Received: 27 September 2021 | Revised: 7 December 2021 | Accepted: 20 December 2021

DOI: 10.1002/spe.3073

RESEARCH ARTICLE

WILEY

Standards-based modeling and deployment of serverless function orchestrations using BPMN and TOSCA

Vladimir Yussupov[1] | **Jacopo Soldani**[2] | **Uwe Breitenbücher**[1] | **Frank Leymann**[1]

[1]Institute of Architecture of Application Systems, University of Stuttgart, Stuttgart, Germany

[2]Department of Computer Science, University of Pisa, Pisa, Italy

Correspondence
Vladimir Yussupov, Institute of Architecture of Application Systems, University of Stuttgart, Universitätsstrasse 38, 70569 Stuttgart, Germany.
Email: yussupov@iaas.uni-stuttgart.de

Abstract

Function-as-a-Service (FaaS) is a cloud service model enabling to implement serverless applications for a variety of use cases. These range from scheduled calls of single functions to complex function orchestrations executed using orchestration services such as AWS step functions. However, since the available function orchestration technologies vary in functionalities, supported modeling languages, and APIs, modeling such function orchestrations and their deployment require significant technology-specific expertise. Moreover, the resulting models are typically not portable due to provider- and technology-specific details, and major efforts are required when exchanging an orchestrator or provider due to such lock-ins. To tackle this issue, we introduce a vendor- and technology-agnostic method for the modeling and deployment of serverless function orchestrations, which relies on the business process model and notation (BPMN) and topology and orchestration specification for cloud applications (TOSCA) standards for modeling function orchestrations and their deployment, respectively. We also present a toolchain for modeling serverless function orchestrations in BPMN, generating proprietary models supported by different function orchestration technologies from BPMN models, specifying their actual deployment in TOSCA, and then enacting such deployment. Finally, we illustrate a case study applying our method and toolchain in practice.

KEYWORDS

BPMN, FaaS, function orchestrations, function-as-a-service, serverless, TOSCA, workflows

1 | INTRODUCTION

In cloud computing, the term *serverless*[1] is often associated with Function-as-a-Service (FaaS) platforms and the underlying programming model as developers are relieved from many traditional infrastructure management tasks.[2] With FaaS, developers can host event-driven code snippets that are automatically scaled by providers (including scaling to zero instances),[3] which means that providers are responsible for managing a suitable deployment stack for to-be-hosted code snippets. At the same time, the reduced management of component's deployment stack and scaling configuration gives to users the impression that there are no servers at all. Interestingly, these properties can be observed not only in FaaS offerings but also in many other *provider-managed services*.[4] For example, there are database, messaging, logging, or monitoring offerings that can be used "as a service", without requiring the user to manage the deployment stack and scaling configurations. As a result, from the cloud consumer point of view the term "serverless" can be treated as a general pattern for hosting application components in a way that does not require to take care of deploying, managing, and scaling the components.[5]

However, fine granularity, strict separation of concerns, and the inherent properties of FaaS—such as constrained amount of resources a function can use and limited execution time[6]—often result in the need to compose multiple functions together, which can be achieved in different ways. For example, functions are often composed by means of event-driven interactions with other serverless offerings such as databases or message queues.[7] Besides such event-driven compositions of components, there are also technologies that enable the *explicit orchestration* of FaaS functions following the idea of workflows[8] that can be used to implement common control flow patterns such as sequence or parallel branching.[9] Many prominent cloud providers such as AWS, Azure, or IBM offer such capabilities with their function orchestration services: by using provider-specific orchestration modeling languages developers can compose multiple serverless functions into complex serverless function orchestrations. Moreover, besides these provider-managed function orchestration services that can be used directly "as a service", there are also installable function orchestration technologies that provide similar functionalities for installable FaaS platforms, for example, Apache Openwhisk Composer[10] and Fn Flow[11] enable function orchestration for Apache Openwhisk[12] and Fn,[13] respectively. However, also these installable function orchestration technologies are not compatible with each other and with provider services as they employ own technology-specific formats. For the sake of simplicity, in the following, we refer to both installable function orchestrating technologies as well as provider-managed function orchestration services as *function orchestrators*.

Essentially, to use function orchestrators for composing several FaaS functions developers need to follow a set of repetitive tasks. Firstly, the *business logic needs to be implemented* according to provider's requirements, for example, interaction with provider-specific services as well as complying with execution and size quotas of FaaS platforms. Secondly, an *orchestrator-specific function orchestration model*describing the desired function composition needs to be created. Depending on the chosen function orchestrator, function orchestration models can be specified using proprietary technology- or provider-specific DSLs, or by using general-purpose programming languages.[14] As a result, due to the proprietary and nonstandardized characteristics of orchestration modeling languages offered by function orchestrators, the resulting function orchestration models are not portable, for example, a function orchestration modeled for AWS step functions[15] cannot be executed on Azure durable functions.[16] Furthermore, the capabilities of function orchestrators are very different, which results in the problem that it is not always possible to create a semantically equivalent function orchestration model for all providers and technologies. For example, unlike AWS step functions, Apache Openwhisk Composer does not provide a native construct for modeling delayed function invocations.

Moreover, to automate the deployment of modeled function orchestrations, typically a provider- and technology-specific deployment model needs to be defined. Here, apart from provider-specific deployment automation technologies such as AWS CloudFormation,[17] third-party technologies such as Ansible[18] or the Serverless Framework[19] are typically employed in practice. However, also for this task, we face similar problems as discussed before for modeling function orchestrations, since deployment technologies differ significantly from each other leading to lock-ins and portability issues.

As a result, modeling function orchestrations and their deployments requires expertise in the chosen technology- and provider-specific modeling languages, that is, function orchestration modeling languages and deployment modeling languages. Therefore, considerable efforts are required when the chosen function orchestrator or deployment automation technology needs to be exchanged since the resulting target models are not portable and need to be reimplemented using other technology- and provider-specific modeling languages. One way to facilitate these modeling tasks is to employ existing *technology-agnostic standards*in the context of each respective task. However, while there exist standardization efforts such as the Serverless Workflow Specification[20] that focus on standardizing function orchestrations by executing them on dedicated standard-compliant runtimes, no standards are currently adopted neither for executing function orchestration models directly on available function orchestrators such as AWS step functions, nor for modeling their respective deployments on these function orchestrators. Therefore, in this paper, we propose a method that enables deploying and executing modeled function orchestrations directly on chosen existing function orchestrators and is based on established standards, namely business process model and notation (BPMN)[21] for modeling function orchestrations and topology and orchestration specification for cloud applications (TOSCA)[22] for modeling their deployment in provider- and technology-agnostic manner. This enables reducing the amount of technology-specific tasks when modeling function orchestrations and their deployments and fosters the models reuse for different target function orchestrators. As neither BPMN nor TOSCA are supported by available function orchestrators, we also present how BPMN models can be automatically transformed into proprietary orchestration formats and combined with TOSCA-compliant deployment technologies to enable automatically deploying them.

In summary, we present a *method for standards-based modeling and deployment of serverless function orchestrations* that enables modeling and automated deployment of function orchestrations in technology-agnostic manner with BPMN and TOSCA. The main contributions of this work can, hence, be summarized as follows:

(i) We introduce a *uniform technology-agnostic function orchestration modeling approach* in which function orchestrations are modeled using BPMN. Moreover, we show how these BPMN-based function orchestration models can be automatically transformed into different target orchestrator formats, for example, Amazon States Language.[23] Our uniform modeling and transformation approach is obtained by analyzing existing serverless function orchestrators from three major providers (AWS, Azure, and IBM) with respect to the properties of the underlying orchestration modeling languages.

(ii) We present a *TOSCA-based function orchestration deployment modeling approach* enabling to declaratively model deployments of serverless applications that incorporate serverless function orchestrations. This modeling approach is built on top of our previous work for modeling serverless deployments.[24] As TOSCA models can be directly executed by standalone TOSCA deployment systems such as OpenTOSCA[25] or xOpera,[26] there is no need to transform these models to provider-specific formats as needed for our BPMN function orchestration models.

(iii) We implemented a *standards-based function orchestration modeling and deployment toolchain*to enable applying our method in practice. This includes a prototypical implementation of the modeling and transformation tool called *BPMN for function orchestrations (BPMN4FO)* for modeling of serverless function orchestrations using BPMN and an extension of the open source TOSCA-based deployment modeling tool Eclipse Winery.[27] In this work, we use the open source TOSCA deployment automation technology xOpera[26] to automatically deploy the produced TOSCA deployment models.

(iv) We present a *case study* validating our modeling approaches and the introduced toolchain. More precisely, we show how we developed and executed the deployment of a serverless application which incorporates an extract-transform-load (ETL) function orchestration for analyzing the air quality data inspired by an ETL function orchestration example for AWS step functions available on GitHub.[28]

The remainder of this article is structured as follows. Section 2 presents a motivating scenario and the research question, while Section 3 describes the necessary background. Section 4 provides an overview of our method for standards-based modeling and deployment of serverless function orchestrations. Section 5introduces our uniform technology-agnostic function orchestration modeling approach, based on BPMN. Section 6 introduces our TOSCA-based function orchestration deployment modeling approach. Section 7 introduces a toolchain enabling to use our method in practice, whilst Section 8 presents its actual application in a case study. Section 9 discusses some benefits and limitations of our method with respect to different portability aspects of function orchestration models. Finally, Sections 10 and 11 discuss related work and draw some concluding remarks, respectively.

2 | MOTIVATING SCENARIO AND RESEARCH QUESTION

A typical serverless application is often heterogeneous in terms of components it comprises: a variety of different serverless offerings such as databases, message queues, logging, and monitoring services could trigger or be accessed by event-driven FaaS functions.[3] Furthermore, as discussed previously, multiple functions might need to be composed into an orchestration specified by a model, which can be then enacted using function orchestrators such as AWS step functions[15] or Azure durable functions.[16] To demonstrate how a serverless function orchestration can be combined with other components in a serverless application, we introduce a motivating example which describes a function orchestration implementing an ETL process and a deployment architecture of a serverless application that incorporates such a function orchestration. Based on this example, we motivate and elaborate on the research contributions of this work.

2.1 | Modeling function orchestrations

A high-level representation of a sample serverless functions orchestration is shown in Figure 1. In this example, publicly-available air quality data is extracted, transformed, and stored using FaaS functions and serverless object storage offerings. The example is based on the open source ETL function orchestration for AWS step functions available on GitHub.[28] This function orchestration comprises four FaaS functions (coordinated by a function orchestrator) that interact with serverless object storage buckets. The function *get files* is responsible for listing the files with air quality data from the previous day stored in a public object storage bucket *open air quality dataset*, and splitting them into chunks for parallel processing. Afterwards, the function orchestrator invokes multiple instances of the function *transform data*for each chunk containing file paths: the files listed in each chunk are downloaded, transformed into an intermediary result, and stored in the *intermediate results* bucket. After all intermediary results are received by the function orchestrator, the function *aggregate data* is invoked to merge them into a single file, to normalize the file structure, and to store the final result in the *final results* bucket. Finally, the function *clean up intermediary results* is invoked to remove the intermediary results. After the function orchestration is completed, a single file with the summary about the air quality for the previous day is available for further usage in the *final results*bucket.

In general, such kinds of function orchestrations can be modeled and executed both on standalone function orchestrators, for example, Apache Openwhisk Composer, and on provider-managed function orchestration services, for example, AWS step functions. For example, AWS step functions[15] can be used to orchestrate functions hosted on AWS Lambda,[29] with the orchestration models defined using Amazon states language (ASL).[23] The resulting model is a JSON-based specification describing how functions are to be coordinated as shown in Listing 1.

```
{ "StartAt": "GetFiles",
  "States": {
       "GetFiles": {..., "Next": "TransformData" },
       "TransformData": {..., "Next": "AggregateData" },
       "AggregateData": {..., "Next": "CleanUp" },
       "CleanUp": {..., "End": true }}
}
```

Listing 1: A simplified example of the function orchestration from Figure 1 modeled for AWS Step Functions in ASL

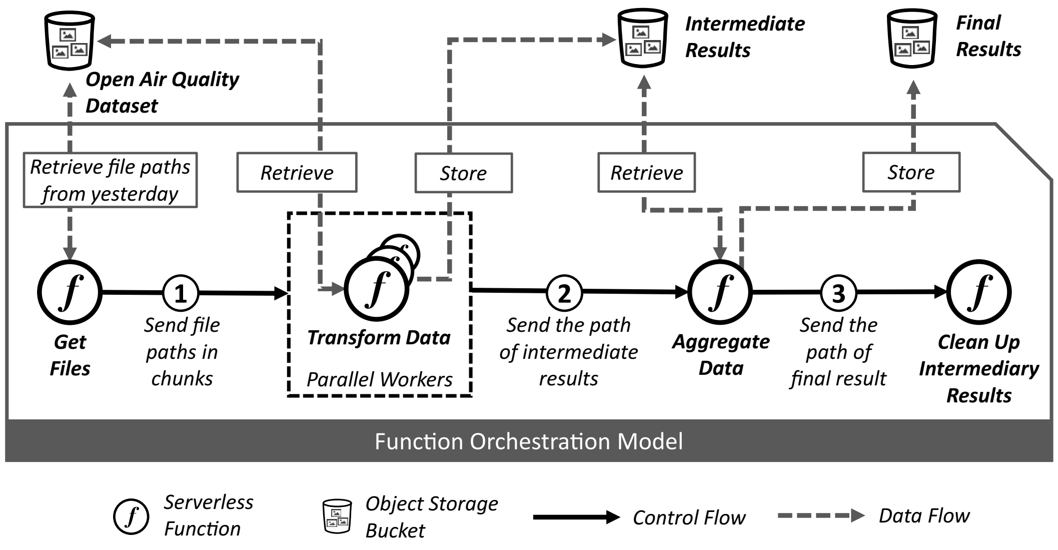


FIGURE 1 A serverless ETL function orchestration for processing open air quality data based on the ETL function orchestration for AWS step functions available on GitHub[28]

On the other hand, many function orchestrators require modeling such function orchestrations in general-purpose programming languages as so-called *orchestrating functions*.[14] For example, to implement such function orchestration on Azure functions[30] using Azure durable functions,[16] modelers can implement and deploy functions enacting the required function orchestrations in Python, JavaScript, or C#. Likewise, IBM Composer[31] can orchestrate functions hosted on IBM Cloud Functions,[32] with orchestrations defined in JavaScript or Python. Listing 2 shows an example modeled in JavaScript.

```
const composer = require('@ibm-functions/composer')
module.exports = composer.sequence(
composer.action('ListFiles', {limits: {timeout: 300000}}),
composer.map( composer.action('TransformData', {limits: {timeout: 300000}}) ),
composer.action('AggregateData', {limits: {timeout: 300000}}),
composer.action('CleanUp', {limits: {timeout: 300000}}) )
```

Listing 2: A simplified example of the function orchestration from Figure 1 modeled for IBM Composer in JavaScript

As seen from these examples, different function orchestrators use heterogeneous modeling styles and formats, that is, DSL-based models versus general-purpose function code, which are not compatible with each other and require understanding orchestration-specific libraries if the orchestration is implemented as function code. Furthermore, the implementation of similar control flow patterns, for example, sequential or parallel execution, often differs due to this heterogeneity of capabilities and features. This results in repetitive and error-prone tasks when the same function orchestration model needs to be ported from one orchestrator to another or produced for multiple target function orchestrators. This leads us to Challenge 1:

Challenge 1: *The heterogeneity of function orchestrators leads to portability problems that require creating individual function orchestration models or function orchestration code for every technology and provider, which significantly complicates exchanging function orchestrators. Moreover, for creating these models considerable technical expertise about the modeling language or libraries, respectively, and functionalities of the corresponding function orchestrator is required. Technology-agnostic standards are currently not supported in the context of function orchestration, which requires building models for every existing function orchestrator separately.*

2.2 | Modeling deployments of serverless applications that include function orchestrations

Although the described function orchestration can be executed independently of other applications, it is also possible that such function orchestration needs to be employed as a part of a larger serverless application. For example, since the function orchestration shown in Figure 1 is designed to process data only for the previous day, it is reasonable to trigger it on a scheduled basis, for example, using a timer event originating from a scheduling service. Furthermore, after the ETL processing is finished and the final result is ready, a nonorchestrated function might need to be invoked based on the notification event emitted by the object storage to notify external clients about the result location. Thus, often function orchestrations are part of larger serverless applications for reasons such as modularization and separation of concerns, which makes the deployment automation even more complex.

Figure 2 shows an example deployment architecture of a serverless application that combines the ETL function orchestration described in Section 2.1 with several additional tasks implemented using serverless components that do not rely on the function orchestrator. Firstly, the deployment of the ETL function orchestration involves (i) hosting the *function orchestration model* on the chosen *function orchestrator* (for example, an ASL model hosted on AWS step functions) and (ii) hosting the orchestrated functions on a compatible FaaS platform, for example, AWS Lambda. Next, the *function orchestration model* is triggered using a user-defined *timer rule* emitted using scheduling services such as Amazon EventBridge.[33] Moreover, after the final result is stored in the *results bucket* hosted on an *object storage service* such as AWS S3, the standalone *notify* function is triggered to generate a message with the details about the final result and publish it to a *message queue* created on a provider-managed

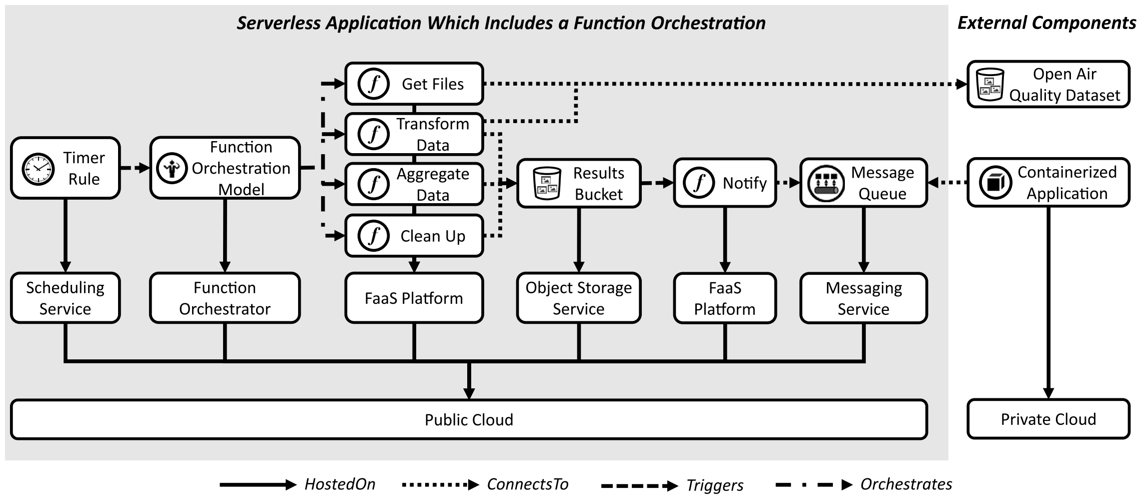


FIGURE 2 A deployment architecture for a serverless application that combines the ETL function orchestration from Figure 1

messaging service. Consequently, the message queue can be accessed by external clients, for example, a *containerized application* hosted on-premises, to process generated messages. Another external component is the open air quality dataset itself, since it is public and the respective bucket is not required to be deployed as shown in Figure 2.

Unsurprisingly, this serverless deployment architecture can be modeled for all major cloud providers using different deployment automation technologies. While the shown abstract model representing only generic service types such as "Object Storage", "FaaS Platform", or "Function Orchestrator" is basically the same for all providers, the actual deployable models employ provider-specific serverless offerings such as "AWS S3", "AWS Lambda", and "AWS step functions". Unfortunately, modeling deployments in an executable manner using heterogeneous deployment modeling languages differs considerably and requires technology-specific knowledge, for example, AWS SAM,[34] Serverless Framework,[19] Terraform,[35] or Ansible[18] all offer different, proprietary deployment modeling languages. For example, modeling such a deployment architecture for AWS using AWS SAM and Ansible would require understanding two different languages and approaches behind them: the formers requires declarative specification of components and their configurations as shown in a simplified example in Listing 3, whereas the latter requires defining sequences of tasks responsible for deploying the underlying components.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
 Resources:
  GetFiles:
   Type: AWS::Serverless::Function
   Properties: ...
  FunctionOrchestrationModel:
   Type: AWS::StepFunctions::StateMachine
   Properties: ...
  ...
```

Listing 3: A simplified excerpt from the function orchestration deployment from Figure 2 created using AWS SAM

As a result, modeling and porting such provider- and technology-specific deployment models requires additional effort, whereas standards such as TOSCA enable abstracting away technology-specific details. However, standards such as TOSCA are currently not supported for modeling function orchestration deployments, which leads us to Challenge 2:

Challenge 2: *The heterogeneity of deployment technologies and underlying deployment modeling languages leads to portability problems that require creating individual serverless application deployment models for every technology or provider, respectively, which complicates abstracting such models and porting them to other environments. Further, creating such models requires considerable technical expertise in the deployment modeling languages and functionalities of the corresponding deployment automation technology. Provider-agnostic standards are currently not supported for deploying serverless applications incorporating function orchestrations, which requires to build separate models for every provider and technology.*

2.3 | Research question

As can be seen from Sections 2.1 and 2.2, both the function orchestration and the deployment architecture for an application incorporating function orchestrations can be modeled in a similar fashion for different providers on an abstract level. However, when it comes to concrete target technologies, the modeling becomes more complex as each technology comes with own modeling languages and noticeable technical differences. Deployment automation also varies from technology to technology, which makes the corresponding deployment models significantly different depending on which technology is chosen, for example, AWS SAM, Serverless Framework, Terraform, or Ansible.

As a result, such technology- or provider-specific models—for both function orchestrations and their deployment—are (i) based on proprietary formats that are not portable between the providers. This (ii) results in major effort if providers need to be exchanged. Moreover, (iii) considerable technical expertise is required which results in error-prone modeling processes. On the other hand, standards such as BPMN and TOSCA are meant to provide a technology-agnostic way to represent workflows and application deployments by separating them from actual technologies used to process a given model, that is, a workflow management system or a deployment automation technology. Therefore, our research question for this work is as follows:

Research question: *"How can the standards BPMN and TOSCA be used to model function orchestrations as well as their deployments in a provider-agnostic manner while keeping the models portable and automatically executable?"*

3 | BACKGROUND

In this section, we briefly recall the fundamentals of workflow and deployment modeling focusing on two well-known standards: BPMN for modeling workflows and TOSCA for modeling application deployments.

3.1 | Workflow modeling and BPMN

A business process comprises one or more activities that need to be performed in a specific order to achieve a business goal.[36] Workflows are key enablers for the automation of business processes spanning multiple nonintegrated systems by providing languages and tools to model, enact, and manage such processes.[8] In particular, *workflow modeling languages (WML)* such as BPMN[21] or BPEL[37] enable representing the arrangements of activities in business processes using a variety of control flow patterns, for example, sequential composition of activities, conditional branches, or parallel execution.[9,38] The resulting models can then be consumed and enacted by *workflow management systems (WFMS)* which become responsible for executing modeled business processes. WFMSs are robust and provide multiple advanced functionalities, such as compensation and external events processing, which makes them a better candidate for enacting modeled business processes compared to custom implementations, for example, implementing a Java application for orchestrating several scripts. Furthermore, having explicit process models instead of hard-coded integration of required systems enables flexibly adjusting and reusing existing processes.[8,39]

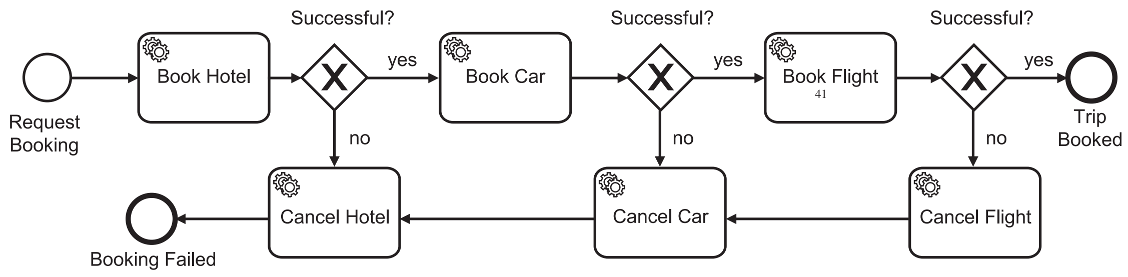


FIGURE 3 An example BPMN model describing an executable travel booking process[41]

BPMN[21] is a prominent standard for modeling workflows. It provides a semantically-transparent[40] graphical notation for representing control flows that span multiple kinds of activities in business processes. Processes in BPMN are defined by connecting *activities* and *events* using *sequence flows*, which express the execution ordering relations among them. Activities are the constructs for expressing units of work in the process, which can be atomic (*tasks*) or compound (*subprocesses*). The former are depicted as rounded rectangles with labels indicating the name of the activity. The latter can be modeled in collapsed and expanded manner: when collapsed, Subprocesses are depicted as regular activities but with a "+" marker in the bottom of the rounded rectangle. In contrast, expanded Subprocesses reflect the entire control flow happening within them similar to regular processes. *events* are modeled as circles and depict an occurrence of a specific fact during different stages of process execution, for example, *start events* initiate process instances (drawn as single-border circles), *end events*model the end of process instances (drawn as thick-border circles), and *intermediate events*happen at any point in-between (drawn as double-border circles). Moreover, *gateways* enable expressing divergence and convergence of sequence flows. Gateways are represented using diamond shapes with an internal marker that describes the routing behavior, for example, the "+" marker is used to represent *AND gateways* for parallel execution of activities, and "X" marker represents *exclusive gateways* where a path is chosen based on some condition.[42]

Figure 3 shows a typical BPMN model for a travel booking process.[41] Several activities must be executed to book a travel, whereas cancelation activities need to be enacted if some of the booking steps were unsuccessful. When a booking request is received a WFMS starts the execution from the *book hotel*activity and calls the next activity based on the result: if the hotel booking was successful, the WFMS invokes the *book car* activity, whereas the *cancel hotel* activity is invoked otherwise. Conditional gateways here ensure that only one path is taken by the WFMS—the travel is only booked when all activities are successful. Each activity can be implemented, for example, as standalone web services for processing hotel or car booking requests. Apart from adding more modeling elements, BPMN 2.0 also defines the operational execution semantics for all standardized model elements.[36] Camunda's BPMN workflow engine[43] is one example of a WFMS that can execute BPMN 2.0 models.

3.2 | Deployment modeling and TOSCA

The manual deployment of applications is inefficient and error-prone. Therefore, *deployment automation technologies* such as Terraform and Ansible have been developed to support modeling and automated deployment of applications.[44] Essentially, such models enabling deployment automation of applications follow one of two different styles: *imperative* and *declarative deployment models*.[45] The former describe a set of ordered operations that must be executed to deploy a given application, for example, a workflow or a script. The latter instead provide a declarative description of the desired configuration of an application including such details as the components forming an application, their configuration, as well as how to interconnect the components. Based on such declarative models, deployment automation technologies derive the concrete operations that should be enacted to deploy the application and reach its desired configuration. For instance, Terraform[35] and AWS cloud formation[17] consume declarative models defined using proprietary deployment modeling languages. Since all major deployment automation technologies support declarative deployment modeling,[44] in the following we focus on declarative deployment models.

TOSCA[22] is a standard by OASIS that introduces a vendor-neutral deployment modeling language which enables modeling application deployments in a declarative and imperative manner. TOSCA provides means to create reusable and portable models that describe the deployment of an application, which can then be executed using TOSCA-compliant deployment automation technologies, for example, OpenTOSCA[46] or xOpera.[26] In TOSCA terms, the structure and

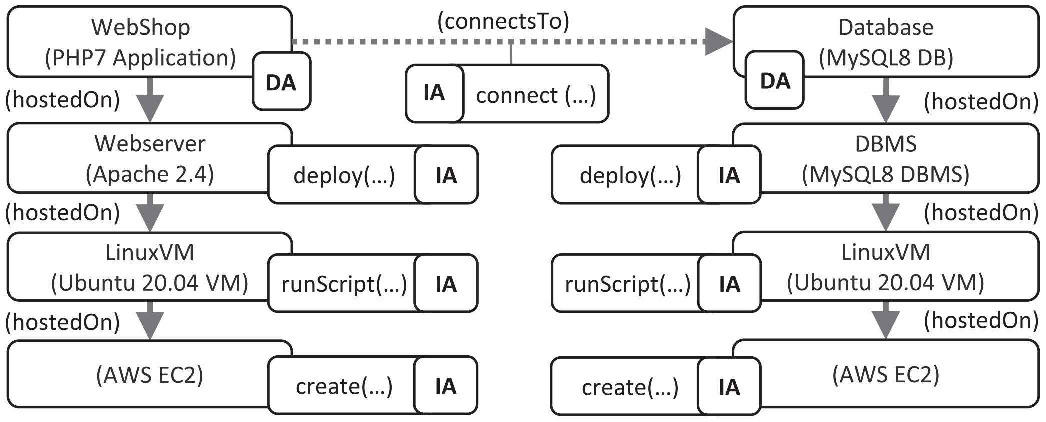


FIGURE 4 A simplified TOSCA topology template based on a LAMP-based application example[47]

configuration of an application is described as *topology template*, which specifies the components constituting the application, their properties, and the application topology, that is, the details on how components are interconnected. Essentially, topology templates are directed typed graphs with nodes representing application components and edges defining their relations. Additionally, modelers are able to introduce type hierarchies in TOSCA, that is, any desired semantics can be encoded in both nodes and edges, for example, direct calls or event-based invocation of components. In TOSCA terms, *node and relationship types* would represent specific component types and connectivity semantics, and can be instantiated as *node and relationship templates* in the topology template. To enable connecting nodes using a specific relationship, modelers can introduce a *requirement* in the source node type which matches with a specific *capability* on the target node type. Furthermore, TOSCA enables defining interface operations on node and relationship types, which are executed by the TOSCA-compliant deployment technologies in a standardized order, for example, to create, configure, and start a component. The actual deployment logic for a component, or *implementation artifacts (IAs)* in TOSCA terms, can be provided in multiple forms, for example, shell scripts or Ansible playbooks, allowing to deploy the same TOSCA model using different technologies by defining different implementation artifacts for node or relationship types. The actual business logic, for example, component's code packaged in a .zip archive, are attached to corresponding node templates as *deployment artifacts (DAs)*.[24] TOSCA supports ontological typing by the possibility to create own node and relationship types that can be referred to in node templates. Further, application models in TOSCA can be packaged using so-called *cloud service archives (CSARs)*, which group all the required information including model definitions, file artifacts, and metadata listing the contents of the CSAR. CSARs can then be consumed by TOSCA-compliant deployment technologies to enact the deployment of the application.

As a short introduction to modeling in TOSCA, Figure 4 shows an example e-commerce application topology.[24,47] In this topology, a *PHP7* application hosted on *Apache Web Server* interacts with a *MySQL8* database, both of which are hosted on separate *Ubuntu 20.04 VM* instances provisioned using the *AWS EC2* service offering. The resulting TOSCA topology is a directed typed graph with application components modeled as its nodes (*node templates* in TOSCA terms). Each Node Template is related to a specific type, for example, *WebShop* node template is of type *PHP7 Application*. The relationships among components (*relationship templates*) are modeled as graph edges, whose types in this example are either *hostedOn* or *connectsTo*. The application business logic and the required database schema are attached to the corresponding Node Templates as *deployment artifacts*. The actual deployment logic implementations reside in the corresponding types as *implementation artifacts*, enabling required lifecycle operations, for example, the *Ubuntu 20.04 VM* node type exposes the *runScript()* operation which enables executing scripts on the operating system.

4 | A METHOD FOR STANDARDS-BASED MODELING AND DEPLOYMENT OF SERVERLESS FUNCTION ORCHESTRATIONS

Serverless function orchestrations rely at least on two different modeling aspects: (i) modeling orchestrations that describe control flows spanning multiple functions and (ii) modeling the deployment of serverless applications incorporating such function orchestrations. In this section, we present our first contribution of this article—a method for using BPMN and TOSCA to model the function orchestrations and their deployment in a technology-agnostic manner. Thus, this overview of our method is a first step for answering our research question described in Section 2.

To tackle the heterogeneity of orchestration modeling languages described by *Challenge 1* (see Section 2.1), we abstract the level of details by employing BPMN as orchestrator-agnostic modeling layer that includes no details about concrete orchestration modeling languages and other technical details. The resulting BPMN models are intended to provide only a uniform, orchestrator-agnostic view on modeled function orchestrations without focusing on specifics of a certain function orchestrator. Further, to tackle the heterogeneity of deployment modeling languages described by *Challenge 2* (see Section 2.2), we abstract the level of details by employing TOSCA as technology-agnostic deployment modeling layer that hides details specific to deployment automation technologies.[22] The resulting TOSCA models aim to enable deploying serverless applications incorporating function orchestrations without references to particular deployment automation technologies. By combining the BPMN- and TOSCA-based modeling approaches, function orchestrations can be modeled and automatically deployed in a technology-agnostic manner. For simplicity, we describe the overall process as sequential.

As outlined by Figure 5, our method consists of two main phases. In the *technology-agnostic workflow modeling phase*, modelers first use BPMN to create *generic BPMN-based workflow models* using a BPMN-compliant modeling tool in *Step 1*. We, therefore, describe how BPMN can be used for modeling function orchestrations and which restrictions must be regarded to enable the following transformation into orchestrator-specific models using a common set of BPMN constructs for representing different function orchestration models. We present our BPMN-based approach, the underlying BPMN constructs, and modeling guidelines in Section 5. In *Step 2*, using a dedicated workflow transformation tool, the resulting BPMN model is transformed into a *proprietary function orchestration model* for the chosen function orchestrator, for example, ASL model for AWS step functions. Thus, this tackles the *challenge 1* presented in Section 2.1, as the resulting models can be executed automatically by providers.

In the *technology-agnostic deployment modeling phase*, modelers use TOSCA to define the *technology-agnostic function orchestration deployment model* for a desired serverless application incorporating function orchestration using a TOSCA-compliant modeling tool in *Step 3*. We therefore describe how TOSCA can be used for modeling function orchestration deployments using a common set of TOSCA modeling constructs for representing heterogeneous function orchestration deployments. In our previous work,[24] we showed that TOSCA can be used to represent the event-driven semantics commonly used in serverless applications, for example, establishing event bindings and describing events in a technology-agnostic manner. However, our previously introduced approach[24] does not support deployment modeling of function orchestrations. Therefore, we extend our previous approach in this article to include details relevant for modeling function orchestrations, hence, enabling the deployment of serverless applications incorporating such function orchestrations. We present our TOSCA-based approach, the underlying TOSCA constructs, and modeling guidelines in Section 6. Next, when the TOSCA model is created, in *Step 4*the generated function orchestration model needs to be attached as a TOSCA deployment artifact to enable deploying the orchestration itself. Finally, in *Step 5* the CSAR is exported using

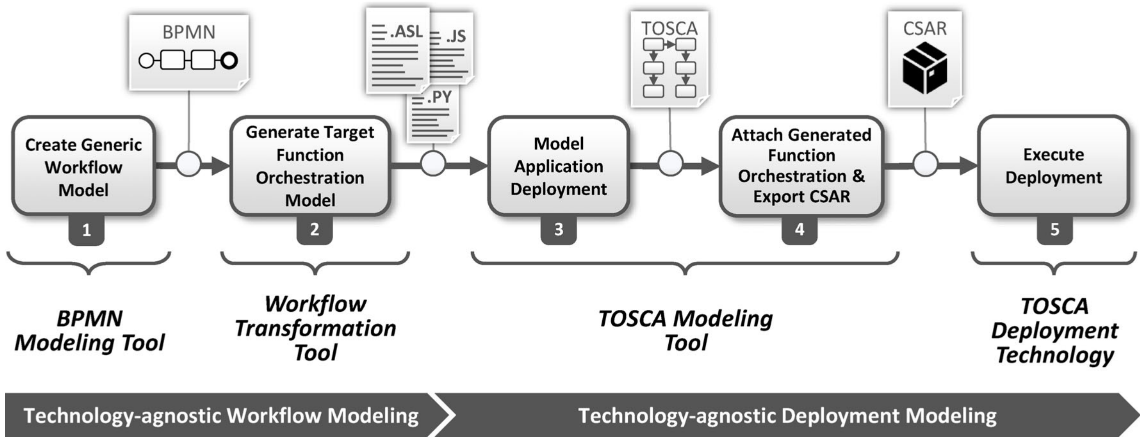

BPMN
.ASL
.JS
.PY
TOSCA
CSAR
Create Generic Workflow Model
Generate Target Function Orchestration Model
Model Application Deployment
Attach Generated Function Orchestration & Export CSAR
Execute Deployment
1
2
3
4
5
BPMN Modeling Tool
Workflow Transformation Tool
TOSCA Modeling Tool
TOSCA Deployment Technology
Technology-agnostic Workflow Modeling
Technology-agnostic Deployment Modeling

FIGURE 5 A method for standards-based modeling and deployment of serverless function orchestrations

the TOSCA modeling tool and can then be enacted by a TOSCA-compliant deployment technology of choice. Hence, this tackles the *Challenge 2* presented in Section 2.2 as the resulting deployment models can be executed fully automatically using TOSCA-compliant deployment technologies.

5 | UNIFORM TECHNOLOGY-AGNOSTIC FUNCTION ORCHESTRATION MODELING

This section presents our uniform technology-agnostic function orchestration modeling approach, which relies on a set of common BPMN constructs and restrictions that must be followed to enable uniform transformation into orchestrator-specific formats. The resulting BPMN models can be transformed into proprietary function orchestration model formats, for example, ASL for AWS step functions. This is achieved by identifying the mappings between BPMN and function orchestrators-specific modeling languages.

5.1 | Overview of the BPMN-based function orchestration modeling approach

To enable the transformation of BPMN-based function orchestrations into proprietary function orchestration formats, in our approach, we use: (1) a set of technology-agnostic *generic function orchestration modeling constructs* commonly encountered in function orchestration models, (2) a set of mappings from these *generic function orchestration modeling constructs* to *proprietary function orchestrator-specific modeling constructs*, and (3) a set of mappings from BPMN to *generic function orchestration modeling constructs*. As a result, the *BPMN function orchestration models* can be transformed into orchestrator-specific formats if compatible mappings *from BPMN 2.0 to target function orchestrator* are present, with the *generic function orchestration modeling constructs* serving as a "common ground" between these two worlds as shown in Figure 6. The list of considered generic function orchestration modeling constructs shown in Table 1 is inspired by well-known *workflow control flow patterns* by Russel et al.[38] and *serverless patterns* by Taibi et al.,[48] which we combined and adapted to the needs of function orchestration. The high-level idea is to have a generic list of constructs not directly

TABLE 1 Generic function orchestration modeling constructs based on workflow control flow and serverless patterns[38,48]

Generic function orchestration modeling construct	Description
Task	The *Task* construct represents an invocation of a serverless function.
Sequence	A *Sequence* is an ordered arrangement of serverless function invocations executed by a function orchestrator. After a function invocation is completed, the next one in the sequence is executed.
Conditional branching	A *Conditional branching* construct splits the function orchestration into two or more separate branches, of which exactly one is taken depending on associated branching conditions.
Parallel branching	A *Parallel branching* construct diverges a function orchestration into multiple branches, which are all executed concurrently. The next construct after parallel branching is only executed after all branches have completed.
Fan-out	A *Fan-out* construct takes in an array-like structure of data elements, and for each individual element in this array a new function instance is invoked for processing this element. The next construct after Fan-out is only executed after all invoked functions have completed.
Looping	A *Looping* construct repeats execution of a looped construct, for example, invocation of one function or a sequence of functions, as long as an associated looping condition evaluates to true.
Delay	A *Delay* pauses the execution of the function orchestration for a specified amount of time.
Subworkflow	A *Subworkflow* is a serverless function orchestration invoked from within an enclosing serverless function orchestration. Within the enclosing function orchestration, subworkflow construct behaves similar to a regular Task construct.
Error handling	Errors may occur during the execution of function orchestration models. The *error handling* construct represents the fault handling task(s) that need to be performed in case an error occurs.

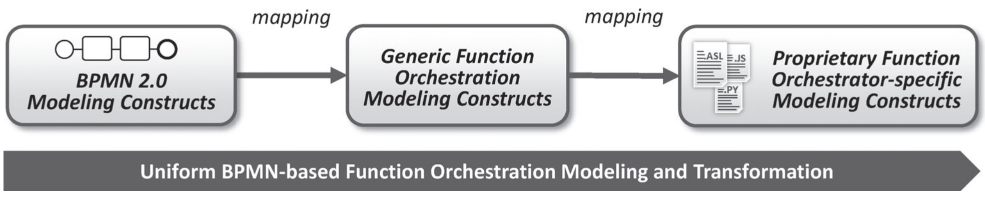

mapping
mapping
BPMN 2.0 Modeling Constructs
Generic Function Orchestration Modeling Constructs
ASL
JS
PY
Proprietary Function Orchestrator-specific Modeling Constructs
Uniform BPMN-based Function Orchestration Modeling and Transformation

FIGURE 6 An approach for uniform modeling and transformation of function orchestrations using BPMN

linked to a particular modeling language, but instead representing specific control flow semantics, for example, execution of *one* specific task, invocation of a *sequence* of tasks, execution of tasks *in parallel*, *delaying* the execution, and so forth. For example, a task in general workflow modeling represents an action to be done, whereas in the function orchestration context we require tasks for representing serverless function calls. Hence, as shown in Table 1, we use the *task*generic function orchestration modeling construct as a way to represent an *invocation of one function* in a function orchestration model independently of specific function orchestrators. Likewise, the *sequence*generic function orchestration modeling construct shown in Table 1 represents *a sequential execution of one or more functions* independently of specific function orchestrators. Another example orchestration construct is the *fan-out*, which is inspired by the idea of the *fan-out pattern*,[48] but implemented in the context of a function orchestration. We provide detailed descriptions for each generic function orchestration modeling construct in Table 1 and use those constructs as a language-independent way to analyze function orchestration modeling languages.

Having specified a set of generic function orchestration modeling constructs, to enable our BPMN-based modeling and transformation approach, we further need to obtain two sets of mappings: (1) *from BPMN to generic function orchestration modeling constructs* and (2) *from generic function orchestration modeling constructs to proprietary function orchestrator-specific modeling constructs* as shown in Figure 6. To analyze the mappings from generic to proprietary modeling constructs, we (i) conducted a review of the current capabilities of three major function orchestrators and documented their ways of implementing the generic function orchestration modeling constructs shown in Table 1. We present this technology review in Section 5.2 also including example snippets for each analyzed function orchestrator (simplified due to space constraints). Next, we perform the same analysis to identify mappings from BPMN 2.0 to the Generic Function Orchestration Modeling Constructs shown in Table 1. This review is presented in Section 5.3, together with the combination of both sets of identified mappings to show how function orchestration models created in BPMN can be transformed into the three analyzed function orchestration model formats.

5.2 | A review of serverless function orchestrators

In this section, we identify the mappings between the generic function orchestration modeling constructs shown in Table 1 and proprietary function orchestrator-specific modeling constructs (see Figure 6) for three prominent function orchestrators, namely AWS step functions,[15] Azure durable functions,[16] and Apache Openwhisk Composer.[10] To achieve this, we conduct a review of these function orchestrators to analyze which generic function orchestration modeling constructs these function orchestrators commonly support for deriving a uniform BPMN-based modeling approach compatible with these function orchestrators. AWS step functions and Azure durable functions are selected since they represent function orchestrators offered by the two largest cloud providers by market revenues.[49] Openwhisk Composer has been selected as a prominent installable function orchestrator, whilst still being used by IBM's commercial cloud offering, hence, making our approach also compatible with the proprietary function orchestrator from IBM, namely IBM Composer.[31]

5.2.1 | Function orchestration in AWS step functions

AWS enables expressing function orchestrations comprising multiple AWS Lambda functions (and other AWS services) through AWS step functions. Function orchestrations are modeled as finite state machines, declared with the Amazon state language (ASL).[23] The inputs and outputs of the entire state machine, as well as of its constituent states, are modeled in JSON.[15] The states of a state machine are all assigned with name and type, which determine their role in directing the control flow in the modeled function orchestration. We hereafter analyze whether/how the different types of states supported by AWS step functions enable realizing the generic function orchestration modeling constructs in Table 1.

Task. The generic function orchestration modeling construct *task* is expressed as the `ASL Task` state type, within which some work is to be executed.[23] The unit of work to be executed can be (i) an AWS Lambda function, (ii) a supported AWS service, or (iii) a worker running anywhere and implementing a special API.[15] For AWS-specific services, the Amazon resource name (ARN) needs to be specified, for example, ARN of the AWS Lambda function to be executed. It also constitutes the most relevant case to this work, as the Task type in ASL actually corresponds to executing a serverless function deployed on AWS.

Sequence. The generic function orchestration modeling construct *sequence* is realized by connecting each nonterminal state in ASL models, for example, connecting one ASL Task state to a subsequent ASL Task state. This is done by exploiting the `Next` property available in ASL states, which allows to put two states in a sequence, also prescribing that the JSON output produced by a state is passed as input to the subsequent state[23]—a simplified example is shown in Listing 4

```
{ "StartAt": "F1", "States":{"F1":{ ..., "Next": "F2"},..., "FX":{ ..., "End": true}}}
```

Listing 4: Sequence modeled in AWS Step Functions executing functions from F1 to FX

Conditional branching. The generic function orchestration modeling construct *conditional branching* is realized by `ASL Choice` state type.[23] By using this ASL state type, the execution flow can be directed to one branch from a set of possible branches, depending on given conditions. Each condition is associated with a subsequent state, which is the starting one of the branch to execute when the condition is satisfied. Should no defined condition be met, a `Default` branch is executed. An example of the conditional branching realization for AWS step functions is shown in Listing 5.

```
"myCond": {"Type": "Choice",
            "Choices": [{"Variable": "$.x","BooleanEquals": true,"Next": "true -> F1"}],
            "Default": "false -> F2" }
```

Listing 5: Conditional Branching inAWS Step Functions executing function F1 if the variable x is true and function F2 otherwise

Parallel branching. The generic function orchestration modeling construct *parallel branching* is realized by `ASL Parallel` state type, which allows executing multiple independent branches in parallel.[23] The input of such state is copied and passed to every parallel branch. The branches are then defined by their own state machines, possibly different one from another, and declared in the `Branches` array. After the execution, the outputs of all parallel branches are collected into an array of results and passed to the state following the `Parallel` state. A simple example of the parallel branching realization for AWS step functions is shown in Listing 6.

```
{"myParallel": {
    "Type": "Parallel", ...,
    "Branches": [
       {"StartAt": "F1", "States": {"F1": {"Type": "Task", ..., "End": true }}},
       {"StartAt": "F2", "States": {"F2": {"Type": "Task", ..., "End": true }}}]}
```

Listing 6: Parallel Branching modeled in AWS Step Functions executing functions F1 and F2 concurrently

Fan-out. The generic function orchestration modeling construct *fan-out* can be implemented through the `ASL Map` state type.[23] One such state takes an input array and uses an iterator to process each element of the input array, with the iterator being itself a function orchestration defined throughout a state machine. Once all elements of the input array are processed, the results are collected into an array and passed to the state following the `ASL Map` state—a simplified example is shown in Listing 7.

```
{"myFanout": {
    "Type": "Map",
    "ItemsPath": "$.inputElements",
    "Next": "ProcessResults",
    "Iterator": {"StartAt": "F1","States": {"F1": {"Type": "Task", ...,"End": true}} }}
```

Listing 7: Fan-out modeled in AWS Step Functions executing function F1 for each item in the inputElements list

Looping. AWS step functions does not directly feature a way of concisely expressing the generic function orchestration modeling construct *looping*. The `ASL Choice` state type can anyhow be used to manually construct loops: the state transitions to and from the looped unit of work can be arranged into a cycle, with the loop conditions evaluated in the `ASL Choice`state.

Delay. The generic function orchestration modeling construct *delay* can be implemented through the `ASL Wait` state type,[23] which allows delaying the execution flow for a given amount of time.

Subworkflow. The generic function orchestration modeling construct *subworkflow* is realized as a separate state machine started from within a running state machine.[23] The `ASL Task` state type can indeed be used for invoking it, by just indicating the ARN of the step functions state machine to be executed. The invocation can be synchronous, hence resulting in the main workflow waiting for the completion of the subworkflow, or asynchronous. In the latter case, a callback where the result of the subworkflow is expected can also be provided.

Error handling. The `ASL Task`, `ASL Parallel`, and `ASL Map` may specify the `Retry` and `Catch`properties.[23] `Retry` defines the retry behavior for a state for each error type, for example, maximum attempts or waiting interval between attempts. `Catch` instead specifies the state reached when no retries are available anymore, for a given error type. Execution flow may thus be redirected to any other state within the current state machine.

5.2.2 | Function orchestration in Azure durable functions

Azure durable functions enables modeling function orchestrations using the so-called *orchestrating functions* implemented in any of the programming languages supported by durable functions, that is, C#, JavaScript, or Python.[16] Essentially, an orchestrating function is a specific type of function that can be deployed to Azure functions to enable coordinating other "orchestrated types" of functions. In the following, we analyze how the generic function orchestration modeling constructs described in Table 1 can be realized in Azure durable functions using JavaScript-based examples.

Task. The generic function orchestration modeling construct *task* is realized by means of so-called `Activity Functions`. An `Activity Function` represents a basic unit of work within durable functions. Essentially, it is a specific type of functions deployed to Azure Functions and orchestrated using the Azure durable functions function orchestrator. They can be invoked by passing the `Activity Function` to run to the instruction `context.df.callActivity`.[16]

Sequence. The generic function orchestration modeling construct *sequence* is realized by chaining one or more `Activity Functions` in the function orchestration. For instance, two `Activity Function` calls can be chained one after the other, while using `yield` to await and store the output of the first call, which is then to be used as input for the subsequent one.[16] A simple example of the sequence realization for Azure durable functions is shown in Listing 8.

```
const df = require("durable-functions")
module.exports = df.orchestrator(function* (context) {
    const elements = yield context.df.callActivity("F1")
    const results = yield context.df.callActivity("F2", elements)
    return results });
```

Listing 8: Sequence modeled in Azure Durable Functions executing functions F1 and F2 with the results of F1 passed to F2

Conditional branching. The generic function orchestration modeling construct *conditional branching* is realized by using the conventional `if-else` control flow construct available in any programming language.[16] Using these standard constructs enables realizing the conditional branches and associated conditions by means of `if`, `else if`, and `else` blocks.

Parallel branching. The generic function orchestration modeling construct *parallel branching* is realized by asynchronously calling multiple `Activity Functions` (or subworkflows, modeling of which is presented later in this section) representing separate branches without using the `yield` keyword to avoid waiting for the results of each branch.[16] Instead, the task descriptors returned for each invoked branch can later be used to check whether this branch is completed and access to its results. Waiting for completion of all parallel branches and retrieving their results can be achieved by referring to their task descriptors, or by using the `yield context.df.Task.all(...)` instruction which awaits the results of all tasks in an input set—a simplified example is shown in Listing 9.

```
const df = require("durable-functions")
module.exports = df.orchestrator(function* (context) {
  const elements = context.df.getInput().elements
  const tasks = []
  tasks.push(context.df.callActivity("F1", elements))
  tasks.push(context.df.callActivity("F2", elements))
  const results = yield context.df.Task.all(tasks)
  return results });
```

Listing 9: Parallel Branching modeled in Azure Durable Functions executing functions F1 and F2 in parallel

Fan-out. The generic function orchestration modeling construct *fan-out* is realized similar to the *parallel branching*, with the only difference that the same branch is called multiple times for each element in the input data array. This is realized by exploiting conventional iterative control flow constructs to invoking the same branch (activity function or subworkflow) for different elements of the input array. The completion of all tasks in the realized *fan-out* is accomplished in the very same way as that for the *parallel branching* of tasks.[16]

Looping. As function orchestrations are implemented with imperative programming languages, the generic function orchestration modeling construct *looping* is natively supported by such languages. They indeed all feature, for example, `for` and `while` loops to repeat the same tasks until given conditions are met.[16]

Delay. The generic function orchestration modeling construct *delay* in Azure durable functions is implemented using `Durable Timers`, which enable delaying execution of tasks in a function orchestration by a given amount of time, or until a given time is reached.[16]

Subworkflow. The generic function orchestration modeling construct *subworkflow* is realized by calling separately defined and deployed Orchestrating Functions using the `context.df.callSubOrchestrator(...)` instruction.[16] These suborchestrations can be invoked synchronously, if `yield` is used to await for their results, or asynchronously. In the latter case, the task descriptor returned after the invocation of suborchestrations can later be used to check whether it is completed and access its results—a simplified example is shown in Listing 10.

```
const df = require("durable-functions")
module.exports = df.orchestrator(function* (context) {
   const input = context.df.getInput()
   const result = context.df.callSubOrchestrator("mySubOrchestration", input)
   return result });
```

Listing 10: Sub-Workflow modeled in Azure Durable Functions executing a sub-irchestration called *mySubOrchestration*

Error handling. Error handling can be realized by exploiting the constructs natively featured by the used programming language, for example, *try-catch* in JavaScript.[16] Automated retry policies can also be configured when invoking activity functions or subworflows. Such policies allow setting, for example, the maximum number of attempts or the waiting interval between attempts.

5.2.3 | Function Orchestration in Apache Openwhisk Composer

Apache Openwhisk enables deploying serverless functions as Openwhisk *actions*.[12] Such actions can then be composed in function orchestrations by exploiting the Apache Openwhisk Composer,[10] which is also used in the proprietary function orchestrator IBM Composer[31] for orchestrating functions hosted on IBM Cloud Functions FaaS platform.[32] For both, Openwhisk Composer and IBM Composer, function orchestrations can be defined as JavaScript orchestrating functions that need to be hosted on Apache Openwshik or IBM Cloud Functions, respectively. The basic building blocks for such orchestrating functions are called *combinators*, which represent common control flow constructs and are used to compose Openwhisk Actions.

Task. The generic function orchestration modeling construct *task* is realized using an Openwhisk `Action`, which represents the basic unit of work.[12] An `Action` can be invoked using the `composer.action` combinator in Openwhisk Composer.[10]

Sequence. The generic function orchestration modeling construct *sequence* is realized using the `composer.sequence` combinator, which enables chaining Actions and other combinators provided by Openwhisk Composer.[10] The `composer.sequence`'s input is passed to the first element of the sequence, which processes it and produces an output, which is in turn passed as input to the second element, and so on. The output produced by the last element in the `composer.sequence` constitutes the output of the `composer.sequence` itself. Listing 11 shows a simple Sequence executing functions named "F1" and "F2", which are assumed to be deployed on the Apache Openwhisk FaaS platform.

```
const composer = require('openwhisk-composer')
module.exports = composer.sequence('F1', 'F2');
```

Listing 11: Sequence modeled for Apache Openwhisk Composer sequentially executing functions F1 and F2

conditional branching. The generic function orchestration modeling construct *conditional branching* is realized by the `composer.if(condition, trueBranch, falseBranch)` combinator, which takes as input a condition and two alternative branches (single actions or subworkflows), one to be executed if the condition evaluates to true and the other to be executed otherwise.[10] The `condition` itself is a composition of `Actions`, which returns a JSON object with a field named `value`. If this `value` is set to `true`, the `trueBranch` is executed, whereas the `falseBranch` is executed otherwise—as shown in a simple example in Listing 12.

```
const composer = require('openwhisk-composer')
module.exports = composer.if(
  composer.action('MyCond', {action:function(params){...}}),
  'F1', 'F2');
```

Listing 12: Conditional Branching modeled for Apache Openwhisk Composer executing function F1 or F2 depending on the result returned by the MyCondition function

Parallel branching. The generic function orchestration modeling construct *parallel branching* is realized using the `composer.parallel(branch1, branch2, ...)` combinator, which enables running multiple branches (for example, single actions or subworkflows) concurrently.[10] Each branch receives a copy of the combinator's input, and the outputs generated by all parallel branches are collected into an array (a simple example is shown in Listing 13).

```
const composer = require('openwhisk-composer')
module.exports = composer.sequence(
    composer.parallel('F1','F2'),
    composer.action('Sum', {action: function(params) {
        return params.value.map(x =>x.value).reduce((a, b) => a + b, 0) }}))
```

Listing 13: Parallel Branching modeled for Apache Openwhisk Composer executing functions F1 and F2 in parallel and then summing up their results using the Sum function

Fan-out. The generic function orchestration modeling construct *fan-out* is realized using the `composer.map`, which allows executing the same combination of Actions in parallel for different inputs.[10] In particular, given a JSON object containing an array in the field `value`, a different instance of the composition specified in `composer.map` is called to process each different element in the `value` array. The outputs are then collected in an array and stored in the `value` field of the JSON return object.

Looping. The generic function orchestration modeling construct *looping* is realized by exploiting the `composer.while(condition, bodyComposition)` combinator, which takes the guard condition and `bodyComposition` to repeat at each iteration. Similarly to `Conditional Branching`, the condition itself is a composition of Actions, which returns a JSON object with a field named `value`. As long as the `value` field in such object evaluates to true, the `bodyComposition` is executed repeatedly.[10]

Delay. Openwhisk Composer currently does not provide a combinator to represent the generic function orchestration modeling construct *delay*, which could enable executing a composition of `Actions`. The desired *delay* can anyhow be realized by defining a helper function blocking the execution until a given period of time has expired, which however loads the CPU during the delay.

Subworkflow. The generic function orchestration modeling construct *subworkflow* is just any composition of `Actions` modeled as function orchestration and referenced from another function orchestration. A function orchestration created with Openwhisk Composer is treated as an Action, hence meaning that a *subworkflow* is fundamentally an `Action` itself, and that its invocation is identical to that of any other Action in a composition. For example, if we created and deployed a function orchestration named *MyOrchestration1*, we can invoke it from another function orchestration using the same combinator as for regular `Actions`, that is, `composer.action('MyOrchestration1')`.[10]

Error handling. If an error occurs in a function orchestration, an error object is returned as result, which causes the containing composition to fail.[10] The `composer.try` combinator enables realizing a try-catch construct: if the executed composition of Actions returns an error object, an error handler is invoked on such error objects. If the error handler does not return another error object, then the composition is not aborted and continues with the handler's output. Unfortunately, the `composer.try` combinator does not work on parallel executions, as the outputs (including error objects) of the parallel branches are collected in an array. In this case, the resulting array can be checked manually for error objects after all parallel branches are terminated.

5.2.4 | Summary

The analyzed function orchestrators provide native support for the majority of generic function orchestration modeling constructs in Table 1. The modeling constructs that do not have a 1:1 mapping, for example, *looping* for AWS step functions or *delay* for Openwhisk Composer, can nevertheless be implemented using workarounds, hence enabling the uniform transformation from BPMN to target orchestration formats for all analyzed generic function orchestration modeling constructs. Table 2summarizes the mapping for each generic function orchestration modeling construct to its corresponding orchestrator-specific alternative. The support differs from function orchestrator to function orchestrator, with most differences and limitations occurring on the level of distinct constructs. For instance, while Azure durable functions and Apache Openwhisk Composer natively feature *looping* constructs, AWS step functions does not have a direct mapping for it. Another example comes from the *parallel branching* and *fan-out* limitations in Azure durable functions, due to which the parallel branches must be executed as subworkflows. The same limitation instead does not apply to AWS step functions and Apache Openwhisk Composer.

5.3 | Modeling serverless function orchestrations with BPMN

As described in Section 5.1, after having identified the mappings between the generic function orchestration modeling constructs described in Table 1 and the Proprietary Function Orchestrator-specific Modeling Constructs, the similar process has to be repeated for BPMN Modeling Constructs. We hereafter introduce our uniform modeling of serverless function orchestrations in BPMN, which enables the specification of serverless function orchestrations compatible with the three orchestrators analyzed in Section 5.2. BPMN 2.0 is chosen as the basis for our uniform modeling of serverless function orchestrations for two main reasons, namely: (i) BPMN 2.0 is the well-established standard for modeling business processes, (ii) it provides a machine-readable workflow modeling language supporting the generic function orchestration modeling constructs common to the analyzed orchestrators, and (iii) it offers a visual notation that facilitates explaining produced models to nontechnical personnel.

5.3.1 | BPMN construct mapping

Our BPMN-based uniform modeling aims at fulfilling the goals outlined in Sections 1 and 2, whilst at the same time reconciling the differences and limitations of the analyzed function orchestrators. To achieve this, as shown in Figure 6, the BPMN Modeling Constructs are analyzed and mapped to the suitable generic function orchestration modeling constructs from Table 2. Essentially, this yields the subset of BPMN constructs that can be mapped fully and equally among the supported function orchestrators, which means that any serverless function orchestration model defined in this BPMN subset can be transformed into any orchestrator-specific model that can actually run on the corresponding orchestrator. At the same time, a drawback of this design decision is that unique capabilities of certain orchestrators cannot be used due to missing mappings for orchestrators lacking such capabilities (Section 5.2). This is, however, not preventing to use such orchestrator-specific features, since the generated function orchestration models can be later enhanced to exploit such features.

As a first step for modeling function orchestrations in BPMN, we need to identify the initial BPMN constructs which can be used to represent them. Unsurprisingly, the basis is given by the use of `BPMN Process` to represent one modeled serverless function orchestration and it is specified in its own model. Any number of `BPMN Sub-Processes` can be included, each expressed in their own separate model, which are then referenced and invoked from within the main BPMN workflow. A `BPMN Start Event`and a `BPMN End Event` mark the beginning and end of a modeled serverless function orchestration, respectively. There must be one `BPMN Start Event` and `BPMN End Event` per each `BPMN Process`or `BPMN Sub-Process`, so that the beginning and end of a modeled function orchestration can be identified unambiguously. By themselves, such `BPMN Start Event` and `BPMN End Event` are not mapped to any orchestrator-specific constructs. The `BPMN Start Event` indeed just marks its subsequent `BPMN Activity`, `BPMN Event`, or `BPMN Gateway` as the beginning of a modeled function orchestration, to which the inputs are passed. The `BPMN End Event` instead marks its preceding BPMN element as the end of a modeled function orchestration, whose outputs are the outputs of the orchestration itself. As the

TABLE 2 Identified mappings from generic to proprietary function orchestrator-specific modeling constructs

Generic function orchestration modeling construct	**AWS step functions**	**Azure durable functions**	**Openwhisk Composer**
Task	*ASL task* state type	*Activity function*	*composer.action* combinator
Sequence	*Next* property in states	Subsequent synchronous invocation of activities	*composer.sequence* combinator
Conditional branching	*ASL choice* state type	*if-else*	*composer.if* combinator
Parallel branching	*ASL parallel* state type	Asynchronous invocation of subworkflows	*composer.parallel* combinator
Fan-out	*ASL map* state type	Repeated and asynchronous invocation of a subworkflow	*composer.map* combinator
Looping	Explicit loops with conditions via *ASL Choice*[1]	*while*	*composer.while* combinator
Delay	*ASL wait* state type	*Durable timer*	Custom delay functions[1]
Subworkflow	Invocation of AWS step function workflow	Invocation of durable function workflow	*composer.action* combinator
Error handling	*Catch* property in states	*try-catch*	*composer.try* combinator

Note: No native modeling construct is present—requires workaround realization.

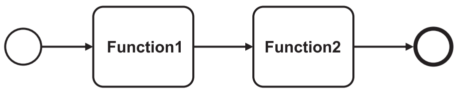


FIGURE 7 Mapping the sequence generic function orchestration modeling construct to BPMN

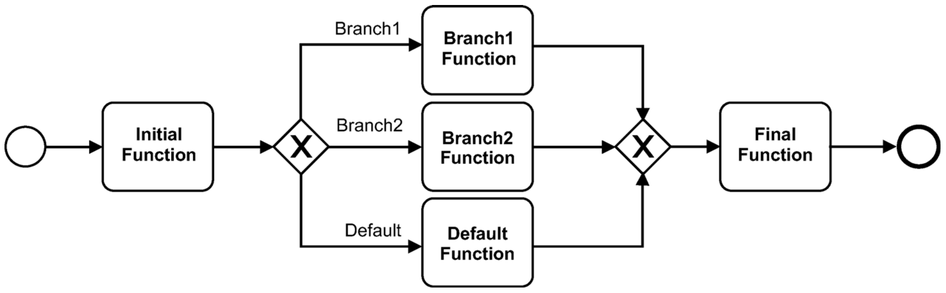


FIGURE 8 Mapping the conditional branching generic function orchestration modeling construct to BPMN

next step, we present the details on our uniform modeling in BPMN for serverless function orchestrations by discussing the mappings from BPMN Modeling Constructs to generic function orchestration modeling constructs in Table 1 separately. An actual use of our uniform modeling approach is then shown in the case study presented in Section 8.

Tasks. `BPMN Tasks` can be used to represent the generic function orchestration modeling construct *task*from Table 2. Since it represents the invocation of a serverless function, the `BPMN Task` has a mandatory `name` attribute where to specify the name of the function to invoke. It must also have exactly one incoming and one outgoing BPMN `Sequence Flow` to enable placing the function call directly in the function orchestration models.

For AWS step functions, the `BPMN Task` modeling a function call is mapped directly to a `ASL Task`state type(Section 5.2.1). In Azure durable functions and Openwhisk Composer, instead, the same `BPMN Task` is mapped to an `Activity Function` (Section 5.2.2) or to an `Action` combinator (Section 5.2.3), respectively. In all three cases, the mapping is such that the function with the specified `name` gets invoked.

Sequences. `BPMN Sequence Flows` enable connecting various BPMN activities, events, or gateways together to form a sequence (Figure 7), hence, when combined with `BPMN Tasks` they enable modeling the generic function orchestration modeling construct *sequence* from Table 2. The initial input is passed to the first element in the sequence, which processes it and passes the produced output to the second element, and so on.

A direct mapping for `BPMN Sequence Flows` exists for AWS step functions and Openwhisk Composer, namely, the `Next` property of states in AWS Step Function (Section 5.2.1) and the `composer.sequence` combinator in Openwhisk Composer (Section 5.2.3). In the case of Azure durable functions, the `BPMN Sequence Flow` is realized by sequentially invoking functions, by also using `yield` to wait for the outcomes of each function before invoking the subsequent one (Section 5.2.2).

Conditional branching. `BPMN Exclusive Gateways` enable expressing the generic function orchestration modeling construct *conditional branching* from Table 2. A `BPMN Exclusive Gateway`indeed enables executing only one among a set of branches, based on conditions associated with each branch (Figure 8). Conditions must be expressed in a uniform way and so that they can be translated to conditions supported by the analyzed function orchestrators. We hence defined a uniform modeling for conditions as well, which comes as an XSD schema alongside the currently available prototype implementation (which we later present in Section 7.2).

A direct mapping for `BPMN Exclusive Gateways` exists in all the three analyzed function orchestrators. In AWS step functions, the `BPMN Exclusive Gateway` can be realized through `ASL Choice` state type (Section 5.2.1). In Azure durable functions and Openwhisk Composer, instead, it can be realized through native `if-else` constructs (Section 5.2.2) or through the `composer.if`combinator (Section 5.2.3), respectively.

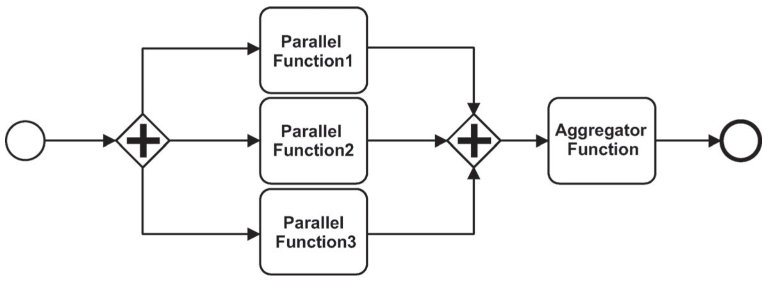


FIGURE 9 Mapping the parallel branching generic function orchestration modeling construct to BPMN

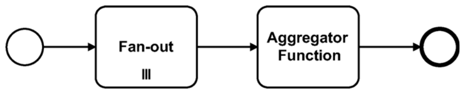


FIGURE 10 Mapping the fan-out generic function orchestration modeling construct to BPMN

Parallel branching. `BPMN Parallel Gateways` natively enable executing multiple branches in parallel (Figure 9), hence realizing the generic function orchestration modeling construct *parallel branching* from Table 2. Each branch independently operates on its own copy of the input data arriving at the opening `BPMN Parallel Gateway`. The results computed by all branches are collected together when the flow reaches the closing `Parallel Gateway`, and passed to the next workflow step.

For the *parallel branching* realized using `BPMN Parallel Gateways`, AWS step functions and Openwhisk Composer have direct mappings, namely, `ASL Parallel` state type (Section 5.2.1) and the `composer.parallel` combinator (Section 5.2.3), respectively. The parallel branches can then be any combination of the generic function orchestration modeling constructs, which will be mapped to their own corresponding Proprietary Function Orchestrator-specific Modeling Constructs. The same does not hold for Azure durable functions, as the `BPMN Parallel Gateway` structure is mapped to a sequence of asynchronously invoked activities, for example, single `Activity Functions`. The completion of all parallel tasks and retrieving their results is then achieved by using the `yield` instruction for awaiting the results of all tasks in a set. Hence, for ensuring the portability of a specified function orchestration to any of the three analyzed function orchestrators, the length of modeled parallel branches should be equal to one, with each branch being either a single `BPMN Task` or a `BPMN Sub-Process` that describes a separate function orchestration (as shown later in this section).

Fan-out. `BPMN Multi-Instance Markers` express multiple instances of a marked activity running in parallel (Figure 10), which enables realizing the generic function orchestration modeling construct *fan-out*from Table 2. Each instance operates on different elements from the original input: assuming the input to be in an array-like structure, the elements of the array are iterated and each element is processed by a different instance. When all parallel instances complete processing their elements, the results are collected into an array of results and passed to the next function orchestration step.

AWS step functions and Openwhisk Composer have direct mappings for the *fan-out* realized by `BPMN Multi-Instance Markers`, namely, `ASL Map` state type (Section 5.2.1) and the `composer.map`combinator (Section 5.2.3), respectively. In Azure durable functions, instead, the *fan-out* is realized by asynchronously invoking multiple instances of the fan-out activity in parallel, with such activity being either a single `Activity Function` or separate function orchestration (we discuss the modeling of the generic function orchestration modeling construct *subworkflow* later in this section). The completion of all parallelized tasks and retrieving their results is then achieved by exploiting the `yield` instruction for collecting the results of all tasks in a set. Notice that the limitations in realizing the *fan-out* in Azure durable functions are quite similar to those for the *parallel branching* discussed above. Similarly, it also holds that, for ensuring the portability of a specified function orchestration to any of the three analyzed function orchestrators, the fanned-out activity should be either a single `BPMN Task` or a `BPMN Sub-Process` that describes a separate function orchestration (see the discussion on *subworkflow* later in this section).

Looping. `BPMN Loop Marker` marks a BPMN activity to be repeated as long as a given condition is satisfied (Figure 11), thus, enabling to use it for modeling the generic function orchestration modeling construct *looping* from Table 2. The loop condition must be supplied in an additional `loopCondition`attribute of the marked activity (with the same uniform condition structure as for *conditional branching*). The first iteration's input is the original input, whereas the input for each other iteration is the output of the previous iteration. The output of the last iteration is then passed to the next activity in the function orchestration.

Azure durable functions and Openwhisk Composer have direct mappings for the *looping* construct realized by `BPMN Loop Markers`, namely, `while` loops (Section 5.2.2) and the `composer.while`combinator (Section 5.2.3), respectively. Instead, given that AWS step functions does not directly feature a way of concisely expressing loops (Section 5.2.1), `ASL Choice` state type is used to realize loops: the state transitions to and from the looped unit of work can be arranged into a cycle, with the loop conditions being evaluated in `ASL Choice` state type.

Delay. `BPMN Timer Intermediate Catching Events` with one incoming and one outgoing `BPMN Sequence Flow` (Figure 12) are used for modeling the generic function orchestration modeling construct *delay* from Table 2. The `BPMN Timer Intermediate Catching Event` must include the `milliseconds` attribute, which specifies how long the running function orchestration should delay before proceeding with the subsequent orchestration step.

AWS step functions and Azure durable functions have a direct mapping for `BPMN Timer Intermediate Catching Events`, namely, `ASL wait` state type (Section 5.2.1) and `Durable Timers` (Section 5.2.2), respectively. Openwhisk Composer instead does not feature any mapping for the *delay* realized by `BPMN Timer Intermediate Catching Events` (Section 5.2.3), which can anyhow be realized via helper functions blocking the execution for a given period of time.

Subworkflow. `BPMN Subprocesses` enable representing the generic function orchestration modeling construct *subworkflow* from Table 2. Each `BPMN Subprocess` represents a self-contained function orchestration with its own `Start Event` and `End Event` (Figure 13). A `BPMN Subprocess` is similar to the `BPMN Task` concerning its placement into a function orchestration and its additional attributes, for example, it has one incoming and one outgoing `BPMN Sequence Flow`, a `name` attribute to specify the name of the function orchestration to be invoked as subworkflow, and it can be marked with `BPMN Multiinstance Marker` and `BPMN Loop Marker` to run multiple instances of the subworkflow or to repeat it until a given condition is satisfied.

When porting a modeled function orchestration to any of the analyzed orchestrators, each *subworkflow* is transformed in its own separate orchestrator-specific function orchestration model, which is then invoked by the orchestrator-specific main function orchestration. This is done in AWS step functions by mapping the `BPMN Subprocess` to the `ASL Task` state type, which actually performs the invocation of the Step Function realizing the function orchestration in a *subworkflow* (Section 5.2.1). In Azure durable functions, a similar solution is obtained by calling the subworkflow with the `callSubOrchestrator` instruction (Section 5.2.2). Finally, in Openwhisk Composer, a `BPMN Sub-Process` directly maps to an `Openwhisk Action` (Section 5.2.3).

Error handling. `BPMN Error Boundary Events` are applied to `BPMN Tasks` or `Sub-Processes`to catch errors they may raise, thus, enabling to model *error handling* The `BPMN Error Boundary Event` has one outgoing `BPMN Sequence Flow`, which leads to the branch executed if an error occurs (Figure 14). `BPMN Error Boundary Events` handling possible errors have direct mappings in all the three analyzed orchestrators. These are `ASL Catch`

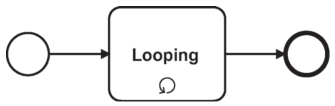


FIGURE 11 Mapping the looping generic function orchestration modeling construct to BPMN

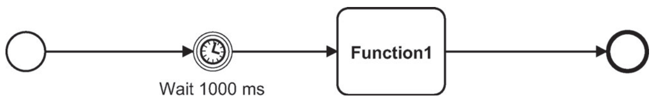


FIGURE 12 Mapping the delay generic function orchestration modeling construct to BPMN

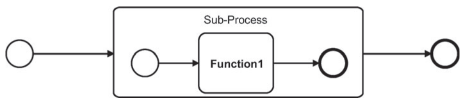


FIGURE 13 Mapping the subworkflow generic function orchestration modeling construct to BPMN

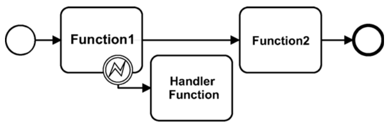


FIGURE 14 Mapping the error handling generic function orchestration modeling construct to BPMN

state types for AWS step functions (Section 5.2.1), `try-catch` blocks for Azure durable functions (Section 5.2.2), and `composer.try` combinators for Openwhisk Composer (Section 5.2.3).

5.3.2 | BPMN modeling restrictions for portable function orchestrations

Our uniform modeling for serverless function orchestration aims to be portable and to support the same common subset of capabilities of the function orchestrators as this enables transforming any modeled function orchestration to any supported orchestrator-specific model format. For this purpose, the mappings from the previous section are discussed in more detail and restricted to express only common capabilities that can be transformed to all proprietary formats discussed in this article.

BPMN process requirements. One requirement for the `BPMN Process` is that a `BPMN End Event`marks the end of the modeled function orchestration and occurs only once—this enables unambiguously generating target function orchestration models. An exception to this is when using `BPMN Error Boudary Events`, as error handling branches starting from such events may specify additional `BPMN End Event` to denote their termination.

For transforming a specified workflow to AWS step functions, the `BPMN Process` might include optional attributes `awsAccountId` and `awsAccountRegion`, which are needed to derive valid ARNs to the corresponding AWS lambda functions modeled as named `BPMN Tasks`. These attributes can also be left blank—and the model for AWS step functions will then be generated without function references.

BPMN element input and output. The input and output of states and function orchestrations in AWS step functions is given as JSON objects Section 5.2.1. We apply this concept also to our uniform modeling of serverless function orchestrations, assuming the data moving along the `BPMN Sequence Flows` to be JSON objects. This is only a minor restriction given that Azure durable functions and Openwhisk Composer natively support the input/output of JSON objects (Sections 5.2.2 and 5.2.3).

BPMN parallel gateway and multiinstance marker requirements. Handling errors that occur in parallel branches is not equally supported among the analyzed function orchestrators. AWS step functions and Azure durable functions allow handling errors occurring in parallel branches and fanned-out activities (Sections 5.2.1 and 5.2.2). Openwhisk Composer instead does not allow directly handling errors occurring in parallel/fanned-out activities run through the `composer.parallel` or `comoposer.map`combinators (Section 5.2.3). For this reason, to ensure the porting to all analyzed function orchestrators, no error handling in parallel can be specified, namely, `BPMN Error Boundary Events` must not be attached to activities running as parallel branches nor to activities marked with `BPMN Multi-Instance Markers`.

Openwhisk Composer imposes requirements on the input/output of parallel branches specified with `BPMN Parallel Gateways` and on activities fanned-out with `BPMN Multi-Instance Markers`. In particular, their output is assumed to be collected in an array and placed in the `value` field of a JSON object. The latter is the format prescribed by Openwhisk Composer's `composer.parallel` and `composer.map` combinators. The same format is expected as input for activities marked with `Multi-Instance Markers`, since the `composer.map` combinator expects an input structured according to such format.

BPMN markers requirements. The use of `BPMN Multi-Instance Marker` and `BPMN Loop Marker` is limited within the use of our uniform modeling. Indeed, to enable unambiguously generating target function orchestration models, only one marker can be applied to the same BPMN element at a time, for example, if a `BPMN Multi-Instance Marker` is applied to a `BPMN Task`, a `BPMN Loop Marker` can no longer be applied to the same `BPMN Task`. Although loops in BPMN can also be modeled, for example, using `BPMN Tasks` and `BPMN Conditional Gateways`, in our uniform modeling and transformation approach we only support the `BPMN Loop Marker` for modeling the generic function orchestration modeling construct *looping*.

6 | TOSCA-BASED FUNCTION ORCHESTRATION DEPLOYMENT MODELING

Similar to function orchestration modeling, deployment modeling for function orchestrations requires considerable technology-specific expertise and is hard to accomplish in a portable manner as described by Challenge 2 in Section 2.2. Therefore, in this section, we present our TOSCA-based approach for modeling the deployment of serverless function orchestrations. In our approach, we aim to cover deployment modeling of function orchestrations as independent applications and as parts of larger serverless applications, thus, requiring us to consider not only the function orchestration deployment modeling aspects, but also the specifics of modeling event-driven serverless applications in TOSCA. To achieve this, we use our previous work[24] as a basis—while documenting initial decisions for modeling serverless functions and event-triggering semantics in TOSCA, our previous work does not cover the specifics of function orchestration modeling.

Therefore, the contributions of this section are as follows. We (i) revise and extend our previous work by presenting additional modeling alternatives related to modeling of serverless functions and event-triggering semantics which were not present in the original work, and as a novel contribution (ii) we introduce the concepts for modeling function orchestrations independently and as parts of serverless applications with TOSCA. To illustrate the core aspects of our modeling approach, we model excerpts of the serverless deployment architecture presented in Section 2 (see Figure 2) using the AWS cloud infrastructure as an example.

Furthermore, as a proof of concept, we provide a *revised hierarchy of TOSCA types* (implemented using the YAML-based TOSCA Simple Profile v1.3 specification) to enable using our function orchestration modeling approach, which is publicly available on Github.[50] This hierarchy of TOSCA types comprises a set of abstract and concrete TOSCA types. The former represent generic component types such as *Function* or *ObjectStorage*, and do not reference any particular technologies such as AWS Lambda or Azure durable functions. The latter are technology-specific, deployable TOSCA modeling constructs for deploying function orchestrations to three public cloud infrastructures—AWS, Microsoft Azure, and IBM Cloud. While function orchestration modeling and transformation described in Section 5 is supported for both IBM Composer and Apache Openwshik, we focused on supporting the deployment of function orchestrations to IBM Composer by providing the corresponding IBM-specific TOSCA node types since the IBM cloud infrastructure comprises a variety of serverless offerings, for example, object storage and messaging services, which is typically not the case for the self-hosted deployments.[5] Since IBM Composer is based on the Apache Openwhisk Composer and orchestrates functions hosted on IBM Cloud Functions, which is in turn based on Apache Openwhisk, our TOSCA types can be adapted for deploying to Apache Openwhisk and Openwhisk Composer.

6.1 | Modeling the deployment of functions, event emitters, and event bindings

The deployment architecture presented in Figure 2 shows how a serverless function orchestration can be combined with standalone serverless components, for example, the notification function, message queue, and timer event binding for calling the function orchestration on a scheduled basis. Thus, to model such deployment architectures in TOSCA, apart from the function orchestration aspects, event-driven deployment aspects must also be reflected in the model. Unsurprisingly, all components, for example, functions (orchestrated and standalone), object storage buckets, and message queues, must be deployed and configured. This also includes configuring required event-bindings between components after the components are deployed, for example, a binding between the *results* bucket and the *notify* function in Figure 2.

6.1.1 | Modeling the hosting of serverless components

Figure 15 shows two variants of modeling the hosting for serverless components: (a) the *per service configuration* approach from our previous work,[24] and (b) an alternative added in our revised TOSCA types hierarchy. In our previous work, we described modeling of serverless components on a per service basis, for example, functions are hosted on a FaaS platform and buckets are hosted on an object storage service using the normative relationship type *hostedOn* (see Figure 15A). Essentially, since the actual serverless services are *always on* and do not need to be deployed, the node types representing them group respective configurations, for example, access policies. Since typical serverless applications are fine-grained and comprise multiple components sharing similar configurations, in the revised version of our TOSCA types hierarchy, we use another modeling alternative to reduce the amount of hosting nodes, and hence, the redundancy in the model. More specifically, we introduce generic node types representing an *ecosystem of services* can be used, for example, the *AwsPlatform* node type which inherits from the abstract *CloudPlatform* node type enables grouping shared configurations for multiple components, for example, functions deployed in the same region, and reducing unnecessary visual clutter in the model as shown in Figure 15B. Since TOSCA natively supports the ontological extension of component types, no linguistic changes of TOSCA such as introducing nonstandardized constructs or features were needed for our approach. Hence, we utilize TOSCA as-is for creating our type system using the ontological extensibility of the standard. As a result, all introduced TOSCA constructs are TOSCA-compliant and can be processed by TOSCA-compliant orchestrators.

6.1.2 | Modeling serverless components and their properties

In our previous work,[24] we modeled FaaS functions as node templates of the corresponding platform-specific node type such as *AwsLambdaFunction*, *AzureFunction*, or *IbmCloudFunction*. To generalize this, in the revised TOSCA types hierarchy introduced in this article, we add abstract node types from which all technology-specific node types are derived, that is, the abstract node type *function* is the direct parent of all FaaS-specific node types. Properties present in all platform-specific node types such as *name* reside in the abstract *function* node type, while the platform-specific properties reside in corresponding platform-specific node types. For modeling other serverless components such as message queues and object storage buckets we also use this revised approach, for example, technology-specific node types such as *AwsS3Bucket* shown in Figure 15inherit from the abstract the *ObjectStorage* node type which groups only common properties. Moreover, while in our previous work[24] we modeled function runtimes as separate node types (see *AwsLambdaNodeJSFunction* in Figure 15A), in our revised TOSCA types hierarchy we model runtimes as properties as shown in Figure 15B—this helps reusing the same node types for many runtimes, for example, *AwsLambdaFunction* node type for "Java8" and "node.js12" runtimes.

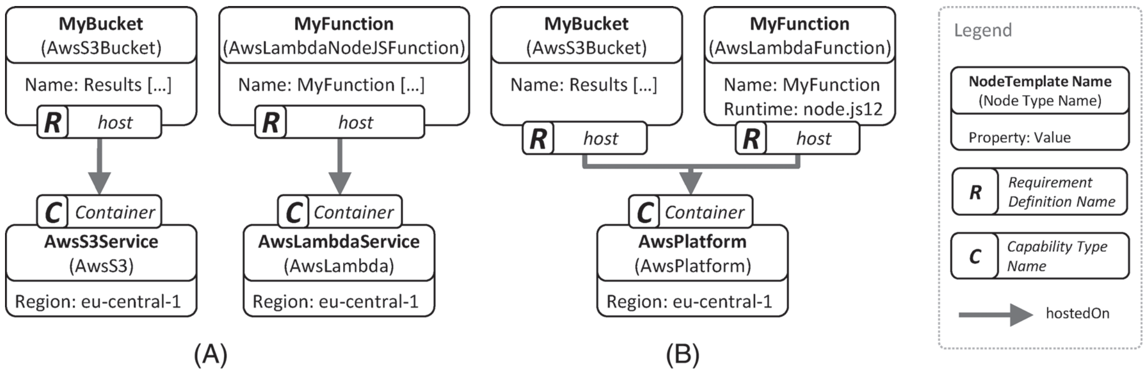


FIGURE 15 Modeling hosting of serverless components: (A) per service configuration and (B) shared configuration

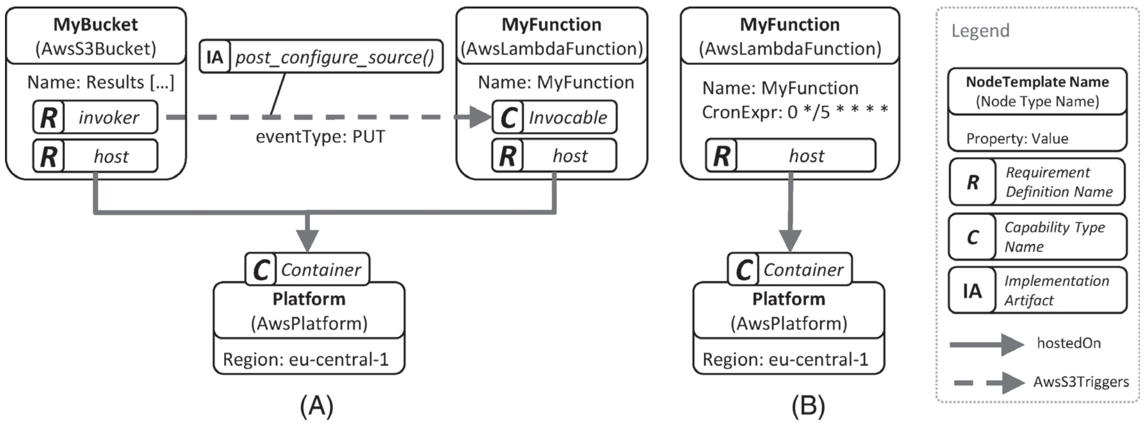


FIGURE 16 Modeling event-driven semantics with TOSCA: (A) explicit and (B) implicit event flows

6.1.3 | Modeling event-driven semantics

Typically, serverless FaaS-based applications rely on binding functions to events emitted by different kinds of components, for example, databases, message queues, or logging services. As in our previous work,[24] we model event-driven semantics using the technology-specific derivatives of the abstract relationship type *triggers*, for example, *AwsS3Triggers* relationship type shown in Figure 16A. Note, that the abstract relationship type *triggers* is derived from the normative TOSCA *dependsOn* relationship type to enforce the correct deployment order—functions must be deployed before configuring event bindings with a given event emitter. The matching of this relationship happens using the requirement definition *invoker* and the capability type *invocable*, which can easily be used in other kinds of serverless components, for example, containers hosted on serverless container hosting services such as AWS Fargate can be *invocable*, too. Event types that trigger the function are modeled as properties of the relationship template, for example, as custom TOSCA data types following the CloudEvents specification, or using the primitive *string* data type—*eventType* property is a string "PUT" in Figure 16A. Next, the deployment logic for establishing event bindings is attached to relationships as TOSCA Implementation Artifacts that connect event emitters with FaaS functions. In terms of lifecycle operations, the actual execution leverages normative relationship operations such as *post_configure_target* or *post_configure_source*. For example, the *post_configure_source* normative operation can be used to establish a binding between a bucket and a function as it is invoked only after both of the components are deployed. As an additional enhancement, to introduce modeling restrictions, the relationship type fields *valid_source_types* and *valid_target_types* can be defined to allow using only certain relationship types between nodes. Finally, in the revised version of our TOSCA types hierarchy introduced in this article, we show an alternative variant for modeling event-driven semantics—implicit event flows. For example, for timer-based function invocations the event binding logic can be represented implicitly as a part of the function as shown in Figure 16B. The actual timer expression, for example, the cron expression to trigger a function every five minutes (see *CronExpr* property in Figure 16B), is then modeled as a property of the function. Thus, the event binding logic for such implicit modeling is triggered as an extra step during the actual deployment of the Node Template instead of calling a separate operation on the Relationship Template.

6.2 | Modeling deployments of standalone function orchestrations

In this section, we present our new approach for modeling standalone function orchestrations in TOSCA using the aforementioned ETL function orchestration scenario from Figure 1 deployed to AWS as an example. As seen in the motivating example in Section 2, apart from regular serverless components, a function orchestration model must be deployed to a compatible function orchestrator, for example, ASL model for AWS step functions. Figure 17 shows how standalone function orchestrations can be modeled in TOSCA. Similar to regular serverless components, we represent the deployment of function orchestration models as node templates of the corresponding technology-specific node type such as *AwsSFOrchestration*, *AzureDFOrchestration*, or *IbmComposerOrchestration*, which all inherit from the abstract node type *workflow*. The abstract *workflow* node type represents a

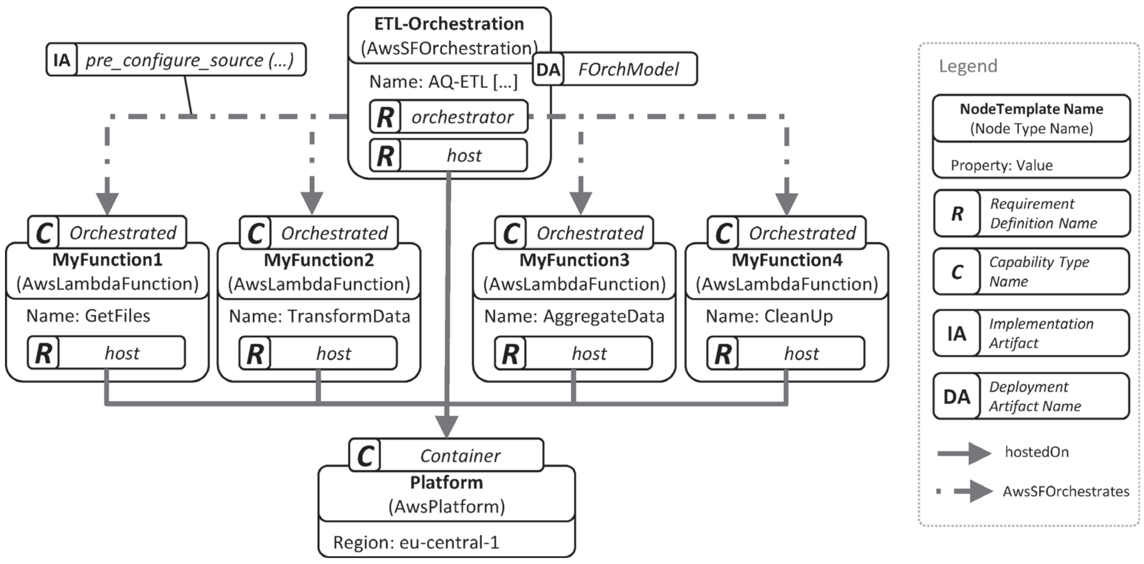


FIGURE 17 Modeling serverless function orchestrations with TOSCA

generic workflow model, which serves as a parent type for technology-specific constructs representing deployments to function orchestrators or general-purpose workflow engines. Similar to other serverless components including the orchestrated functions, the hosting semantics for function orchestration models is represented via the normative *hostedOn* relationship type—in our example we use the shared configuration modeling via the *AwsPlatform* node type.

Furthermore, we model the orchestration semantics using the technology-specific derivatives of the abstract relationship type *orchestrates*, for example, *AwsSFOrchestrates* relationship type shown in Figure 17. Note, that the abstract relationship type *orchestrates* is derived from the normative TOSCA *dependsOn* relationship type to enforce the correct deployment order—functions must be deployed before deploying the function orchestration model. The matching of source and target nodes with this relationship is achieved using the Requirement Definition *orchestrator* and capability type *orchestrated*. As in event-driven relationships, these capabilities can be modeled in other kinds of components in case they are supported by the orchestrator, for example, AWS step functions enables orchestrating containers hosted on AWS Fargate. The actual function orchestration model defining the control flow for orchestrated functions is attached as a deployment artifact to the *AwsSFOrchestration* node type. The Implementation Artifact enabling configuration of the function orchestration model is attached to the *AwsSFOrchestrates* relationship type as shown in Figure 17. The required configuration depends on the target technology, for example, function orchestration models in AWS step functions require specifying the Amazon resource names (ARNs) of orchestrated functions meaning that after deploying each function, the *pre_configure_source()* Implementation Artifact must be executed for each corresponding relationship to update the function orchestration model with the correct ARN.

6.3 | Modeling the deployment of function orchestrations as parts of serverless applications

To enable incorporating function orchestrations in larger application models, the concepts introduced in Sections 6.1 and 6.2 can be combined. To present how we combine these concepts, we discuss the modeling of serverless deployment architecture from Figure 2 using AWS services as an example. This example combines both modeling aspects: a function orchestration model and regular serverless components that interact in event-driven fashion. We assume that the external components (containerized application and public dataset bucket) are already deployed by third parties and the access is configured in the business logic, hence, omitting them from our model. We provide more implementation details when discussing the case study in Section 8, in which we implement and deploy this deployment architecture from Figure 2 for three cloud providers: AWS, Azure, and IBM.

Essentially, the concepts described in Section 6.1 can be easily applied to the *workflow* node type and its concrete derivatives such as *AwsSFOrchestration*, too. For example, as discussed in Section 6.1.3, the timer-based invocation of function orchestrations can be modeled either as explicitly using the derivatives of *triggers* relationship type or implicitly in node templates—timer expressions are specified as properties of the function orchestration model with the deployment logic enabling configuration of such event bindings. Furthermore, normative TOSCA relationships such as *connectsTo* can be used to model other components and combining them with the technology-specific derivatives of the abstract *triggered* relationship type for deploying and configuring components of the application that are not a part of the function orchestration model.

Figure 18 shows how using the concepts described in Sections 6.1 and 6.2 the motivating scenario introduced in Section 2 can be modeled in TOSCA to deploy it on the AWS infrastructure. The resulting application topology is a directed typed graph in which nodes represent technology-specific components such as *AwsLambdaFunction* and *AwsSQSQueue*, and edges represent the interaction types among these components, for example, even-driven invocation, orchestration, or hosting relationships.

When enacting this model, the overall deployment procedure would happen as follows. As a first step, shared configurations specified using the *AwsPlatform* node type are performed. Next, the message queue is created using the *AwsSQSQueue* node type and the standalone notification function is deployed using the *AwsLambdaFunction* node type. Afterwards, the object storage bucket is deployed using the *AwsS3Bucket* node type and the corresponding event binding between the bucket and the notification function is created after both components are successfully deployed. Furthermore, four orchestrated function are deployed using the same *AwsLambdaFunction* node type, with runtimes and other required properties defined in each function to avoid introducing too fine-grained node types, for example, separate node type per runtime. When all orchestrated functions are deployed, the ASL function orchestration model can be deployed using the *AwsSFOrchestration* node type, with the timer-based invocation being modeled implicitly using the *CronExpr* property. At deployment, the corresponding *Amazon even bridge rule* is created and bound to the deployed function orchestration model, hence enabling the timer-based invocation. As a result, using a combination of event-driven and orchestration modeling specifics, the motivating example can be successfully represented in TOSCA and deployed using a TOSCA-compliant deployment automation technology.

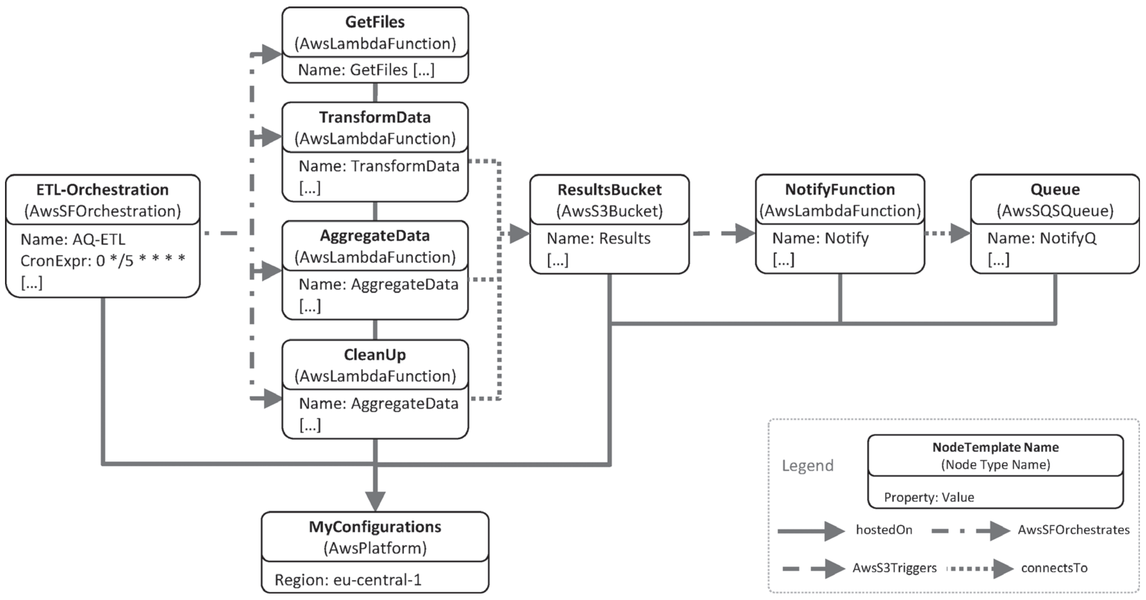


FIGURE 18 A TOSCA-based modeling of the serverless deployment architecture shown in Figure 2 for AWS

7 | A FUNCTION ORCHESTRATION MODELING AND DEPLOYMENT TOOLCHAIN

In this section, we show how our method presented in Section 4 can be enabled using an open source toolchain. We first provide an overview how each step of the method from Section 4 is mapped to a particular open source tool and then discuss the overall way of chaining the tools together. Next, we present each involved tool in more details and, when applicable, elaborate on our contributions related to implementation of these tools.

7.1 | A toolchain for standards-based modeling and deployment of function orchestrations

Following our method described in Section 4, both modeling approaches introduced in Sections 5 and 6can be combined in an end-to-end scenario using dedicated BPMN and TOSCA modeling tools. In such end-to-end scenario, in *Step 1*, a generic BPMN function orchestration model is created using our BPMN for function orchestrations (BPMN4FO) tool that enables our uniform function orchestration modeling approach presented in Section 5. Moreover, BPMN4FO includes a BPMN transformation layer for generating the function orchestration model formats for AWS step functions, Azure durable functions, Apache Openwhisk Composer, and IBM Composer. Then, the resulting provider-agnostic BPMN function orchestration model is used to generate supported target model formats, for example, an ASL definition for AWS step functions (*Step 2*). Afterwards, the overall deployment of the intended function orchestration model, for example, as a standalone orchestration of functions, or as a part of a larger serverless application as described in Sections 5 and 6, is modeled using TOSCA (*Step 3*). The resulting TOSCA model (service template definition) is also enriched with the deployment artifacts such as functions code and the provider-specific workflow definition generated using the BPMN4FO. We employ Eclipse Winery,[27] a well-known graphical modeling tool for creating deployment models in TOSCA, for representing deployments of function orchestrations. In *Step 4*, a CSAR export is triggered using Eclipse Winery to generate a self-contained deployment package which is then consumed by xOpera to enact the deployment (Step 5). Since our TOSCA types introduced in Section 6do not utilize any nonstandard TOSCA constructs or features, the generated self-contained deployment packages are standard-compliant. Therefore, we employ xOpera[26] for our toolchain without any modifications as a TOSCA-compliant deployment automation technology. Note, that the overall process is not required to be sequential, as tools can be reused for model refinement, adding new versions of existing applications, or using created models separately, for example, a BPMN-based function orchestration model can be used as a high-level overview of the intended function orchestration. Figure 19 shows the overall process of chaining these tools together.

7.2 | Tool support for modeling and transformation of function orchestrations with BPMN

To enable specifying portable serverless function orchestration models with our uniform function orchestration modeling approach introduced in Section 5, we implemented the prototype for modeling and transformation of serverless function

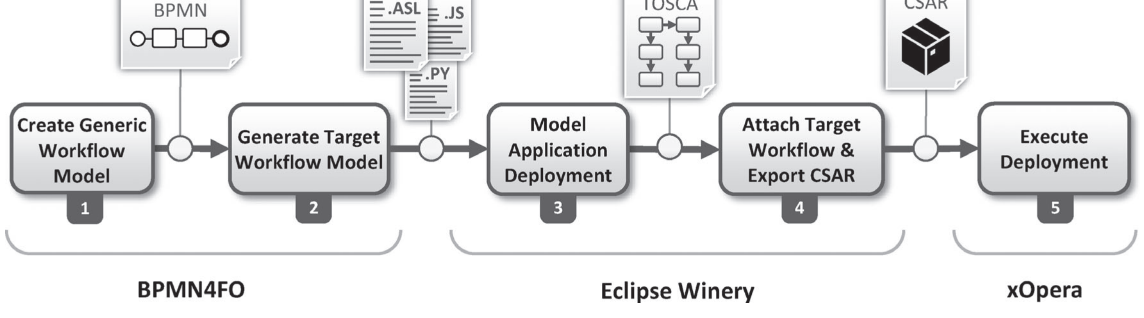

BPMN
.ASL
.JS
.PY
TOSCA
CSAR
Create Generic Workflow Model
1
Generate Target Workflow Model
2
Model Application Deployment
3
Attach Target Workflow & Export CSAR
4
Execute Deployment
5
BPMN4FO
Eclipse Winery
xOpera

FIGURE 19 A toolchain for standards-based modeling and deployment of function orchestrations

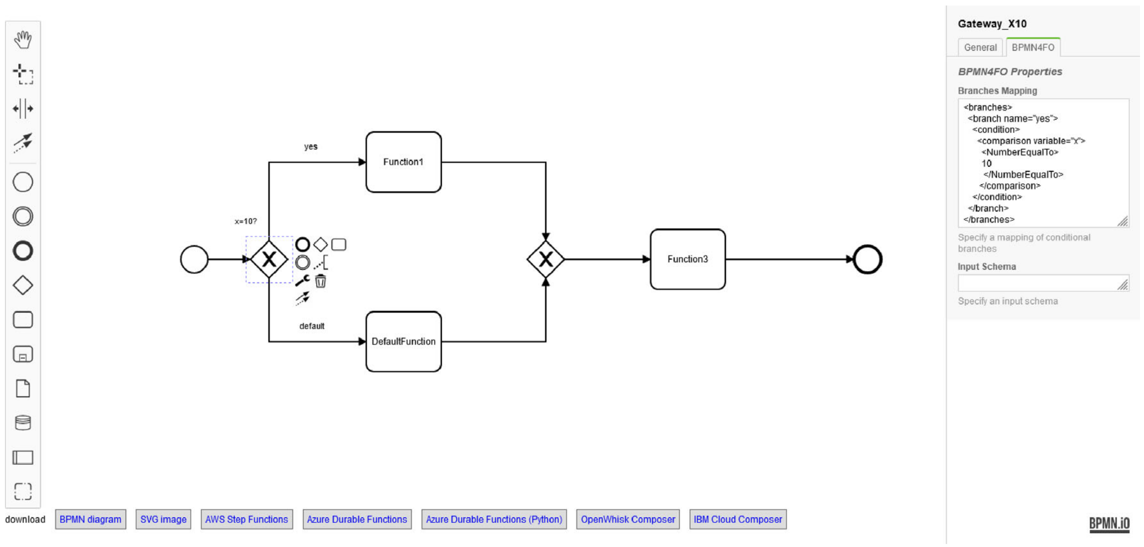
orchestrations called BPMN4FO. BPMN4FO is a JavaScript-based client-side web application that enables (i) creating and editing serverless function orchestrations using our BPMN-based uniform modeling approach and (ii) transforming the resulting *BPMN function orchestration models* into orchestrator-specific model formats that can be executed with the supported function orchestrators, for example, ASL models for AWS step functions. BPMN4FO is open source and publicly available on GitHub (see https://github.com/iaas-splab/matoswo#bpmn4fo), also with a demo instance available via GitHub Pages for this repository.

7.2.1 | Graphical function orchestration modeling with BPMN4FO

Figure 20 provides a snapshot of the graphical editor implemented in BPMN4FO based on the bpmn-js library. The central pane of the editor provides modeling area, where BPMN function orchestration models are visually edited. BPMN function orchestration models may be imported by dragging-and-dropping BPMN XML files to the modeling area, or they can be created from scratch by dragging and dropping BPMN elements from the pane on the left hand side. The right hand side of the editor instead enables setting the properties of each BPMN element in the model, for example, the element name, or custom properties such as conditions for `BPMN Conditional Gateways` included in our uniform function orchestration modeling approach. The available properties actually depend on the selected model element type. For instance, if a `BPMN Exclusive Gateway`is selected (as in Figure 20), the branches property holding the mapping from conditions to associated branches, displayed as `Branches Mapping`, becomes available. Finally, the buttons at the bottom of the editor enable exporting a modeled BPMN function orchestration model in XML format or as a vector image, as well as transforming and downloading the orchestrator-specific function orchestration model formats that can be run with supported function orchestrators, namely, AWS step functions (ASL models), Azure durable functions (JavaScript- and Python-based models), Apache Openwhisk Composer (JavaScript-based models) and IBM Composer (JavaScript-based models). For each supported function orchestrator, the orchestrator-specific function orchestration model representing the created BPMN function orchestration model workflow is downloaded as ZIP archive. When a created BPMN model is invalid, for example, added property is not supported by a function orchestrator, the buttons generating orchestrator-specific model on incompatible orchestrators will be disabled and an error message will be written to the console.

7.2.2 | Transformation of modeled function orchestrations with BPMN4FO

Apart from graphical modeling, BPMN4FO supports transforming BPMN function orchestration models into orchestrator-specific model formats. The architecture of BPMN4FO is shown in Figure 21. The *function orchestration*



```
<branches>
  <branch name="yes">
    <condition>
      <comparison variable="x">
        <NumberEqualTo>
        10
        </NumberEqualTo>
      </comparison>
    </condition>
  </branch>
</branches>
```

FIGURE 20 The graphical BPMN editor in BPMN4FO

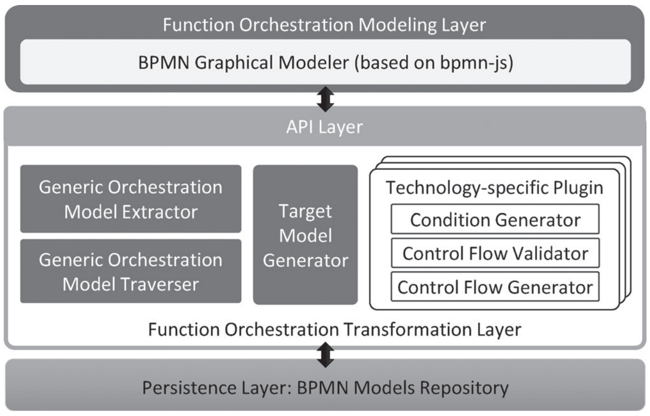

Function Orchestration Modeling Layer
BPMN Graphical Modeler (based on bpmn-js)
API Layer
Generic Orchestration Model Extractor
Generic Orchestration Model Traverser
Target Model Generator
Technology-specific Plugin
Condition Generator
Control Flow Validator
Control Flow Generator
Function Orchestration Transformation Layer
Persistence Layer: BPMN Models Repository

FIGURE 21 The conceptual architecture of BPMN4FO

modeling layer has a *BPMN graphical modeler* component that provides the graphical user interface in Figure 20. As mentioned before, it is realized by extending a production-ready BPMN 2.0 web modeler by Camunda called `bpmn-js`.[51] We chose `bpmn-js` (see https://bpmn.io/) since it is already used in industry and since it natively features import/export of BPMN models to facilitate their storage. With `bpmn-js`, BPMN function orchestration models can indeed be saved and loaded from a local file system, hence, the *persistence layer* in our implementation relies on the local file system.

The buttons available in the *BPMN graphical modeler* component enable generating runnable orchestrator-specific function orchestration models by interacting with the *function orchestration transformation layer* via the *API layer*. We implemented BPMN4FO as a frontend-only monolithic application, therefore, all the exposed functionalities are invoked via the corresponding visual elements in the graphical modeling interface. The transformation of the BPMN function orchestration model into the desired orchestrator-specific formats is performed by coordinating the four main components in the *function orchestration transformation layer* as shown in Figure 21, namely, *generic orchestration model extractor*, *generic orchestration model traverser*, *target model generator*,and a set of *technology-specific plugins*. Firstly, given the user-created BPMN function orchestration model, the corresponding *generic orchestration model extractor* component generates an internal, language-agnostic model describing each construct and the control flow in the BPMN function orchestration model, which is used for further processing. Some basic validation is also performed at this stage, for example, verifying the correct number of input and output `BPMN Sequence Flows` for the modeled elements. This generic model extracted from the created BPMN model is then used for generating target function orchestration models using the *target model generator*, *generic orchestration model traverser* components and the corresponding *technology-specific plugin*, for example, AWS-specific plugin for generating ASL models for AWS step functions. Essentially, these generic control flow models are traversed and processed by the *target model generator* using the corresponding *technology-specific plugin* which comprises three major subcomponents: *condition generator*, *control flow validator*, and *control flow generator*. The *control flow validator*, and *control flow generator* sub-components are responsible for traversing the control flow model and ensuring that the control flow hierarchy is valid according to our uniform function orchestration modeling approach w.r.t. the chosen target format. The *control flow generator* transforms the control flow hierarchy into the desired orchestrator-specific function orchestration model format based on the traversal results. When encountering conditional control flow constructs, representations of the actual conditions must be generated according to each specific orchestrator. For this purpose, the *condition generator* is invoked by the *target model generator* when needed. the *condition generator* provides orchestrator-specific condition generation logic, which traverses the modeled conditions and output their corresponding orchestrator-specific representation, for example, for ASL models. Notice that the transformation logic is realized in a pluggable manner: to add support for more orchestrators than those currently supported, additional technology-specific plugins comprising corresponding generators can be implemented.

7.3 | Tool support for modeling and deployment of function orchestrations with TOSCA

To enable modeling of serverless function orchestrations with TOSCA using our approach described in Section 6, we use the graphical modeling tool Eclipse Winery,[27] which is a part of the OpenTOSCA ecosystem,[46] and which provides user interfaces for modeling TOSCA application topologies graphically. Eclipse Winery provides a management GUI for defining TOSCA constructs such as Node and relationship types, service templates, and so forth. Furthermore, Winery provides a topology modeling GUI, which enables creating application topologies graphically. The TOSCA types hierarchy implemented according to the approach in Section 6 is imported into Eclipse Winery to create deployment models for function orchestrations. After the desired deployment models are ready, they can be exported as deployable CSARs and processed by a TOSCA-compliant orchestrator such as xOpera. We provide examples of modeled function orchestrations in Eclipse Winery in the case study in Section 8.

Further, to enable the automated deployment of TOSCA models, we employ (as-is) an open source and lightweight TOSCA orchestrator called xOpera.[26] xOpera supports the YAML-based TOSCA v1.3 specification[22] and relies on Ansible for implementing the deployment logic. Essentially, we provide the CSARs produced by Winery as an input for the CLI version of xOpera, which enacts the deployment of serverless applications modeled using concrete, provider-specific TOSCA types. To enable the deployment on public cloud providers, xOpera requires specifying credentials data locally on the device used for enacting the deployment, more information on how to use it is in the official documentation (see https://xlab-si.github.io/xopera-docs).

8 | CASE STUDY

In this section, we present how our method for standards-based modeling of function orchestrations introduced in Section 4 and relying on the two modeling approaches described in Sections 5 and 6 can be used in the context of implementing the serverless application from our motivating scenario (Section 2). This scenario comes from an existing, third-party function orchestration[28] for processing the OpenAQ dataset (see https://registry.opendata.aws/openaq) that contains aggregated physical air quality data from public data sources regularly provided by government, research-grade, and other sources. The resulting ETL function orchestrations generate the minimum, maximum, and average ratings for air quality measurements on a daily basis and store them in a corresponding cloud object storage bucket. The toolchain presented in Section 7 is employed for modeling the function orchestration and its deployment. All produced artifacts are open source and available on GitHub.[50]

8.1 | Modeling and transformation of the air quality data ETL function orchestration

In *Step 1* of our method described in Section 4, we need to model the ETL function orchestration for processing air quality data in BPMN. We create this BPMN function orchestration model as a `BPMN Sequence`, which starts with the invocation of the *ListFiles* function to collect the file paths from the previous day in chunks. Afterwards, since the *TransformData* function must be instantiated in parallel for each chunk, we model it using `BPMN Task` with a `BPMN Multi-Instance Marker`, hence representing the generic function orchestration modeling construct "*Fan-out*". This *fan-out* activity maps one chunk to a separate processor that downloads the files listed in this chunk and transforms them into an intermediary format. As a next step of this `BPMN Sequence`, the *AggregateData* function modeled as `BPMN Task` reduces all intermediary files into one final result. Finally, the *CleanUp* function modeled as `BPMN Task` deletes all intermediary files and reports about completing the function orchestration. Figure 22 shows the resulting model of the BPMN function orchestration model created as described in Section 5. Next, in *Step 2* of our method described in Section 4), we transform the BPMN function orchestration model into three target formats using our BPMN4FO described in Section 7. While the resulting formats share the same control flow structure and function

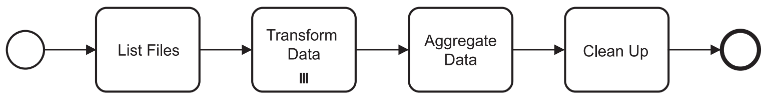


FIGURE 22 BPMN function orchestration model for processing air quality data from Section 2

names, it can be observed that the output for the chosen function orchestrators differs significantly. Next, we elaborate on how the BPMN function orchestration model is transformed into ASL definition for AWS step functions, a Python-based orchestrator function for Azure durable functions, and a JavaScript-based function composition for IBM Composer.

Transforming the BPMN function orchestration model for AWS step functions. The transformed function orchestration for processing air quality data for AWS step functions is shown in Listing 14—it is an ASL model describing the desired control flow as described in Section 5, with some details omitted from the listing for the sake of brevity. The ASL definition needs to reference corresponding ARNs of the orchestrated functions, which means that the deployment information is needed. To enable the automated deployment of this workflow, ARNs of functions can be specified directly in our prototype in case already deployed functions are planned to be used. However, if the orchestrated functions are deployed together with the function orchestration model, their ARNs are substituted in the ASL definition after the functions are deployed. This substitution of ARNs happens when xOpera deploys the entire application model and invokes the `pre_configure_source` operation on each SFOrchestrates relationship—this updates the initial definition with the ARN of each connected and already deployed function.

```
{ "StartAt": "ListFilesActivity",
 "States": {
 "ListFilesActivity":{"Type":"Task","Resource":"LF_ARN",...,"Next":"TransformDataFanout"},
 "TransformDataFanout": {
   "Type": "Map", "ItemsPath": "$.value", "ResultPath": "$.value",
   "Iterator": {
     "StartAt": "TransformData",
     "States": {"TransformData": {"Type": "Task", "Resource": "TD_ARN", ...,"End":true}}},
     "Next": "AggregateData" },
 "AggregateData": {"Type": "Task", ..., "Next": "CleanUp" },
 "CleanUp": {"Type": "Task", "Resource": "CleanUp_ARN", ..., "End": true }}}
```

Listing 14: AWS Step Functions ASL model generated from the BPMN function orchestration model in Figure 22

Transforming the BPMN function orchestration model for Azure durable functions. Unlike AWS step functions, Azure durable functions requires defining function orchestrations in general-purpose programming languages such as JavaScript or Python by using constructs from libraries that provide orchestration-specific constructs as described in Section 5. Listing 15 shows the listing for the transformed BPMN function orchestration model: For Azure durable functions it is transformed into an `orchestrator_function` in Python. The generated function relies on the `azure.durable_functions` extension to define the control flow. The generated orchestration uses function names as references to the functions in the workflow, for example, *"ListFiles"* references the corresponding function in the function orchestration. Usage of the `yield` keyword ensures the receipt of the results, also including the execution of multiple parallel instances of the TransformData function using the `task_all()` method.

```
import azure.functions as func
import azure.durable_functions as df
def orchestrator_function(context: df.DurableOrchestrationContext):
  result = context.get_input()
  result = yield context.call_activity("ListFiles", result)
  tasks = []
  for item in result["value"]:
    tasks.append(context.call_activity("TransformData", item))
  result = yield context.task_all(tasks)
  result = yield context.call_activity("AggregateData", result)
  result = yield context.call_activity("CleanUp", result)
  return result
main = df.Orchestrator.create(orchestrator_function)
```

Listing 15: Azure Durable Functions orchestrating function generated from theBPMNfunction orchestration model in Figure 22

```
const composer = require('@ibm-functions/composer')
module.exports = composer.sequence(
composer.action('ListFiles', { limits: { timeout: 300000 } }),
composer.map( composer.action('TransformData', { limits: { timeout: 300000 } }) ),
composer.action('AggregateData', { limits: { timeout: 300000 } }),
composer.action('CleanUp', { limits: { timeout: 300000 } }) )
```

Listing 16: IBM Composer orchestrating function generated from the BPMN function orchestration model in Figure 22

Transforming the BPMN function orchestration model for IBM composer. Finally, Listing 16 shows the listing for the transformed BPMN function orchestration model, that is, the `function composition` in JavaScript that can be consumed by the IBM Composer. Similar to Azure durable functions, IBM Composer (as well as the Apache Openwhisk Composer) require defining the function orchestration models using an orchestrating function that uses a dedicated library providing orchestration-specific constructs such as parallel invocation or conditional branching for functions as described in Section 5. The default timeout of five minutes is generated by the BPMN4FO, but can easily be modified if necessary in the generated function orchestration model. After generating the function orchestration model for three target serverless function orchestrators, we can incorporate them in the desired deployment architecture using our TOSCA modeling approach.

8.2 | Modeling the deployment of the air quality data ETL function orchestration

In *Step 3* of our method described in Section 4, we implement the required business logic and model the deployment architecture shown in Figure 2: It incorporates three generated technology-specific function orchestration models with four orchestrated functions implemented for each respective provider, and integrates them with regular serverless components by means of event-driven and direct calls. All orchestrated functions are implemented in Python and rely on `numpy` and `pandas` for processing the open air quality data. The implementations for all three function orchestrators are based on the open source example for AWS step functions available on GitHub[28] and differ mainly in the usage of provider-specific service offerings. For example, the implementations for Azure and IBM had to be reworked due to usage of provider-specific libraries and interaction with provider-specific services, that is, object storage services, FaaS platforms, and message queue services from Microsoft and IBM. We used the concrete, deployable TOSCA types, which we implemented to enable our modeling approach as described in Section 6 for the YAML-based version of the TOSCA standard. The provider-specific deployment logic is implemented using Ansible to enable deploying the resulting deployment architectures with xOpera. However, since TOSCA abstracts away the deployment technology in use, the same models can be deployed using other deployment logic implementations, for example, Terraform, thus, making these TOSCA models portable across deployment automation technologies. All modeling constructs are available on GitHub.[50]

TOSCA-based function orchestration deployment model for AWS. Figure 23 shows the deployment model implementing our example serverless application for AWS. All components are hosted on the AWS-specific *AwsPlatform* node type, which groups the policy configurations and exposes them for reuse as TOSCA attributes after the *configure* operation is completed as described in Section 6.1. All functions are hosted on AWS Lambda and four of them are orchestrated using AWS step functions, that is, *ListFiles*, *TransformData*, *AggregateData*, and *CleanUp* functions. The timer-based invocation of the function orchestration is modeled implicitly (see Section 6.1) by defining a property in the *AwsSFOrchestration* node type and implementing the corresponding AWS event bridge rule configuration in Ansible as a part of the *create* operation for this type. The *AwsSFOrchestrates* relationship type relies on the *pre_configure_source* interface operation for updating the workflow definition shown in Listing 14 with correct function ARNs. After the workflow definition is updated, the *configure*operation on the *AwsSFOrchestration* deploys the updated ASL function orchestration model to AWS step functions. AWS S3 is used as the object storage offering for storing intermediate and final results, and the *AwsS3Triggers* relationship type is used to implement the trigger configuration (using *post_configure_source* interface operation) for invoking AWS Lambda functions based on S3 events, for example, S3 PUT event. Finally, as a message queue offering we use AWS SQS, in which a dedicated queue is created and

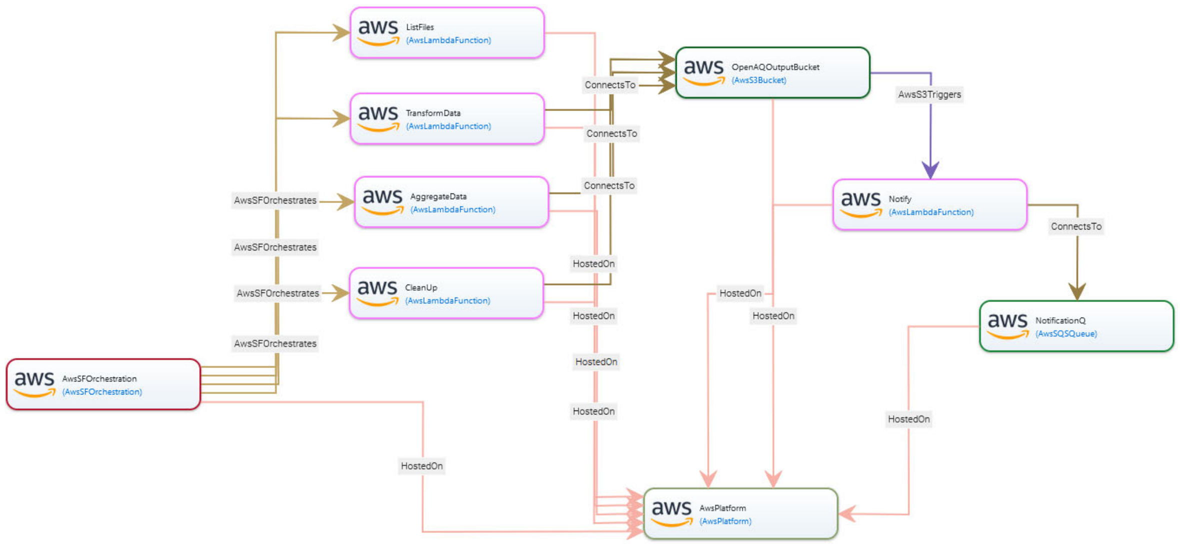

ListFiles (AwsLambdaFunction)
TransformData (AwsLambdaFunction)
AggregateData (AwsLambdaFunction)
CleanUp (AwsLambdaFunction)
AwsSFOrchestration (AwsSFOrchestration)
OpenAQOutputBucket (AwsS3Bucket)
Notify (AwsLambdaFunction)
NotificationQ (AwsSQSQueue)
AwsPlatform (AwsPlatform)
AwsSFOrchestrates
ConnectsTo
AwsS3Triggers
HostedOn

FIGURE 23 The deployment architecture from Section 2 modeled for AWS using TOSCA and Eclipse Winery

to which the *notify* function sends notifications that can be consumed by external applications, for example, deployed on premises.

TOSCA-based function orchestration deployment model for Azure. Figure 24 depicts the deployment model implementing our example serverless application for Azure. All components are hosted on the *AzurePlatform* node type, which is used to configure their deployment (see Section 6.1). The functions are hosted using Azure functions platform, whereas the function orchestration model uses the Azure durable functions extension. Similar to AWS, Azure-specific relationship type *AzureDFOrchestrates*, which inherits from the abstract *orchestrates* relationship, is used to model the orchestration semantics. Azure Blob Storage is used to implement the object storage for storing results, and Azure Storage Queue is used to implement a message queue. While the majority of topological information is similar to the AWS-specific model, the model for Azure has specific constructs not used anywhere else, namely (i) the *AzureFunctionApp* node type which groups functions using the *groups* relationship type, and (ii) the *MainOrchestratorStarter*

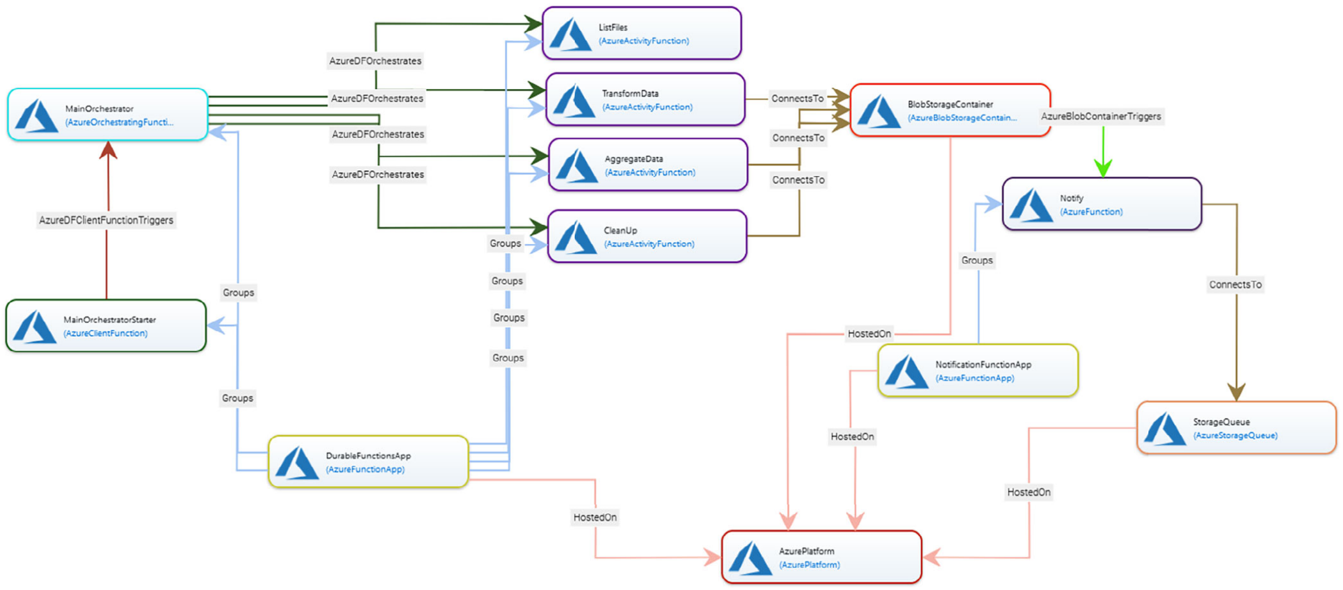

ListFiles (AzureActivityFunction)
TransformData (AzureActivityFunction)
AggregateData (AzureActivityFunction)
CleanUp (AzureActivityFunction)
MainOrchestrator (AzureOrchestratingFuncti...
MainOrchestratorStarter (AzureClientFunction)
DurableFunctionsApp (AzureFunctionApp)
BlobStorageContainer (AzureBlobStorageContain...
Notify (AzureFunction)
NotificationFunctionApp (AzureFunctionApp)
StorageQueue (AzureStorageQueue)
AzurePlatform (AzurePlatform)
AzureDFOrchestrates
AzureDFClientFunctionTriggers
AzureBlobContainerTriggers
ConnectsTo
Groups
HostedOn

FIGURE 24 The deployment architecture from Section 2 modeled for Azure using TOSCA and Eclipse Winery

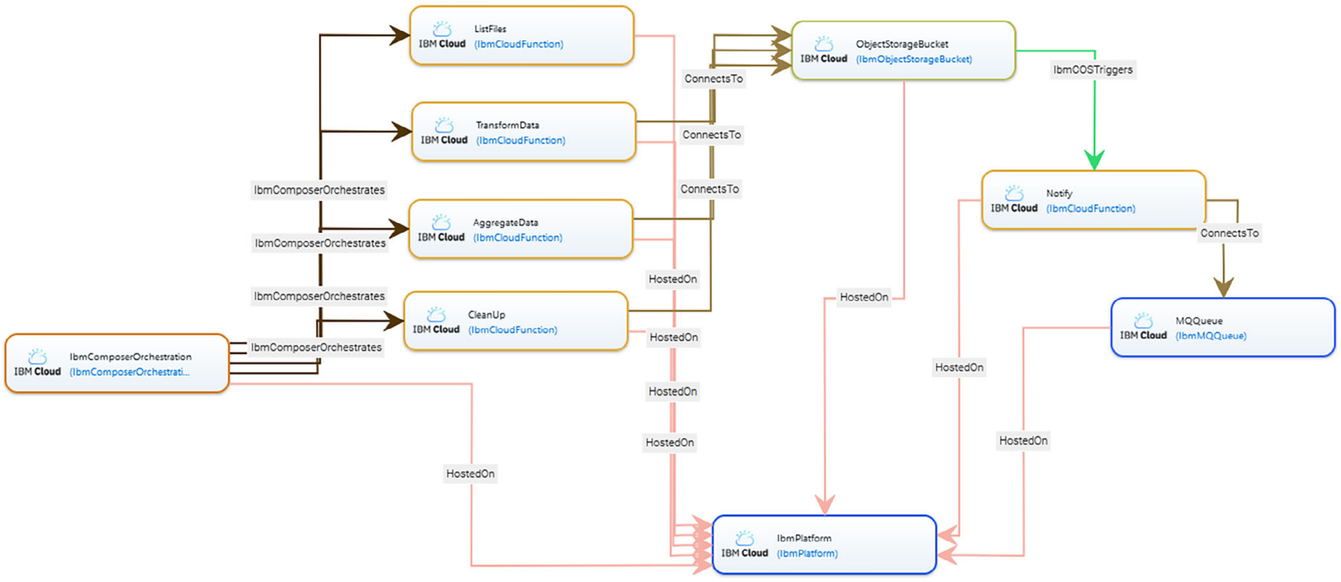

ListFiles
IBM Cloud (IbmCloudFunction)
TransformData
IBM Cloud (IbmCloudFunction)
AggregateData
IBM Cloud (IbmCloudFunction)
CleanUp
IBM Cloud (IbmCloudFunction)
IbmComposerOrchestration
IBM Cloud (IbmComposerOrchestrati...
ObjectStorageBucket
IBM Cloud (IbmObjectStorageBucket)
Notify
IBM Cloud (IbmCloudFunction)
MQQueue
IBM Cloud (IbmMQQueue)
IbmPlatform
IBM Cloud (IbmPlatform)
IbmComposerOrchestrates
ConnectsTo
HostedOn
IbmCOSTriggers

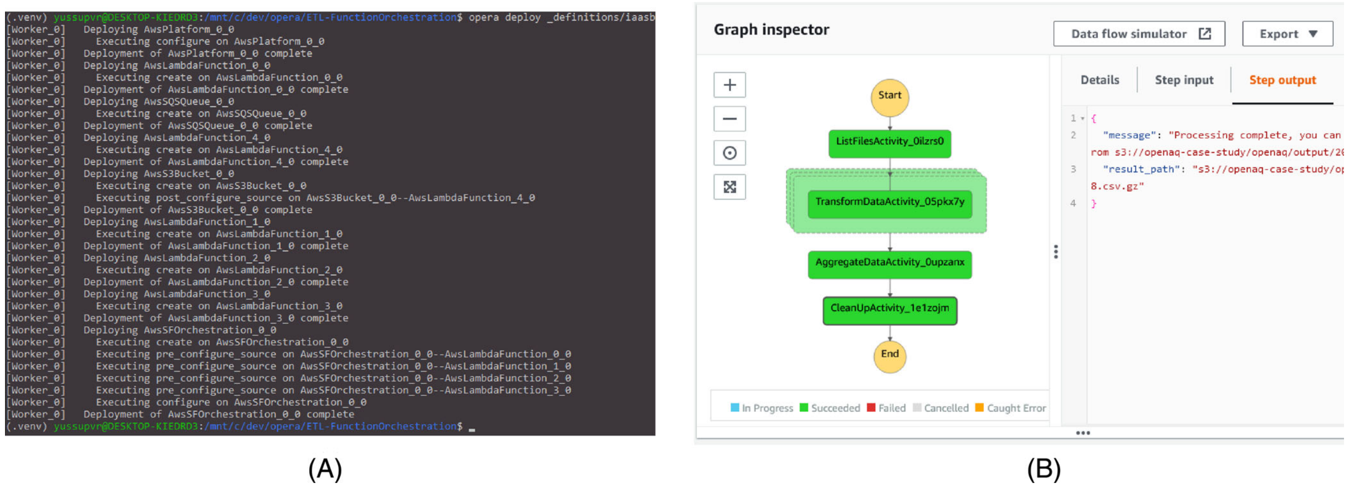
FIGURE 25 The deployment architecture from Section 2 modeled for IBM using TOSCA and Eclipse Winery

function of type *AzureClientFunction* which is used in durable functions to enable triggering the function orchestration. The *AzureFunctionApp* is a convenient way to group functions and their dependencies for deploying the entire sets of functions simultaneously—in our case, the ETL function orchestration is modeled as a dedicated *AzureFunctionApp*, whereas the Notify function is hosted in a separate *AzureFunctionApp*. While bindings can be generated based on the configuration in relationships, in this work we define bindings as separate artifacts in respective functions and the actual binding happens during the *AzureFunctionApp*'s *create* operation together with the deployment of functions. The timer-based triggering is also modeled as a binding definition of the Azure *MainOrchestratorStarter* function.

TOSCA-based function orchestration deployment model for IBM. Finally, Figure 25 demonstrates the deployment model implementing our example serverless application for IBM. As previously, the IBM-specific *IbmPlatform* node type is used to group the deployment configuration for all components in the model (see Section 6.1). It can be observed that the resulting topology is similar to the models presented before, only the IBM-specific services are used. The functions are hosted using IBM cloud functions platform, whereas the function orchestration relies on the IBM Composer, with the relationship type *IbmComposerOrchestrates* representing the orchestration semantics, which also inherits from the abstract *Orchestrates* relationship. The JavaScript-based function orchestration model generated using BPMN4FO is deployed using the *create* operation on the *IbmComposerOrchestration* node type. One important point to highlight is that IBM Composer requires a Redis instance for running workflows that use parallel compositions, meaning that a reference to a running and accessible Redis deployment must be provided as a property of the IBM *IbmComposerOrchestration* node type—we assume that this instance is not the part of the application model and has to be deployed separately, but extending our model to add such information is straightforward using TOSCA. Further, we employ the majority of decisions discussed in Section 6, that is, timer-based triggering is modeled implicitly using the property of the *IbmComposerOrchestration* node type. Finally, as an object storage we use IBM Cloud Object Storage and we employ IBM MQ Cloud for implementing the message queue.

8.3 | Enacting the deployment of the air quality data ETL function orchestration

In *Step 4* of our method described in Section 4, all three deployment models are then exported as respective CSARs using Eclipse Winery to be automatically deployed with xOpera. In *Step 5* of our method described in Section 4, the exported CSARs are deployed using xOpera resulting in running applications for extracting, transforming, and loading the data summary on open air quality and sending the notification to message queues, which can be accessed by external applications, also hosted on premises. Figure 26A shows the deployment of the serverless application model shown



```
(.venv) yussupvr@DESKTOP-KIEDRD3:/mnt/c/dev/opera/ETL-FunctionOrchestration$ opera deploy _definitions/iaasb
[Worker_0]   Deploying AwsPlatform_0_0
[Worker_0]     Executing configure on AwsPlatform_0_0
[Worker_0]   Deployment of AwsPlatform_0_0 complete
[Worker_0]   Deploying AwsLambdaFunction_0_0
[Worker_0]     Executing create on AwsLambdaFunction_0_0
[Worker_0]   Deployment of AwsLambdaFunction_0_0 complete
[Worker_0]   Deploying AwsSQSQueue_0_0
[Worker_0]     Executing create on AwsSQSQueue_0_0
[Worker_0]   Deployment of AwsSQSQueue_0_0 complete
[Worker_0]   Deploying AwsLambdaFunction_4_0
[Worker_0]     Executing create on AwsLambdaFunction_4_0
[Worker_0]   Deployment of AwsLambdaFunction_4_0 complete
[Worker_0]   Deploying AwsS3Bucket_0_0
[Worker_0]     Executing create on AwsS3Bucket_0_0
[Worker_0]     Executing post_configure_source on AwsS3Bucket_0_0--AwsLambdaFunction_4_0
[Worker_0]   Deployment of AwsS3Bucket_0_0 complete
[Worker_0]   Deploying AwsLambdaFunction_1_0
[Worker_0]     Executing create on AwsLambdaFunction_1_0
[Worker_0]   Deployment of AwsLambdaFunction_1_0 complete
[Worker_0]   Deploying AwsLambdaFunction_2_0
[Worker_0]     Executing create on AwsLambdaFunction_2_0
[Worker_0]   Deployment of AwsLambdaFunction_2_0 complete
[Worker_0]   Deploying AwsLambdaFunction_3_0
[Worker_0]     Executing create on AwsLambdaFunction_3_0
[Worker_0]   Deployment of AwsLambdaFunction_3_0 complete
[Worker_0]   Deploying AwsSFOrchestration_0_0
[Worker_0]     Executing create on AwsSFOrchestration_0_0
[Worker_0]     Executing pre_configure_source on AwsSFOrchestration_0_0--AwsLambdaFunction_0_0
[Worker_0]     Executing pre_configure_source on AwsSFOrchestration_0_0--AwsLambdaFunction_1_0
[Worker_0]     Executing pre_configure_source on AwsSFOrchestration_0_0--AwsLambdaFunction_2_0
[Worker_0]     Executing pre_configure_source on AwsSFOrchestration_0_0--AwsLambdaFunction_3_0
[Worker_0]     Executing configure on AwsSFOrchestration_0_0
[Worker_0]   Deployment of AwsSFOrchestration_0_0 complete
(.venv) yussupvr@DESKTOP-KIEDRD3:/mnt/c/dev/opera/ETL-FunctionOrchestration$ _
```

```
{
  "message": "Processing complete, you can
rom s3://openaq-case-study/openaq/output/2
  "result_path": "s3://openaq-case-study/o
8.csv.gz"
}
```

FIGURE 26 Deployment of the model from Figure 23 using xOpera (A) and its execution on AWS step functions (B)

in Figure 23 using xOpera: interface operations for each node in the modeled topology are executed in the correct order, for example, the function orchestration model deployment happens only after all orchestrated functions are deployed and the corresponding ARNs are substituted in the initial function orchestration model using the *pre_configure_source*operations for each modeled relationship. After the function orchestration model is deployed, it gets triggered on a scheduled basis, which in our case every five minutes as it was faster to verify the expected behavior. As discussed previously, the actual execution of the function orchestration is not required to be triggered frequently. Figure 26B shows one instance of successfully executed ETL function orchestration on AWS step functions. For the sake of brevity, we omit the figures depicting the deployment and execution of the workflow models for Azure and IBM clouds.

9 | DISCUSSION AND LIMITATIONS

In this section, we discuss how our contributions are aligned with respect to different facets of portability, and possible ways to address existing limitations of our method.

Portability of function orchestration models. The heterogeneity of modeling languages and orchestrator-specific features makes it nontrivial to obtain portable function orchestration models. Here, developers basically can use our transformation-based approach for establishing a uniform way of modeling function orchestrations that can be executed on different target function orchestrators. Unlike approaches that rely on a dedicated runtime in the target infrastructure to execute the modeled function orchestration such as the serverless workflow specification,[20] one advantage of our approach is that there is no dependency on an external function orchestration runtime, hence, enabling developers to employ existing function orchestrators as-is. Furthermore, compared to approaches without visual notation such as the serverless workflow specification, the visual notation offered by BPMN simplifies explaining function orchestration models produced using our approach to nontechnical personnel, hence bridging the collaboration gap between managerial and DevOps teams.

Our decision to focus on portable function orchestration modeling inherently requires to restrict the support to only the subset of features shared by all function orchestrators. Certainly, such restriction for our BPMN-based modeling and transformation approach is not always necessary, especially for cases when only one target orchestrator format is needed. Essentially, this limitation can be addressed by extending BPMN4FO. Firstly, by adding the "nonportable" mappings for desired function orchestrators, hence enabling the transformation of orchestrator-specific features. Secondly, the BPMN modeling experience can be enhanced by supporting user-provided configurations defining which orchestrator-specific features to keep during the transformation, for example, via corresponding UI elements or external configuration files. Furthermore, for semiautomated processes in which transformed models are intended to be refined and enriched by modelers, the "nonportable" features can be manually added into transformed models. It is also worth emphasizing that our method is not mutually-exclusive with approaches such as the Serverless Workflow Specification: by introducing respective plugins in BPMN4FO we can enable the transformation of produced BPMN function orchestration models into new formats.

Another aspect worth discussing is related to changes in provider-specific orchestration services, for example, adding new or modifying existing features. Clearly, in such cases, the corresponding orchestrator-specific mappings might need to be adjusted to enable generating updated target function orchestration models. However, even for such cases there are still several advantages of using our approach. Firstly, previously created models can be used for generating updated versions of the target formats, hence, facilitating the reuse of existing artifacts based on the changed service requirements. Moreover, the pluggable architecture of BPMN4FO enables updating only the respective technology-specific plugins without requiring general changes in the toolchain.

Portability of function orchestration deployment models. In this work (see Sections 4,6,7, and 8), we partially address the aspect of portability of function orchestration deployment models by defining the abstract-to-concrete TOSCA types hierarchy. Despite relying on the same abstract TOSCA types hierarchy, to actually deploy modeled function orchestrations, one needs to create deployment models for each target cloud infrastructure manually. To reduce the amount of modeling efforts, our approach can be extended by providing automatic refinement from abstract TOSCA-based deployment models (using technology-agnostic TOSCA types such as *function*and *workflow*) to orchestrator-specific TOSCA function orchestration deployment models, for example, by extending Eclipse Winery to support automated refinement of our TOSCA types hierarchy from abstract to concrete models. With this extension, a single function orchestration deployment model needs to be produced and then automatically refined using the underlying toolchain. It is worth highlighting that this limitation appears in the majority of deployment modeling languages, for example, Ansible, Terraform, or serverless framework would require creating infrastructure-specific deployment models referring to particular service offerings from cloud providers. A more portable option is to use Kubernetes-based function orchestrations, since Kubernetes deployment are easier to port across infrastructures. However, the usage of Kubernetes also does not solve the problem of producing portable function orchestration deployment models executable on provider-specific services such as AWS step functions, thus, making the TOSCA models refinement an interesting open research question to be tackled in future work.

Interchangeability of deployment automation technologies. In TOSCA, declarative application deployment models are decoupled from the actual deployment automation technologies, that is, the same TOSCA models can be deployed using different deployment automation technologies by implementing respective implementation artifacts, for example, using Terraform scripts as implementation artifacts. From the modeling perspective, our approach fully addresses this aspect since the types hierarchy can be extended with new artifact types representing other deployment technologies. As a result, the same TOSCA models created using our approach can be automatically deployed with various kinds of deployment logic implementations, also by using other TOSCA-compliant deployment automation technology such as OpenTOSCA. However, to be able to use these TOSCA models with other deployment technologies such as Terraform instead of Ansible, the chosen TOSCA-compliant deployment automation technology must support them. In case of xOpera this would mean that new technology-specific executor plugins need to be introduced, for example, an executor for Terraform.[26] Another alternative is to employ existing approaches[52,53] for transforming TOSCA models into formats consumable by the chosen target technology, for example, TOSCA to Terraform transformations.

Portability of function orchestration business logic implementation. In this article, we do not address the portability aspect related to orchestrator-specific function implementations. However, we made the first step toward portable function orchestrations by introducing concepts for portability of the involved models. As shown in Section 8, the major differences in provider-specific implementations were related to service-specific interactions and code packaging requirements. For example, the actual business logic for processing the open air quality data remained mainly the same for all three provider-specific implementations. Therefore, our next step is to work on identification, decoupling, and description of provider-independent function code that enables its automated wrapping and packaging for target infrastructures. While existing FaaSification approaches[54] could improve the portability aspect for function implementations, often such approaches are restricted and focus on very specific kinds of code extraction, for example, invocation of functions via API gateways. Here, approaches such as Any2API[55] could be used to automate the wrapping of provider-independent business logic in future work.

10 | RELATED WORK

Multiple publications focus on the topic of function composition including the underlying programming models and orchestration engines. Baldini et al.[56] formulate the so-called serverless trilemma and implement an extension for Apache Openwhisk enabling composing functions, which is used as a basis in Openwhisk (and IBM) Composer. Burckhardt et al.[57] introduce a programming model and a distributed execution engine for running serverless function orchestrations that is used by Azure durable functions, that is, enabling modeling of function orchestrations using general-purpose programming languages such as Python. Carreira et al.[58] introduce a framework for defining machine learning workflows, which relies on orchestrating AWS Lambdas and serverless storage offerings for processing workflow tasks. John et al.[59] present a serverless workflow engine for running scientific workflows, which enables executing tasks on serverless offerings such as AWS Lambda and AWS Fargate. Ristov et al.[60] introduce an abstract language for modeling serverless workflows and a system enabling to execute the created models directly on FaaS platforms, hence, circumventing provider-specific function orchestrators. López et al.[61] present a trigger-based orchestrator for serverless functions, designed with long-running tasks in mind. Further, there exist several calculi for expressing the specifics of serverless (FaaS-centric) programming model. For instance, Jangda et al.[62] introduce a calculus capturing the operational semantics of serverless programming model, also including a composition language and the engine which can execute it, implemented on top of Apache Openwhisk. Giallorenzo et al.[63] present a concurrent lambda-calculus which aims to facilitate abstract reasoning about serverless programs for developers and simplify modeling of the serverless implementation layer w.r.t. interactions among processes. Next, several works focus on analyzing the capabilities of existing serverless orchestrators. López et al.[14] analyze and compare the function orchestrators from AWS, Azure, and IBM, also categorizing them based on several criteria such as the type of workflow definition, and Barcelona-Pons et al.[64] investigate the capabilities of the same serverless orchestrators w.r.t. processing of highly-parallel workloads. Finally, it is worth mentioning several projects related to usage of standards in the context of serverless function orchestrations. For example, it was shown that Camunda Cloud (a commercial BPMN engine offered as a service) can be used to orchestrate AWS Lambda functions,[41] highlighting the possibility of using standards such as BPMN in the context of specific cloud infrastructures. The Cloudstate project[65] aims at providing a standard for development of general-purpose applications incorporating stateful services, reactive, and data-intensive components for the Kubernetes ecosystem. The serverless workflow specification[20] aims at providing a standardized workflow modeling language in a form of a custom DSL for defining serverless function orchestrations. Apart from the workflow modeling language, a set of language-specific tools and SDKs are offered, as well as the Kubernetes-native workflow engine supporting the introduced workflow modeling language. This specification focuses on providing a uniform way of describing and executing function orchestrations, which means that a standardized runtime must be used in combination with the models created using the Serverless Workflow Specification. With respect to the topic of function orchestration modeling, *our work differs from all the above approaches* since instead of introducing a new workflow engine, or a new workflow modeling language, we provide an approach for using an existing standard and mapping to target orchestration format, together with the transformation engine supporting three function orchestrators. The introduced approach and system architecture are extensible and can easily support mapping and transformation into other target formats, for example, FaaS-specific engines such as Fission Flow and Fn Flow, or more general-purpose engines such as Apache Airflow.

Multiple modeling approaches for specifying the deployment of serverless applications are available as well. For instance, Bogo et al.[66] and Brogi et al.[67] propose two different solutions to deploy multiservice applications, where the serverless deployment is realized by relying on Docker containers, whilst at the same time not supporting function orchestration mechanisms. Kritikos et al.[68] instead introduce a set of extensions for the CAMEL cloud modeling language, which enable modeling of serverless components including function compositions. However, the extension for modeling compositions is covered briefly, and the references to documentation and implemented example which does not involve a function composition are not available. In our previous work,[24] we described an initial approach for modeling serverless applications with TOSCA without going into specifics of workflow modeling and incorporating them as parts of serverless applications. Several transformation-centric approaches exist which aim at uniform representations of application deployment and configuration models which can be transformed into the formats of chosen target environments. Samea et al.[69] present a model-driven configuration approach for cloud environments using a UML profile. A transformation engine, which enables generating configurations for target environments is mentioned in the context of future work. Wurster et al.[44] introduce a so-called essential deployment metamodel and implement a transformation system which can translate a subset of TOSCA into multiple target deployment formats such as Terraform or Ansible. Dehury et al.[70] introduce a TOSCA-based approach for modeling data pipelines which can also incorporate serverless components such as FaaS functions. Tsagkaropoulos et al.[71] introduce TOSCA extensions for edge and fog deployment modeling which also include constructs for modeling FaaS-hosted functions on provider-managed platforms and on premises platforms, however, without covering the topic of modeling function orchestrations as well as their deployments. Moreover, existing graph-based models can be used for analyzing the application structure and identification of hot spots[72] or comparing deployment models in a uniform way.[73] With respect to the topic of function orchestration deployment modeling, *our work differs from all aforementioned approaches* since we not only enable including serverless components in deployment models, but since we enable also modeling serverless function orchestration deployment models, generating orchestrator-specific, deployable function orchestration models, and including such function orchestration models in deployment models.

11 | CONCLUSIONS

In this article, we analyzed how existing standards for modeling workflows (BPMN) and application deployments (TOSCA) can be used in the context of serverless function orchestrations. As a result, we introduced a method for standards-based modeling and deploying serverless function orchestrations that relies on BPMN and TOSCA modeling approaches. The main contributions of our work are as follows. We introduced (i) a uniform modeling and transformation approach for representing serverless function orchestrations in BPMN and transforming them into target orchestration formats such as ASL for AWS step functions. We (ii) extended our previous TOSCA-based deployment modeling approach with support for modeling the deployment of serverless function orchestrations. Furthermore, we implemented (iii) an open source modeling toolchain for using both approaches together, enabling to automatically deploy resulting deployment models using the open source TOSCA deployment automation technology called xOpera. Finally, we (iv) implemented a case study application and successfully deployed it using our approaches and the underlying toolchain to three public cloud providers—AWS, Azure, and IBM.

We believe that the standards-based modeling approaches and the toolchain presented in this article can be of help to both researchers and practitioners wishing to develop and automatically deploy serverless function orchestration in technology-agnostic fashion. Indeed, as we shown in Section 8, the uniform BPMN-based function orchestration modeling can first be used to specify portable function orchestration models, which can then be automatically translated into orchestrator-specific, runnable model formats. Our TOSCA-based deployment modeling approach (and the employed Eclipse Winery) can then be used to specify the deployment of a serverless function orchestrations for coordinating multiple functions independently, or as parts of larger serverless applications as was shown in Section 8. The CSARs obtained at the end of the process can actually be processed with a TOSCA-compliant deployment technology such as xOpera to concretely deploy serverless applications and execute the function orchestrations therein. Another example where standards can be helpful is the topic of decision support: by defining standards-based models, developers can easier understand whether the given model is supported by the target technology. Both BPMN4FO and the Eclipse Winery described in Section 7 can support researchers and practitioners in deciding which technology to target, for example, if specifying some function orchestration configuration that is not supported by an orchestrator, BPMN4FO will not permit generating the corresponding orchestrator-specific model, hence informing developers that they cannot deploy their application with such orchestrator. Finally, the audience employing these particular standards can benefit from the presented experience by building on top of these approaches for modeling serverless function orchestrations and their deployments. Firstly, we illustrated how BPMN can be used as-is to model serverless function orchestrations, based on an analysis of existing serverless function orchestrators from three major providers (AWS, Azure, and IBM) with respect to the properties of the underlying function orchestration modeling languages. Secondly, we also discussed how to extend TOSCA with additional types to support the deployment of serverless applications incorporating serverless function orchestrations, whilst always motivating the rationale of our modeling choices. Both can then be of help to researchers and practitioners wishing to perform similar modeling tasks, either using the same or different standards for specifying function orchestrations and application deployments.

In future work, we plan to extend our method and the underlying toolchain to support automatic refinement of abtsract TOSCA-based function orchestration deployment models into concrete, orchestrator-specific orchestration models. As discussed in Section 9, this would enable reducing the amount of required deployment modeling efforts. Further, we aim to extend our BPMN4FO prototype to support orchestrator-specific mappings, hence, enabling modelers to use features that are unique to specific orchestrators for nonportable transformation scenarios and more proprietary function orchestration formats, also including the Serverless Workflows Specification format. Together with corresponding

TOSCA models, this extension will make our approach more flexible w.r.t. supported target infrastructures, for example, enabling the support for Kubernetes-based modeling or the deployment of function orchestrations using Apache Openwhisk and Apache Openwhisk Composer. Finally, to enable using other deployment automation technologies, we plan to combine our method with existing approach for transforming TOSCA models into deployable, technology-specific models, for example, in Ansible or Terraform.[52,53] With this extension, we will also enable deploying the produced models without using TOSCA deployment automation technologies.

ACKNOWLEDGMENT

Open access funding enabled and organized by Projekt DEAL.

AUTHOR CONTRIBUTIONS

Vladimir Yussupov: Conceptualization, methodology, investigation, data curation, software, writing–original draft, writing–review & editing. **Jacopo Soldani**: Conceptualization, methodology, investigation, data curation, writing–original draft, writing–review & editing. **Uwe Breitenbücher**: Conceptualization, methodology, investigation, writing–review & editing. **Frank Leymann**: Conceptualization, methodology, investigation, writing–review & editing, funding acquisition.

ORCID

Vladimir Yussupov https://orcid.org/0000-0002-6498-637X
Jacopo Soldani https://orcid.org/0000-0002-2435-3543
Uwe Breitenbücher https://orcid.org/0000-0002-8816-5541
Frank Leymann https://orcid.org/0000-0002-9123-259X

REFERENCES

1. Kounev S, Abad C, Foster IT, et al. Toward a definition for serverless computing. *Proceedings of the Serverless Computing (Dagstuhl Seminar 21201)*. Vol 11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik; 2021:56-59.
2. Baldini I, Castro P, Chang K, et al. Serverless computing: current trends and open problems. *Research Advances in Cloud Computing*. 1st ed. Springer; 2017:1-20.
3. Yussupov V, Soldani J, Breitenbücher U, Brogi A, Leymann F. FaaSten your decisions: a classification framework and technology review of function-as-a-service platforms. *J Syst Softw*. 2021;175:1-24.
4. Bouzerzour NEH, Ghazouani S, Slimani Y. A survey on the service interoperability in cloud computing: client-centric and provider-centric perspectives. *Softw Pract Exper*. 2020;50(7):1025-1060. doi:10.1002/spe.2794
5. Yussupov V, Soldani J, Breitenbücher U, Brogi A, Leymann F. From serverful to serverless: a spectrum of patterns for hosting application components. In: Helfert M, Ferguson D, Pahl C, eds. *Proceedings of the 11th International Conference on Cloud Computing and Services Science . 2021 CLOSER. Virtual Conference*. SciTePress; 2021:268-279.
6. Hellerstein JM, Faleiro J, Gonzalez JE, et al. Serverless computing: one step forward, two steps back; 2018. arXiv preprint arXiv:1812.03651.
7. Bruns R, Dunkel J. Towards pattern-based architectures for event processing systems. *Softw Pract Exper*. 2014;44(11):1395-1416. doi:10.1002/spe.2204
8. Leymann F, Roller D. *Production Workflow: Concepts and Techniques*. 1st ed. Prentice Hall PTR; 1999.
9. Van der Aalst WM, Ter Hofstede AH, Kiepuszewski B, Barros AP. Workflow patterns. *Distrib Parallel Databases*. 2003;14(1):5-51.
10. Apache. OpenWhisk composer; 2021. https://github.com/apache/openwhisk-composer
11. Fn Project. Contributors. Fn Flow; 2021. https://github.com/fnproject/flow
12. Apache. OpenWhisk; 2021. https://openwhisk.apache.org/.
13. Fn Project. Contributors. Fn; 2021. https://github.com/fnproject/fn
14. García López P, Sánchez-Artigas M, París G, Barcelona Pons D, Ruiz Ollobarren Á, Arroyo Pinto D. Comparison of FaaS orchestration systems. In: Sill A, Spillner J. , eds. *Proceedings of the 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion. 2018 edition of UCC Companion*; 2018:148-153; IEEE.
15. Amazon Web Services, Inc. AWS step functions; 2020. https://docs.aws.amazon.com/step-functions
16. Microsoft Azure, Inc. Azure durable functions; 2021. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-%functions-overview
17. Amazon Web Services, Inc. AWS CloudFormation; 2021. https://aws.amazon.com/cloudformation/
18. Red Hat. Ansible; 2021. https://www.ansible.com
19. Serverless, Inc. Serverless framework; 2021. https://www.serverless.com
20. Serverless workflow specification authors serverless workflow specification; 2021. https://serverlessworkflow.io
21. OMG. Business process model and notation (BPMN) version 2.0; 2011. https://www.omg.org/spec/BPMN/2.0/PDF

22. OASIS. TOSCA simple profile in YAML version 1.3; 2020. https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/TOSCA-Simple-Profile-YAML-v1.3.html
23. Amazon Web Services, Inc. Amazon states language; 2021. https://states-language.net/spec
24. Wurster M, Breitenbücher U, Képes K, Leymann F, Yussupov V. Modeling and automated deployment of serverless applications using TOSCA. In: Kobusinska A, Fan J, eds. *Proceedings of the IEEE 11th International Conference on Service-Oriented Computing and Applications*. Vol 2018. IEEE; 2018:73-80.
25. Binz T, Breitenbücher U, Kopp O, Leymann F. TOSCA: portable automated deployment and management of cloud applications. *Advanced Web Services*. Springer; 2014:527-549.
26. Luzar A, Stanovnik S, Cankar M. Examination and comparison of TOSCA orchestration tools. In: Muccini H, Avgeriou P, Buhnova B, et al., eds. *Software Architecture. 1269 of CCIS. Virtual Conference*. Springer; 2020:247-259.
27. Kopp O, Binz T, Breitenbücher U, Leymann F. Winery – a modeling tool for TOSCA-based cloud applications. In: Basu S, Pautasso C, Zhang L, Fu X, eds. *Proceedings of the 11th International Conference on Service-Oriented Computing. 2013 edition of ICSOC*. Springer; 2013:700-704.
28. Contributors to AWS samples repository. serverless reference architecture: extract transfer load; 2020. https://github.com/aws-samples/aws-lambda-etl-ref-architecture
29. Amazon Web Services, Inc. AWS Lambda; 2021. https://aws.amazon.com/lambda
30. Microsoft Azure, Inc. Azure functions; 2021. https://docs.microsoft.com/en-us/azure/azure-functions
31. IBM. Compositions for IBM composer; 2021. https://github.com/ibm-functions/composer#defining-a-composition
32. IBM. Cloud functions; 2021. https://www.ibm.com/cloud/functions
33. Amazon Web Services, Inc. Amazon event bridge; 2021. https://docs.aws.amazon.com/eventbridge/index.html
34. Amazon Web Services, Inc. AWS serverless application model; 2021. https://aws.amazon.com/serverless/sam/
35. HashiCorp. Terraform; 2021. https://www.terraform.io
36. Chinosi M, Trombetta A. BPMN: an introduction to the standard. *Comput Stand Interfaces*. 2012;34(1):124-134.
37. OASIS. Web services business process execution language (WS-BPEL) version 2.0; 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
38. Russell N, Ter Hofstede AH, Van Der Aalst WM, Mulyar N. Workflow control-flow patterns: a revised view. BPM Center Report BPM-06-22, BPMcenter. org; 2006:06-22.
39. Wieland M, Kopp O, Nicklas D, Leymann F. Towards context-aware workflows. Proceedings of the Workshops and Doctoral Consortium CAiSE07; 2007; Citeseer.
40. Moody D. The ɛphysicsɛ of notations: toward a scientific basis for constructing visual notations in software engineering. *IEEE Trans Softw Eng*. 2009;35(6):756-779.
41. Ruecker B. How to orchestrate AWS lambda using camunda cloud: powerful serverless function orchestration using BPMN and cloud-native workflow technology. https://camunda.com/blog/2020/05/how-to-orchestrate-aws-lambda-using-camunda-cloud
42. Combi C, Oliboni B, Zerbato F. Modeling and handling duration constraints in BPMN 2.0. In: Shin D, Lencastre M, eds. *Proceedings of the Symposium on Applied Computing*. Vol 2017. ACM; 2017:727-734.
43. Camunda. BPMN workflow engine; 2021. https://camunda.com/products/camunda-platform/bpmn-engine
44. Wurster M, Breitenbücher U, Falkenthal M, et al. The essential deployment metamodel: a systematic review of deployment automation technologies. *SICS Softw Intens Cyber-Phys Syst*. 2019;35:63-75.
45. Endres C, Breitenbücher U, Falkenthal M, Kopp O, Leymann F, Wettinger J. Declarative vs. imperative: two modeling patterns for the automated deployment of applications. In: Manaert H, Iwahori Y, Mirnig A, Ortis A, Perez C, Daykin J, eds. *Proceedings of the 9th International Conference on Pervasive Patterns and Applications. 2017 PATTERNS*. Xpert Publishing Services (XPS); 2017:22-27.
46. Breitenbücher U, Endres C, Képes K, et al. The OpenTOSCA ecosystem - concepts & tools. European Space project on Smart Systems, Big Data, Future Internet - Towards Serving the Grand Societal Challenges - Volume 1: EPS Rome 2016; 2016:112-130. doi: 10.5220/0007903201120130
47. Breitenbücher U, Binz T, Képes K, Kopp O, Leymann F, Wettinger J. Combining declarative and imperative cloud application provisioning based on TOSCA. In: Bacon J, Li B, eds. *Proceedings of the IEEE International Conference on Cloud Engineering. 2014 IC2E*. IEEE; 2014:87-96.
48. Taibi D, El Ioini N, Pahl C, Niederkofler JRS. Patterns for serverless functions (Function-as-a-Service): a multivocal literature review. In: Ferguson D, Helfert M, Pahl C, eds. *Proceedings of the 10th International Conference on Cloud Computing and Services Science*. Vol 2020. SciTePress; 2020:181-192.
49. Mlitz K. Distribution of cloud computing market revenues worldwide from 2015 to June 2019, by vendor; 2021. https://www.statista.com/statistics/540511/worldwide-cloud-computing-revenue-share-by-vendor/
50. Yussupov V, Soldani J, Breitenbücher U, Leymann F. TOSCA definitions for modeling serverless function orchestrations; 2021. https://github.com/iaas-splab/function-orchestration-modeling
51. BPMN.io. bpmn-js; 2021. https://bpmn.io/toolkit/bpmn-js
52. Wurster M, Breitenbücher U, Falkenthal M, et al. The essential deployment metamodel: a systematic review of deployment automation technologies. *SICS Softw Intens Cyber-Phys Syst*. 2019;35:63-75.
53. Wurster M, Breitenbücher U, Harzenetter L, Leymann F, Soldani J, Yussupov V. TOSCA light: bridging the gap between the TOSCA specification and production-ready deployment technologies. In: Ferguson D, Helfert M, Pahl C, eds. *Proceedings of the 10th International Conference on Cloud Computing and Services Science*. 2020 . SciTePress; 2020: 216-226

54. Spillner J. Transformation of python applications into function-as-a-service deployments; 2017. arXiv preprint arXiv:1705.08169.
55. Wettinger J, Breitenbücher U, Leymann F. Any2API – automated APIfication. In: Helfert M, Ferguson D, Munoz V, eds. *Proceedings of the 5th International Conference on Cloud Computing and Services Science*. Vol 2015. SciTePress; 2015:475-486.
56. Baldini I, Cheng P, Fink SJ, et al. The serverless trilemma: function composition for serverless computing. In: Torlak E, van der Storm T, Biddle R, eds. *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. 2017 Onward! Vancouver, Canada*. ACM; 2017:89-103.
57. Burckhardt S, Gillum C, Justo D, Kallas K, McMahon C, Meiklejohn CS. Serverless workflows with durable functions and netherite; 2021. arXiv preprint arXiv:2103.00033.
58. Carreira J, Fonseca P, Tumanov A, Zhang A, Katz R. Cirrus: a serverless framework for end-to-end ML workflows. In: Geambasu R, Mozafari B, eds. *Proceedings of the ACM Symposium on Cloud Computing*. Vol 2019. ACM; 2019:13-24.
59. John A, Ausmees K, Muenzen K, Kuhn C, Tan A. SWEEP: accelerating scientific research through scalable serverless workflows. In: Johnson K, Spillner J, Kluscek D, Anjum A, eds. *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion*. Vol 2019. ACM; 2019:43-50.
60. Ristov S, Pedratscher S, Fahringer T. AFCL: an abstract function choreography language for serverless workflow specification. *Futur Gener Comput Syst*. 2021;114:368-382.
61. López PG, Arjona A, Sampé J, Slominski A, Villard L. Triggerflow: trigger-based orchestration of serverless workflows. In: Troubitsyna E, ed. *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*. Vol 2020. ACM; 2020:3-14.
62. Jangda A, Pinckney D, Brun Y, Guha A. Formal foundations of serverless computing. In: Visser E, ed. *Proceedings of the ACM on Programming Languages*. Athens, Greece. ACM; 2019.
63. Giallorenzo S, Lanese I, Montesi F, Sangiorgi D, Zingaro SP. The servers of serverless computing: a formal revisitation of functions as a service. In: de Boer F, Mauro J, eds. *Recent Developments in the Design and Implementation of Programming Languages. 86 of OASIcs. DROPS*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik; 2020:5:1-5:21.
64. Barcelona-Pons D, García-López P, Ruiz Á, Gómez-Gómez A, París G, Sánchez-Artigas M. Faas orchestration of parallel workloads. In: Castro P, Ishakian V, Muthusamy V, Slominski A, eds. *Proceedings of the 5th International Workshop on Serverless Computing*. Vol 2019. ACM; 2019:25-30.
65. Cloudstate. Distributed state management for serverless; 2021. https://cloudstate.io
66. Bogo M, Soldani J, Neri D, Brogi A. Component-aware orchestration of cloud-based enterprise applications, from TOSCA to Docker and Kubernetes. *Softw Pract Exper*. 2020;50(9):1793-1821. doi:10.1002/spe.2848
67. Brogi A, Rinaldi L, Soldani J. TosKer: a synergy between TOSCA and Docker for orchestrating multicomponent applications. *Softw Pract Exper*. 2018;48(11):2061-2079. doi:10.1002/spe.2625
68. Kritikos K, Skrzypek P, Moga A, Matei O. Towards the modelling of hybrid cloud applications. In: Fox G, Leymann F. , eds. 2019 *Proceedings of the IEEE 12th International Conference on Cloud Computing. 2019 CLOUD. Milan, Italy*. IEEE; 291-295.
69. Samea F, Azam F, Anwar MW, Khan M, Rashid M. A UML profile for multi-cloud service configuration (UMLPMSC) in event-driven serverless applications. In: Benedicenti L, Fujita M, Gorlatch S, Malakhov E, eds. *Proceedings of the 2019 8th International Conference on Software and Computer Applications. 2019 ICSCA. Penang, Malaysia*. ACM; 2019:431-435.
70. Dehury C, Jakovits P, Srirama SN, Tountopoulos V, Giotis G. Data pipeline architecture for serverless platform. In: Muccini H, Avgeriou P, Buhnova B, et al., eds. *Software Architecture CCIS. Virtual Conference*. Vol 1269. Springer; 2020:241-246.
71. Tsagkaropoulos A, Verginadis Y, Compastié M, Apostolou D, Mentzas G. Extending TOSCA for edge and fog deployment support. *Electronics*. 2021;10(6):737.
72. Winzinger S, Wirtz G. Model-based analysis of serverless applications. In: Chechik M, Strüber D, Varró D. , eds. 2019 *IEEE/ACM 11th International Workshop on Modelling in Software Engineering*. IEEE; 2019: 82-88.
73. Yussupov V, Breitenbücher U, Kaplan A, Leymann F. SEAPORT: assessing the portability of serverless applications. In: Ferguson D, Helfert M, Pahl C, eds. *Proceedings of the 10th International Conference on Cloud Computing and Services Science*. Vol 2020. SciTePress; 2020:456-467.

How to cite this article: Yussupov V, Soldani J, Breitenbücher U, Leymann F. Standards-based modeling and deployment of serverless function orchestrations using BPMN and TOSCA. *Softw Pract Exper*. 2022;1-42. doi: 10.1002/spe.3073