SEAPORT: Assessing the Portability of Serverless Applications

Vladimir Yussupov[1], Uwe Breitenbücher[1], Ayhan Kaplan[2] and Frank Leymann[1]
[1]*Institute of Architecture of Application Systems, University of Stuttgart, Germany*
[2]*iC Consult GmbH, Stuttgart, Germany*

Keywords: Serverless, Function-as-a-Service, FaaS, Portability, Decision Support System.

Abstract: The term serverless is often used to describe cloud applications that comprise components managed by third parties. Like any other cloud application, serverless applications are often tightly-coupled with providers, their features, models, and APIs. As a result, when their portability to another provider has to be assessed, application owners must deal with identification of heterogeneous lock-in issues and provider-specific technical details. Unfortunately, this process is tedious, error-prone, and requires significant technical expertise in the domains of serverless and cloud computing. In this work, we introduce SEAPORT, a method for automatically assessing the portability of serverless applications with respect to a chosen target provider or platform. The method introduces (i) a canonical serverless application model, and (ii) the concepts for portability assessment involving classification and components similarity calculation together with the static code analysis. The method aims to be compatible with existing migration concepts to allow using it as a complementary part for serverless use cases. We present an architecture of a decision support system supporting automated assessment of the given application model with respect to the target provider. To validate the technical feasibility of the method, we implement the system prototypically.

1 INTRODUCTION

The term *serverless* often refers to applications comprising third party-managed components, which results in reduced maintenance costs and faster time to market (Baldini et al., 2017). *Function-as-a-Service (FaaS)* is a cloud service model allowing to host business logic as fine-grained, executable *functions* managed by providers. Functions can be used within applications, e.g., for data formats conversion, or as standalone components, e.g., single HTTP endpoints. The invocation is mainly event-driven, making functions a *reactive mechanism* for processing various events, e.g., database insert events triggering generation of thumbnails (Cloud Native Computing Foundation (CNCF), 2018). In addition to provider-managed scaling with an unbounded number of instances, functions are also scaled to zero when they are idle. By scaling to zero, FaaS eliminates the expenditures for idle components, which is not the case for, e.g., PaaS deployments. However, despite the benefits, serverless architectures are even more prone to various kinds of lock-in issues due to the relinquished control over the infrastructure and thus stronger dependence on provider's configurations and features, e.g., not supported memory limit or a specific event source integration. As a result, porting such applications requires solving cumbersome serverless-specific issues (Yussupov et al., 2019a). Particularly manual portability assessment is inefficient and error-prone, e.g., search of alternatives for unsupported components or analyze required code modifications.

In this work, we introduce the *SErverless Applications PORtability assessmenT (SEAPORT)* method helping to assess the portability of serverless applications. SEAPORT enables automated assessment of a provided application using a canonical serverless application format, and involves checking the component architecture's portability to the chosen provider and analyzing the included source code artifacts to provide a comprehensive overview of portability issues. We discuss the method's steps, introduce a canonical serverless application model based on the idea of pipes and filters pattern (Hohpe and Woolf, 2004), and present a decision support system's architecture enabling the method. Moreover, to make the method compatible with existing migration concepts, as an integration point we discuss the boilerplate models and code generation, and validate the presented concepts via a prototypical implementation.

Yussupov, V., Breitenbücher, U., Kaplan, A. and Leymann, F.
SEAPORT: Assessing the Portability of Serverless Applications.
DOI: 10.5220/0009574104560467
In *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*, pages 456-467
ISBN: 978-989-758-424-4

2 BACKGROUND AND PROBLEM STATEMENT

In this section, we provide the relevant background, describe the problem and formulate the research question we intend to answer.

2.1 Serverless Computing and FaaS

In general, as serverless applications comprise components managed by third parties, developers are no longer required to deal with managing the underlying infrastructure (Cloud Native Computing Foundation (CNCF), 2018; Baldini et al., 2017). Tasks such as resource provisioning and scaling become provider's burden, which allows focusing more on business logic implementation and reducing the overall time to market. The Function-as-a-Service cloud service model is one of the essential parts in serverless application development as it allows hosting application's business logic in a form of event-driven functions that are often short-lived, stateless and are scaled automatically by cloud providers. One of the main advantages of FaaS is that functions can be scaled to zero after a certain inactivity period, which eliminates the need to pay for idle instances (Baldini et al., 2017; Cloud Native Computing Foundation (CNCF), 2018; Castro et al., 2019). However, FaaS also has some known limitations, e.g., certain FaaS platforms impose quotas on the function execution time, making it more difficult to implement more complex use cases (Hellerstein et al., 2018). Since functions are scaled to zero, typically, it is often recommended making them stateless while storing the application state in, e.g., database services, which is also an example of a restriction imposed by the FaaS cloud service model.

2.2 Lock-in and Portability

The problem of becoming dependent on properties and requirements of a chosen product, i.e., *lock-in*, is well-known (Greenstein, 1997) and can be encountered in various situations, e.g., choosing certain hardware, or software development framework. Cloud computing is also an example of a field with multiple lock-in issues (Satzger et al., 2013; Beslic et al., 2013; Opara-Martins et al., 2014), which occur on various levels including provider-specific runtimes and packaging formats. For instance, applications can be locked into a provider-specific REST API or custom message format requirements, allowed memory and storage limits, or custom deployment automation technologies like AWS Cloud Formation[1].

[1]https://aws.amazon.com/cloudformation

There are various reasons, why changing a provider becomes necessary, e.g., change of technology, costs optimization, decrease in the quality of service, bankruptcy of a provider, legal issues or simply termination of the contract (Petcu, 2011). While customers choose providers willingly and lock-in per se is not a daily problem, the process of migrating existing serverless applications can become a serious lock-in resolution issue because (i) cloud applications are often built without portability in mind (Opara-Martins et al., 2014), (ii) lock-in issues are very heterogeneous, e.g., vendor, product, version, or architecture lock-ins (Hohpe, 2019), and (iii) major parts of serverless applications are prone to lock-in due to reduced control over the infrastructure. For example, the business logic hosted as FaaS functions gets coupled with the specifics of a chosen FaaS platform, e.g., different event formats and triggers configuration for AWS Lambda and Microsoft Azure Functions. Additionally, similar issues occur in other component types in serverless applications, e.g., provider-managed databases which have varying APIs and underlying models. As a result, application owners must face classic portability and interoperability issues in the cloud (Bozman and Chen, 2010; Petcu, 2011; Stravoskoufos et al., 2014) combined together with serverless-specific lock-in issues (Yussupov et al., 2019a) which require a good understanding of, e.g., component mappings, provider-specific development guidelines and requirements, and packaging formats.

2.3 Problem Statement

Unfortunately, assessing the portability for serverless applications across providers manually is an inefficient and error-prone process since (i) the available component mappings are often not known in advance which requires investing time into analyzing provider's offerings, and (ii) the hosted function code can contain incompatible fragments which might be easily overlooked, e.g., usage of a non-portable library. The evaluation phase of a serverless migration process for a given application can, therefore, benefit from an automated portability assessment which can help to minimize the organizational efforts and estimate the needed actions before proceeding with actual migration. However, such evaluation often requires an in-depth knowledge of multiple topics including (i) serverless- and FaaS-related technical details such as trigger specifications, event formats, and configuration requirements, (ii) deployment modeling options with respect to the chosen provider, and (iii) the knowledge of possible provider-specific code fragments that require attention. Moreover, to facili-

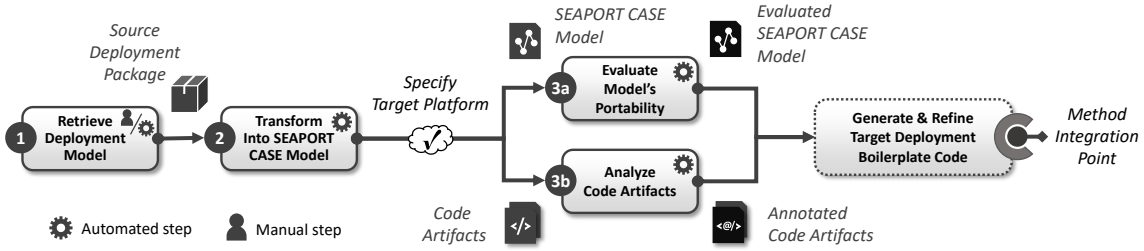

Figure 1: An overview of the SEAPORT method's steps.

tate the method usage as a part of existing migration concepts there should be a feasible *method integration point* allowing to reuse the obtained portability knowledge as a part of the larger migration process, e.g., recommended code modifications and highlighting portability pitfalls. In this work, we formulate and answer the following research question: *"How to automatically assess the portability of a given serverless application's component architecture and the available source code with respect to the selected target provider and allow reusing the obtained knowledge as a part of the overall migration process?"*

3 THE SEAPORT METHOD

In this section, we introduce the SEAPORT method that allows automatically assessing the portability of a given serverless application by (i) checking the compatibility of its component architecture with the selected target provider, and (ii) analyzing the source code of components that host business logic, i.e., FaaS-hosted functions. To answer the research question formulated in Section 2.3, we start with the description of SEAPORT method which comprises three steps shown in Figure 1, together with the method integration point relying on boilerplate code generation to allow using it as a complementary part for migration of serverless use cases.

3.1 Step 1: Retrieve Deployment Model

Obtain the application's deployment model describing application components and their configuration, together with the corresponding code artifacts.

In the first step, the deployment model of a given serverless application is retrieved together with code artifacts, e.g., FaaS components' code. Essentially, a serverless application can be deployed using (i) manual deployment, e.g., deploying components separately without having any deployment model, or (ii) model-based deployment, e.g., via provided CLI or GUI, or using provider-specific and third-party deployment automation technologies like AWS Cloud Formation, Terraform[2], or Serverless Framework[3]. The model-based deployment relies either on imperative or declarative models (Endres et al., 2017), where the former defines a set of required actions and the latter describes the desired state for all related application components (Wurster et al., 2019b). Typically, deployment automation technologies like Terraform or Serverless Framework rely on declarative deployment models, which provide all relevant information for application's portability assessment.

Firstly, the application's provider-specific deployment model contains necessary structural details, e.g., which types of components are used, their interconnections, event bindings, and other configuration details. In cases when applications are deployed using a provider-specific interface, e.g., AWS GUI, the deployment model has to be crawled, e.g., by querying the deployment model-related data using AWS CLI. Another possible option is to extract the application topology of a running instance (Binz et al., 2013).

The reason why source code artifacts must be available is that the source code might contain potential portability pitfalls, e.g., lock-in into unsupported library versions or incompatible service calls. Here, deployment models are helpful for components' code discovery as well, since the source code artifacts are either contained in the deployment package or referenced in the deployment model.

In the next steps, we assume that

(1) a declarative deployment modeling is used and the deployment model is available,

(2) source code is accessible either because it is contained in deployment package or referenced in the model, and

(3) a deployment automation tooling is used to execute the deployment.

[2]https://www.terraform.io
[3]https://serverless.com

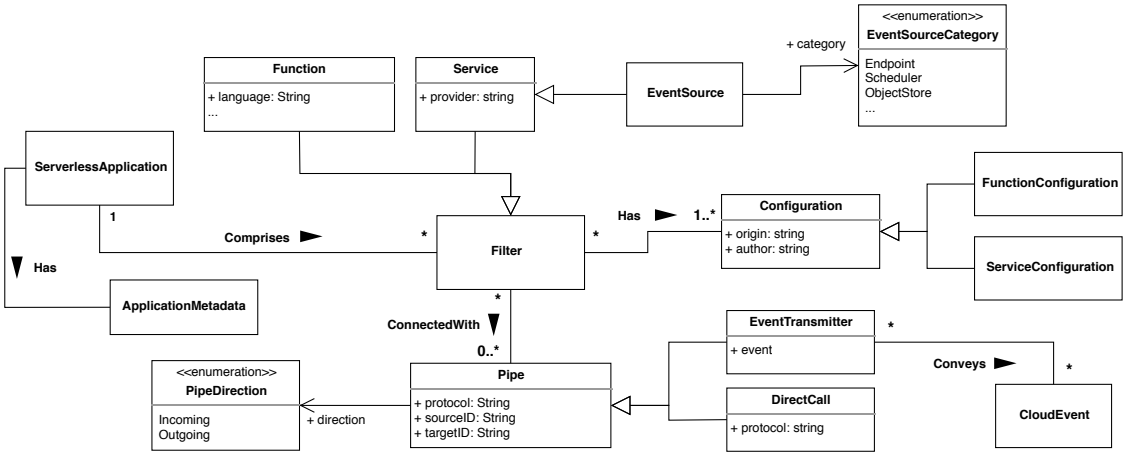

Figure 2: A UML class diagram of the SEAPORT CASE model.

3.2 Step 2: Transform Into the SEAPORT CASE Model

Transform the deployment model into a canonical serverless application format for checking target provider's compatibility.

In the second step, we transform the obtained deployment model into a provider-agnostic format. The main reason for transforming provider-specific deployment models into a canonical format is to simplify evaluating the portability of application's components and enable the results reuse. For example, if several target platforms are to be evaluated, the canonical format eliminates unnecessary point-to-point transformations. To abstract away provider details, as a part of the SEAPORT method we introduce the CAnonical SErverless (CASE) model. Although there exist modeling approaches for serverless applications, e.g., focusing on the detailed deployment and configuration modeling (Wurster et al., 2018; Kritikos et al., 2019; Samea et al., 2019) or abstract dependency graph representation (Winzinger and Wirtz, 2019), the SEAPORT CASE model intends to be a compromise between too detailed and too high-level views while at the same time keeping the topological information together with configuration details. This would allow using the model as a generic serverless application representation that can be mapped to more technology-specific use cases or abstracted away to plain dependency graph representation. The CASE model is based on the idea of *pipes and filters* pattern that describes an architectural style in which a large task is split into smaller processing components called filters that are interconnected using channels, i.e., pipes (Hohpe and Woolf, 2004). Similarly, a typical serverless application topology comprises a set of processing components, e.g., FaaS-hosted functions, storage or messaging services, connected by means of various types of channels, e.g., representing event transmission or direct invocations. Therefore, it is possible to represent a serverless application as a set of interconnected filters and pipes of various types that have different feature sets.

A UML class diagram shown in Figure 2 depicts the introduced CASE model. Essentially, in this model a serverless application comprises one or more *filters* of different type, e.g., functions or service components. Filters of type *Function* are described with various function-related properties, e.g., source code's programming language. Similarly, filters of type *Service* represent different kinds of services in a serverless application topology, e.g., databases, message queues or streaming platforms. In addition, a service might also serve as an event source of various categories such as schedulers or object stores, which is described using a service type *EventSource*. Each filter has zero or more incoming and outgoing *pipes* representing different communication mechanisms, e.g., event-driven with the specification of which events trigger the function using the relation-based event specification approach (Wurster et al., 2018) or direct invocations. The events are specified using CloudEvents, a provider-agnostic event format[4]. One additional connection type that is often not shown in the model explicitly is when a function connects to service from the source code. This type of connections can be identified either by analyzing the security credentials defined/used in the deployment model, or via the static code analysis. In addition, every filter has a type-specific configuration, e.g., function memory limits and invocation schedule.

[4]https://cloudevents.io

3.3 Step 3: Assess Portability of the Application's Model and Code

Assess the portability of a given deployment model to the target provider and analyze the source code of its artifacts for possible migration pitfalls.

To assess portability of a given application to the selected target provider or platform, two parallel tasks have to be performed: (i) the deployment model has to be analyzed with respect to its components' suitability, and (ii) code artifacts have to be analyzed to identify dependencies not covered by the SEAPORT CASE model such as provider-specific SDKs usage or direct HTTP calls to third-party services. CASE model is used as an input for deployment model suitability evaluation, whereas the provider-specific code artifacts from the application's deployment package serve as an input for code analysis. During the model suitability assessment, the components of the application are classified, e.g., functions, database and messaging services, mapped to the service alternatives of the target platform, followed by the deployment model similarity score calculation. The source code analysis task serves multiple purposes including library dependencies identification, code patterns search to find provider-specific fragments to facilitate the boilerplate code generation.

Step 3a: Model Portability Evaluation

As a preliminary step, we describe how components of the application are classified to allow identification of target provider's service alternatives.

Service Classification. Essentially, each service can (i) be an *event source* producing events that trigger one or more functions in the application topology, (ii) an *invoked service*, which can communicate with other components but does not trigger any functions, or (iii) *combine both roles.* There are multiple possible categories of event sources relevant for serverless applications, including various storage types, messaging and streaming platforms, or endpoints. Table 1 demonstrates exemplary service alternatives for a set of commonly-used event source categories offered by several commercial platforms as well as an example of alternatives for OpenFaaS, an open source Kubernetes-based FaaS platform. We describe these categories and list their basic properties relevant for calculation of the coverage score in the following.

A. Endpoints. One of the most common ways to invoke FaaS-hosted functions is via HTTP calls. Often, functions are exposed as endpoints by means of an API Gateway, which is responsible for calling functions that are bound to a particular event type, e.g., HTTP GET event. Note that there is subtle difference between an event-driven HTTP-based invocation and a direct HTTP call. The former is typically achieved via subscribing functions to specific HTTP events that are processed by an API Gateway. In contrast, the latter is achieved by invoking endpoints from within the source code, e.g., AWS Lambda Invoke API. The endpoint event source type covers only event-driven HTTP calls. The main properties that characterize this event source are:

- *Method:* request method for the given endpoint
- *Path:* request path for triggering given functions
- *Auth:* authentication-related configurations.

The service alternatives for this category shown in Table 1 can be described using a combination of these generic properties serving as an assessment baseline for endpoint event sources.

B. Object Stores. The next common event source category is the object storage, a cloud data store type focusing on file-level data abstractions (Mansouri et al., 2018), e.g., AWS S3 or Azure Blob Storage. As a baseline, we consider the following properties, shared by services in this category:

- *DataContainer ID:* storage instance identifier, e.g., S3 bucket or Azure blob
- *Event types:* list of events triggering functions, e.g., PUT or DELETE events
- *API:* interfaces supported by the service.

Exemplary alternatives for the object storage category are listed in Table 1.

C. Non-relational Stores. Non-relational data store type, which groups together NoSQL databases such as AWS DynamoDB or Apache Cassandra, is one more service category common for serverless applications. As a baseline set of properties for the category of non-relational stores, we define:

- *Database ID:* unique identifier of the database instance/partition
- *Type:* supported database models, since one service might support more than one like CosmosDB provides functionalities of, e.g., document, key-value, or wide column stores
- *API:* interfaces supported by the service
- *Query languages:* supported query languages.

Table 1 shows example alternatives for the non-relational store category.

Table 1: Exemplary service alternatives for common event source categories offered by several well-known FaaS platforms.

Category	AWS Lambda	IBM Cloud Functions	Azure Functions	OpenFaaS
Endpoint	AWS API Gateway	IBM API Gateway	API Management	OpenFaaS API Gateway
Schedule	AWS CloudWatch Events	IBM Alarms	Azure Timer Trigger	Cron
Object Storage	AWS S3	Object Storage	Azure Blob Storage	Min.io
NoSQL	AWS DynamoDB	Cloudant	CosmosDB	Cassandra, MongoDB
PubSub Messaging	AWS SNS	Event Streams	Event Grid	NATS
Event Streaming	AWS Kinesis	Event Streams	Event Hubs	Kafka
Point-to-Point Messaging	AWS SQS	Event Streams	Queue Storage	RabbitMQ

D. Schedulers. Another type of event sources are time-based job schedulers such as cron[5]. Scheduled invocation of functions is a common use case for serverless applications (Cloud Native Computing Foundation (CNCF), 2018) offered as an option by most FaaS platforms. Typically, provider use various mechanisms supporting scheduled function invocations, e.g., AWS CloudWatch events or Azure Time Trigger which internally might also rely on cron, but the format of cron expressions might differ, e.g., the specification of time intervals. For schedulers, we define the following baseline set of properties:

- *Cron:* describes, whether cron jobs are supported
- *Cron type:* which cron format is supported, e.g., cron for .NET or crontab(5)
- *Interval:* support for recurring jobs execution based on specified time intervals
- *Once:* once-in-a-lifetime jobs support.

Alternative ways to specify scheduled jobs for different FaaS platforms are listed in Table 1.

E. Streaming and Messaging. One more essential category of event sources for serverless applications comprises various streaming and messaging solutions. Various providers implements these products differently, e.g., Amazon provides AWS SQS for implementing point-to-point messaging, AWS SNS for implementing publish-subscribe messaging, and AWS Kinesis for streaming, whereas IBM allows implementing all of them using its Event Streams service. To describe this category of services, we define the following baseline properties:

- *Queue/Topic:* name or location of the given queue/topic
- *Batch size:* defines the number of events delivered as a bundle
- *Filter:* describes the event filtering policy.

Table 1 lists possible alternatives for different messaging and streaming options.

F. Invoked Services. Due to heterogeneity of invoked services, it is often impossible to find a suitable alternative, e.g., AWS Alexa, IBM Watson, or third-party services like GitHub. As a result, the same service must be also used in a ported application, preferably as-is. However, these components have to be analyzed in detail as in some cases using them as-is will not work, e.g., different authorization and authentication configuration requirements.

Deployment Model's Similarity Measure. Evaluating how portable the given application's model to the target provider, similarity of each component needs to be checked against available alternative offerings. This implies the descriptions of possible provider offerings are present and can be used for comparison. We envision the usage of knowledge bases that provide such information for similarity measure calculation, as in existing cloud migration works (Andrikopoulos et al., 2013b; Andrikopoulos et al., 2014). We elaborate more on this topic in the system architecture description in Section 4.

[5]http://man.openbsd.org/cron.8

To verify that a given application A with K distinct service categories is portable to the selected target provider X, we define the similarity measure as a weighted sum of portability scores for every involved service category:

$$C_X = \sum_{\kappa \in K} S_\kappa \cdot w_\kappa \quad (1)$$

Where:

S_κ: is the portability score for the service of type κ

w_κ: is the relevance factor for the service of type κ

While the assessment of a given component's portability also requires the static code analysis, from the structural similarity's perspective, at least two conditions must hold for the target provider: (i) one or more alternative categories for the given service must be offered by the target provider, and (ii) alternatives must support the same level of configurations, e.g., allowing specification of identical even triggers or other parameters. Therefore, we define the portability score S of a service of type κ with P_κ distinct configured properties with respect to the given alternative category α with T_α distinct properties as follows:

$$S_\kappa = \sum_{\rho \in P} x_\rho \cdot w_\rho \quad (2)$$

Where:

x_ρ: $\begin{cases} 1, & \text{iff } \exists t \in T_\alpha \ s.t. \ \rho \equiv t \\ 0, & otherwise \end{cases}$

w_ρ: is the relevance factor for the property ρ.

While it is possible to include the fact that several alternatives are present in the final similarity measure, we consider only the alternative with the highest portability score. In addition, the relevance factors for both, properties and service categories are introduced to make the decision making process more flexible. In the default state, we assume that all categories and properties have the same relevance factor.

Step 3b: Code Artifacts Analysis

As a next step after assessing model's portability, the available function's source code has to be analyzed. There are several important aspects needed to be checked including used libraries, embedded third-party component calls, and provider-specific code fragments, e.g., implementation of specific interfaces. The identified snippets are automatically annotated during static code analysis to be used for boilerplate code generation and model refinement steps. One important outcome of code artifacts analysis that can affect overall portability score is identification of a code fragment that completely prevents migration of the corresponding component, e.g., usage of incompatible libraries or calls to unsupported remote services.

Same as with model's similarity, the analysis of provider-specific code fragments must rely on the knowledge base which comprises positive and negative facts about target providers. Moreover, to augment this step, an interactive code exploration can follow the automated code analysis to allow users to annotate more advanced fragments and add them to the knowledge base for reuse.

Deciding on Application's Portability

The data resulting from model's similarity calculation and code analysis are combined and presented before generating the target application model. In many cases, making a strong conclusion based on these metrics is not possible, e.g., even if a service category has its portability score equal to zero, there might exist ways to move the component to a new target environment. For example, the component can be hosted using another cloud service model or reengineered. Therefore, the outcome of the evaluation step is presented to the user in a form of the portability report, i.e., components with portability score equals to zero, code snippets that are annotated as problematic or non-portable, and possible recommendations, which can be used as a complimentary input of the employed migration methodology.

3.4 Method Integration Point: Generate and Refine the Boilerplate Code

After verifying that the given application is portable and code artifacts have no incompatible fragments, a target application model can be generated. Hence, the evaluated CASE model is used to generate a deployment model structure for a target deployment automation technology, e.g., AWS Cloud Formation. One possible way to support multiple target transformations out-of-the-box is to use the Essential Deployment Meta Model (EDMM) (Wurster et al., 2019b; Wurster et al., 2019a) as an output of this step, i.e., transform the evaluated CASE model into a corresponding instance of EDMM and use the available tooling to generate the target deployment model. The annotated code artifacts obtained after the source code analysis step are used to generate the boilerplate code for respective function components in the target platform, e.g., generating prepared fragments that wrap the business logic in a provider-specific manner including implementation of specific interfaces and handling of events pre-processing to avoid locking into a provider-specific event format (Yussupov et al.,

2019a). The resulting deployment package can then be refined to include, e.g., security-related configurations, and used for further processing as a part of the employed cloud migration methodology.

4 ARCHITECTURE AND PROTOTYPICAL VALIDATION

In this section, we elaborate on the system architecture enabling SEAPORT method and describe its prototypical implementation. Additionally, we show prototype's output examples for a simple thumbnail generation application with respect to the method's steps.

4.1 System Architecture

Figure 3 shows the conceptual system architecture comprising three layers of components that enable the SEAPORT method. The interfaces layer is responsible for interaction with the system, e.g., by means of a command-line or graphical user interface. The business logic layer comprises three core components responsible for (i) Model Retrieval, (ii) Model Assessment, and (iii) Model Generation.

While model retrieval is not always needed, as discussed in Section 3.1 there might be cases when a model can be retrieved using the provider's interfaces. To support this, the architecture comprises a set of crawlers, e.g., allowing to crawl deployed AWS applications, managed by the retrieval controller.

The application portability assessment includes the assessment controller that manages (i) model assessment engine that calculates similarity measures for given deployment models with respect to the target provider and provides component mappings, and (ii) code analysis engine that consists of language-specific plugins for identifying important code fragments and annotating them. In addition, to simplify the mapping process, the system provides a mapping engine which supports defining technology-specific mapping rules in a form of templates, e.g., mapping rules for Terraform or AWS SAM.

The model generation components contains a respective controller that is responsible for managing model and code generation engines. These engines comprise technology-specific plugins for model and code generation, e.g., for generating an EDMM or Serverless model based on the given evaluated canonical model, or generating a Java boilerplate code fragments for wrapping the existing business logic and hosting it on a new provider.

Finally, the Resources layer includes the Artifacts Repository which is responsible for storing intermediate and final results, and the Portability Knowledge Base. The latter stores several different types of facts. Firstly, the knowledge base stores provider-specific descriptions of service offerings that are used for identification of suitable alternatives and the overall model evaluation process. Moreover, it stores such information as provider-specific mapping rules, e.g., for used libraries, function signatures, or code fragments. Additionally, the offered interfaces must provide users with the possibility to register new and modify existing facts in the knowledge base to improve the coverage of possible portability cases.

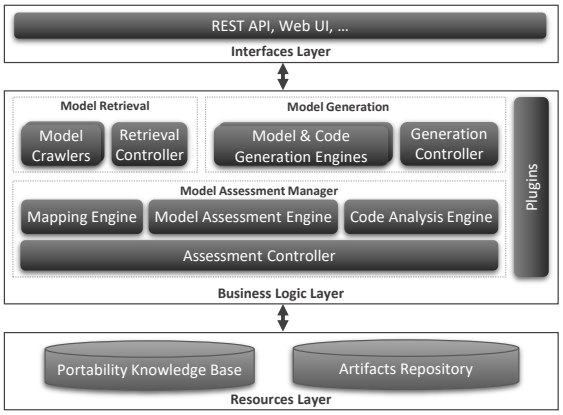

Figure 3: A conceptual system architecture of the portability evaluation and preparation system.

4.2 Prototypical Implementation and Example Code Excerpts

To validate the introduced SEAPORT method and system architecture, we implemented Skywalker[6], an open source web application which consists of the frontend and backend components, and is available on GitHub. In Skywalker, the backend component implemented using Java Spring communicates by means of a REST API with the frontend developed in Angular. For the runtime processing tasks we use a combination of the file system and H2, an in-memory database. Additionally, as a persistence layer we use MongoDB which is repsonsible for two repositories, namely for (i) Service Mappings, and (ii) Service Property Mappings. For the static code analysis we use JavaParser, a library for parsing and analysis of Java code. The source-to-target model assessment and boilerplate code generation is implemented for models defined using Serverless Framework for AWS Lambda and Azure Functions.

In the following, we show excerpts of model and code snippets related to porting a simple thumbnail

[6]https://github.com/iaas-splab/skywalker-prototype

generation application (Yussupov et al., 2019a) from AWS Lambda to Azure Functions. Listing 1 shows the input model defined using Serverless Framework, which specifies a function for uploading images into an images bucket, and a function for generating thumbnails and storing them in the output bucket.

Listing 1: Input deployment model of a thumbnail generation application for AWS Lambda defined using Serverless Framework.

```
service: thumbnail-generator
custom: {in: images-bucket, out: thmbnails-bucket}
provider:
  name: aws
  runtime: java8
  region: us-east-1
  iamRoleStatements:
    - Effect: Allow
      Action:
        - s3:*
      Resource: "*"
  environment:
    INBKT: ${self:custom.in}
    OUTBKT: ${self:custom.out}
package: {artifact: target/thmb-gen.jar}

functions:
  upload:
    handler: tst.UploadHandler
    events:
      - http:
          path: upload
          method: post
  thumbnail-generator:
    handler: tst.GenerationHandler
    events:
      - s3:
          bucket: ${self:custom.in}
          event: s3:ObjectCreated:*
resources:
  Resources:
    ThumbnailBucket:
      Type: AWS::S3::Bucket
      Properties:
        BucketName: ${self:custom.out}
```

When the input model is analyzed and transformed into the CASE model instance, the provider-specific information is abstracted away and the involved components are classified using generic category names as described in Section 3. An excerpt of the resulting generic model which describes component types and their required properties is shown in Listing 2.

Listing 2: CASE model obtained from the input deployment model of a thumbnail generation application.

```
{
  "_id": "thumbnail-generator",
  "eventSources": {
    "http": ["path", "methods"],
    "storage": ["resourceId", "events"]
  },
  "functions": {
    "thumbnail-generator": ["handler", "events"],
    "upload": ["handler", "events"]
  },
  "invokedServices": {
    "Action": [], ...
  }, ...
  // system properties
}
```

To generate the target boilerplate model, the CASE model is analyzed and mapped to the required provider-specific structure. Listing 3 shows a boilerplate model for porting the thumbnail generation application to Azure Functions using Serverless Framework in which the components and properties are mapped to the suitable alternatives. All models are stored separately in MongoDB and can eventually be reused, e.g., for generating the boilerplate model code for another target provider or platform.

Listing 3: A template of the output deployment model for Azure Functions defined using Serverless Framework.

```
service: thumbnail-generator
custom: {in: images-bucket, out: thmbnails-bucket}
provider:
  environment:
    INBKT: ${self:custom.in}
    OUTBKT: ${self:custom.out}
  stage: dev
  name: azure
  runtime: java8
  location: West US
package: {artifact: target/thmb-gen.jar}
functions:
  thumbnail-generator:
    handler: tst.GenerationHandler
    events:
    - blob: {path: ''}
  upload:
    handler: tst.UploadHandler
    events:
    - http: {route: '', methods: ''}
```

Listing 4: Example snippets of the annotated source code.

```
import ...lambda.runtime.Context; // <=={Context}
import ...s3.model.S3Object;      // <=={S3Object}

public class ThumbnailGenerationHandler implements
  RequestHandler<S3Event, Void> { // <=={RequestHandler}
                                  // <=={S3Event}
...
private ObjectMapper mapper = new ObjectMapper();
private AmazonS3Client c= ...; // <=={AmazonS3Client}...
```

Additionally, the prototype annotates functions' source code to highlight usage of provider-specific libraries. Examples of annotated Java function for AWS Lambda are shown in Listing 4.

5 RELATED WORK

To the best of our knowledge, there exist no works on serverless portability assessment, also with no related mentions in a relevant, recently-published systematic mapping study (Yussupov et al., 2019b). Several works focus on deployment and configuration modeling, e.g., using cloud modeling languages like TOSCA (Wurster et al., 2018) or CAMEL (Kritikos et al., 2019), or UML Profile (Samea et al., 2019) which also defines events including provider-specific types, e.g., AWS Kinesis. An abstract, graph-based model representing a serverless dependency

graph (Winzinger and Wirtz, 2019) is used for testing and verification purposes. The SEAPORT CASE aims to describe serverless applications in a language- and technology-agnostic way, and independently of context to enable translation into, e.g., more concrete deployment models or abstract representation models.

A systematic study by Silva et al. (Silva et al., 2013) investigates how cloud lock-in is solved in research literature. Opara-Martins et al. (Opara-Martins et al., 2014) discuss several kinds lock-in and such problems as integration and data portability. Lipton (Lipton, 2012) discusses how vendor lock-in can be avoided by using TOSCA, the cloud modeling language standardized by OASIS. Hohpe (Hohpe, 2019) presents different lock-in types, with vendor lock-in problem being one of them. Miranda et al. (Miranda et al., 2012) present software adaptation methods for overcoming the vendor lock-in problem. Authors describe the relations between service mismatch types and suitable adaptation approaches on the high level. As a possible application, this work can be used as a basis for extending the assessment mechanisms. Andrikopoulos et al. (Andrikopoulos et al., 2013a; Andrikopoulos et al., 2013b; Andrikopoulos et al., 2014) provide an analysis of migration challenges of the decision-making process for migrating applications to the cloud. Various classes of requirements, e.g., multi-tenancy, elasticity, quality of service, are analyzed and combined into a decision support framework for cloud migration. Frey and Hasselbring (Frey and Hasselbring, 2011) introduce a multi-phase approach for migrating legacy software systems to IaaS and PaaS, including such phases as extraction, selection, evaluation, transformation and adaptation. Binz et al. (Binz et al., 2011) present the cloud migration framework that analyzes possible hosting options for the provided model of an application. Strauch et al. (Strauch et al., 2015) elaborate on a vendor-agnostic multi-phase process enabling the migration of a database layer to the cloud. Beslic et al. (Beslic et al., 2013) propose a multi-phase approach for migrating components across providers comprising discovery, transformation, and migration steps. In this work, we rely on existing knowledge to introduce a method covering the specifics of serverless portability assessment and which can be used as a complementary part in larger migration approaches.

6 CONCLUSION

In this work, we presented SEAPORT, a multi-step method for assessing the portability of serverless applications. The core contributions of this work are (i) a canonical serverless application model, (ii) portability assessment concept relying on the deployment model similarity measure and static code analysis, and (iii) a system architecture enabling the method. We validated SEAPORT by implementing an open source prototype available via GitHub. As the next step, we plan to add support for more providers, and evaluate our method using several heterogeneous serverless use case applications. In future work, we will extend the SEAPORT CASE model to support additional usage scenarios, e.g., reasoning on platform selection with respect to specific platform features such as function orchestration support. Another important enhancement is to introduce additional application similarity measures, and standardize the format of knowledge bases with serverless portability facts by adapting existing migration methodologies. The latter can also help defining a set of thorough guidelines for improving portability of serverless applications. Additionally, we plan to support user-driven boilerplate code generation which requires enhancing the system with additional user interfaces.

ACKNOWLEDGMENTS

This work is partially funded by the European Union's Horizon 2020 research and innovation project *RADON* (825040). We would also like to thank the anonymous referees, whose feedback helped improving this paper.

REFERENCES

Andrikopoulos, V., Binz, T., Leymann, F., and Strauch, S. (2013a). How to adapt applications for the cloud environment. *Computing*, 95:493–535.

Andrikopoulos, V., Darsow, A., Karastoyanova, D., and Leymann, F. (2014). Clouddsf–the cloud decision support framework for application migration. In *European Conference on Service-Oriented and Cloud Computing*, pages 1–16. Springer.

Andrikopoulos, V., Song, Z., and Leymann, F. (2013b). Supporting the migration of applications to the cloud through a decision support system. In *Proceedings of the 6th IEEE International Conference on Cloud Computing (CLOUD 2013), June 27-July 2, 2013, Santa Clara Marriott, CA, USA*, pages 565–572. IEEE Computer Society.

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., et al. (2017). Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer.

Beslic, A., Bendraou, R., Sopenal, J., and Rigolet, J.-Y. (2013). Towards a solution avoiding vendor lock-in to enable migration between cloud platforms. In *MDH-PCL@ MoDELS*, pages 5–14. Citeseer.

Binz, T., Breitenbücher, U., Kopp, O., and Leymann, F. (2013). Automated Discovery and Maintenance of Enterprise Topology Graphs. In *Proceedings of the 6th IEEE International Conference on Service Oriented Computing & Applications (SOCA 2013)*, pages 126–134. IEEE Computer Society.

Binz, T., Leymann, F., and Schumm, D. (2011). Cmotion: A framework for migration of applications into and between clouds. In *2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–4. IEEE.

Bozman, J. and Chen, G. (2010). Cloud computing: The need for portability and interoperability. *IDC Executive Insights*.

Castro, P., Ishakian, V., Muthusamy, V., and Slominski, A. (2019). The server is dead, long live the server: Rise of serverless computing, overview of current state and future trends in research and industry. *arXiv preprint arXiv:1906.02888*.

Cloud Native Computing Foundation (CNCF) (2018). CNCF Serverless Whitepaper v1.0. Available online: https://github.com/cncf/wg-serverless/tree/master/whitepapers/serverless-overview.

Endres, C. et al. (2017). Declarative vs. imperative: Two modeling patterns for the automated deployment of applications. In *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*, pages 22–27. Xpert Publishing Services (XPS).

Frey, S. and Hasselbring, W. (2011). The cloudmig approach: Model-based migration of software systems to cloud-optimized applications. *International Journal on Advances in Software*, 4(3 and 4):342–353.

Greenstein, S. M. (1997). Lock-in and the costs of switching mainframe computer vendors: What do buyers see? *Industrial and Corporate Change*, 6(2):247–273.

Hellerstein, J. M., Faleiro, J., Gonzalez, J. E., Schleier-Smith, J., Sreekanti, V., Tumanov, A., and Wu, C. (2018). Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*.

Hohpe, G. (2019). Don't get locked up into avoiding lock-in. Available online: https://martinfowler.com/articles/oss-lockin.html.

Hohpe, G. and Woolf, B. (2004). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional.

Kritikos, K., Skrzypek, P., Moga, A., and Matei, O. (2019). Towards the modelling of hybrid cloud applications. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 291–295. IEEE.

Lipton, P. (2012). Escaping vendor lock-in with tosca, an emerging cloud standard for portability. *CA Labs Research*, 49.

Mansouri, Y., Toosi, A. N., and Buyya, R. (2018). Data storage management in cloud environments: Taxonomy, survey, and future directions. *ACM Computing Surveys (CSUR)*, 50(6):91.

Miranda, J., Murillo, J. M., Guillén, J., and Canal, C. (2012). Identifying adaptation needs to avoid the vendor lock-in effect in the deployment of cloud sbas. In *Proceedings of the 2nd International Workshop on Adaptive Services for the Future Internet and 6th International Workshop on Web APIs and Service Mashups*, pages 12–19. ACM.

Opara-Martins, J., Sahandi, R., and Tian, F. (2014). Critical review of vendor lock-in and its impact on adoption of cloud computing. In *International Conference on Information Society (i-Society 2014)*, pages 92–97. IEEE.

Petcu, D. (2011). Portability and interoperability between clouds: challenges and case study. In *European Conference on a Service-Based Internet*, pages 62–74. Springer.

Samea, F., Azam, F., Anwar, M. W., Khan, M., and Rashid, M. (2019). A uml profile for multi-cloud service configuration (umlpmsc) in event-driven serverless applications. In *Proceedings of the 2019 8th International Conference on Software and Computer Applications*, pages 431–435.

Satzger, B., Hummer, W., Inzinger, C., Leitner, P., and Dustdar, S. (2013). Winds of change: From vendor lock-in to the meta cloud. *IEEE internet computing*, 17(1):69–73.

Silva, G. C., Rose, L. M., and Calinescu, R. (2013). A systematic review of cloud lock-in solutions. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, volume 2, pages 363–368. IEEE.

Strauch, S., Andrikopoulos, V., Karastoyanova, D., and Vukojevic-Haupt, K. (2015). Migrating escience applications to the cloud: methodology and evaluation. *Cloud Computing with e-Science Applications*, pages 89–114.

Stravoskoufos, K., Preventis, A., Sotiriadis, S., and Petrakis, E. G. (2014). A survey on approaches for interoperability and portability of cloud computing services. In *CLOSER*, pages 112–117.

Winzinger, S. and Wirtz, G. (2019). Model-based analysis of serverless applications. In *2019 IEEE/ACM 11th International Workshop on Modelling in Software Engineering (MiSE)*, pages 82–88. IEEE.

Wurster, M., Breitenbücher, U., Brogi, A., Falazi, G., Harzenetter, L., Leymann, F., Soldani, J., and Yussupov, V. (2019a). The EDMM Modeling and Transformation System. In *Service-Oriented Computing – ICSOC 2019 Workshops*. Springer.

Wurster, M., Breitenbücher, U., Képes, K., Leymann, F., and Yussupov, V. (2018). Modeling and Automated Deployment of Serverless Applicationsusing TOSCA. In *Proceedings of the IEEE 11th International Conference onService-Oriented Computing and Applications (SOCA)*, pages 73—80. IEEE Computer Society.

Wurster, M. et al. (2019b). The Essential Deployment Metamodel: A Systematic Review of Deployment Automation Technologies. *SICS Software-Intensive Cyber-Physical Systems*.

Yussupov, V., Breitenbücher, U., Leymann, F., and Müller, C. (2019a). Facing the Unplanned Migration of

Serverless Applications: AStudy on Portability Problems, Solutions, and Dead Ends. In *Proceedings of the 12th IEEE/ACM International Conference onUtility and Cloud Computing (UCC 2019)*, pages 273–283. ACM.

Yussupov, V., Breitenbücher, U., Leymann, F., and Wurster, M. (2019b). A Systematic Mapping Study on Engineering Function-as-a-Service Platforms and Tools. In *Proceedings of the 12th IEEE/ACM International Conference onUtility and Cloud Computing (UCC 2019)*, pages 229–240. ACM.