2018.

[6] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004.

[7] M. Garriga, "Towards a taxonomy of microservices architectures," in *International Conference on Software Engineering and Formal Methods*. Springer, 2017, pp. 203–218.

[8] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: current and future directions," *ACM SIGAPP Applied Computing Review*, vol. 17, no. 4, pp. 29–45, 2018.

[9] J. Turnbull, *The Docker Book*. James Turnbull, Jul. 2014.

[10] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[11] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, and J. Wettinger, "Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications," in *Proceedings of the 9th International Conference on Pervasive Patterns and Applications (PATTERNS)*. Xpert Publishing Services, Feb. 2017, pp. 22–27.

[12] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, Aug. 1977.

[13] R. Richardson, "Application integration patterns for microservices: Fan-out strategies," nov 2019. [Online]. Available: https://aws.amazon.com/blogs/compute/application-integration-patterns-for-microservices-fan-out-strategies/

[14] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds. Springer International Publishing, 2017, pp. 195–216.

[15] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.

[16] J. Bogner and A. Zimmermann, "Towards integrating microservices with adaptable enterprise architecture," in *2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW)*, 2016, pp. 1–6.

[17] G. Granchelli, M. Cardarelli, P. D. Francesco, I. Malavolta, L. Iovino, and A. D. Salle, "Microart: A software architecture recovery tool for maintaining microservice-based systems," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, pp. 298–302.

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

[19] A. Brogi, D. Neri, and J. Soldani, "Freshening the air in microservices: Resolving architectural smells via refactoring," in *ICSOC 2019 Workshops*. Springer, 2019.

[20] J. Ignacio Fernández-Villamor, C. Á. Iglesias, and M. Garijo, "Microservices - lightweight service descriptions for rest architectural style," in *Proceedings of the 2nd International Conference on Agents and Artificial Intelligence - Volume 1: ICAART,*, INSTICC. SciTePress, 2010, pp. 576–579.

[21] T. Cerny, "Aspect-oriented challenges in system integration with microservices, soa and iot," *Enterprise Information Systems*, vol. 13, no. 4, pp. 467–489, 2019.

[22] M. Zuniga-Prieto, E. Insfran, S. Abrahao, and C. Cano-Genoves, "Incremental integration of microservices in cloud applications," in *Information Systems Development: Complexity in Information Systems Development (ISD2016 Proceedings)*, 2016.

[23] C. Guidi, I. Lanese, M. Mazzara, and F. Montesi, "Microservices: A language-based approach," in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds. Springer International Publishing, 2017, pp. 217–225.

[24] F. Montesi and J. Weber, "Circuit breakers, discovery, and API gateways in microservices," *CoRR*, vol. abs/1609.05830, 2016.

[25] S. Rajagopalan and S. Sinha, "Automated test input generation for integration testing of microservice-based web applications," U.S. Patent n. US10261891B2, 2016.

[26] G. Toffetti, S. Brunner, M. Blöchlinger, F. Dudouet, and A. Edmonds, "An architecture for self-managing microservices," in *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*. Association for Computing Machinery, 2015, p. 19–24.

[27] H. Kang, M. Le, and S. Tao, "Container and microservice driven design for cloud infrastructure devops," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, 2016, pp. 202–211.

[28] D. Monteiro, R. Gadelha, P. H. M. Maia, L. S. Rocha, and N. C. Mendonça, "Beethoven: An event-driven lightweight platform for microservice orchestration," in *Software Architecture*, C. E. Cuesta, D. Garlan, and J. Pérez, Eds. Springer International Publishing, 2018, pp. 191–199.

[29] B. Mayer and R. Weinreich, "A dashboard for microservice monitoring and management," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, pp. 66–69.

[30] R. Petrasch, "Model-based engineering for microservice architectures using enterprise integration patterns for inter-service communication," in *2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. IEEE, 2017, pp. 1–4.

[31] E. Ben Hadj Yahia, L. Réveillère, Y.-D. Bromberg, R. Chevalier, and A. Cadot, "Medley: An event-driven lightweight platform for service composition," in *Web Engineering*, A. Bozzon, P. Cudre-Maroux, and C. Pautasso, Eds. Springer International Publishing, 2016, pp. 3–20.

[32] M. Wurster, U. Breitenbücher, A. Brogi, G. Falazi, L. Harzenetter, F. Leymann, J. Soldani, and V. Yussupov, "The edmm modeling and transformation system," in *International Conference on Service-Oriented Computing*. Springer, 2019, pp. 294–298.

[33] C. Richardson, *Microservices Patterns*. Manning Publications Company, 2018.

[34] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann, A. Hadjakos, F. Hentschel, and H. Schulze, "Leveraging Pattern Application via Pattern Refinement," in *Proceedings of the International Conference on Pursuit of Pattern Languages for Societal Change (PURPLSOC 2015)*. epubli, Jun. 2015.

[35] S. Strauch, U. Breitenbücher, O. Kopp, F. Leymann, and T. Unger, "Cloud Data Patterns for Confidentiality," in *Proceedings of the 2nd International Conference on Cloud Computing and Services Science (CLOSER 2012)*. SciTePress, Apr. 2012, pp. 387–394.

All links were last followed on September 3, 2020.