Institute of Architecture of Application Systems

Facing the Unplanned Migration of Serverless Applications: A Study on Portability Problems, Solutions, and Dead Ends

Vladimir Yussupov, Uwe Breitenbücher, Frank Leymann, Christian Müller

Institute of Architecture of Application Systems,
University of Stuttgart, Germany,
{yussupov, breitenbuecher, leymann}@iaas.uni-stuttgart.de
muellercn@posteo.de

BibTeX:

```
@inproceedings{Yussupov2019_FaaSPortability,
  author    = {Vladimir Yussupov and Uwe Breitenb{\"u}cher and Frank Leymann
               and Christian M{\"u}ller},
  title     = {{Facing the Unplanned Migration of Serverless Applications: A
               Study on Portability Problems, Solutions, and Dead Ends}},
  booktitle = {Proceedings of the 12th IEEE/ACM International Conference on
               Utility and Cloud Computing (UCC 2019)},
  publisher = {ACM},
  year      = 2019,
  month     = dec,
  pages     = {273--283},
  doi       = {10.1145/3344341.3368813}
}
```


Universität Stuttgart
Germany

Facing the Unplanned Migration of Serverless Applications: A Study on Portability Problems, Solutions, and Dead Ends

Vladimir Yussupov
Institute of Architecture of Application Systems
University of Stuttgart, Germany
yussupov@iaas.uni-stuttgart.de

Uwe Breitenbücher
Institute of Architecture of Application Systems
University of Stuttgart, Germany
breitenbuecher@iaas.uni-stuttgart.de

Frank Leymann
Institute of Architecture of Application Systems
University of Stuttgart, Germany
leymann@iaas.uni-stuttgart.de

Christian Müller
Institute of Architecture of Application Systems
University of Stuttgart, Germany
muellercn@posteo.de

ABSTRACT

Serverless computing focuses on developing cloud applications that comprise components fully managed by providers. Function-as-a-Service (FaaS) service model is often associated with the term serverless as it allows developing entire applications by composing provider-managed, event-driven code snippets. However, such reduced control over the infrastructure and tight-coupling with provider's services amplify the various lock-in problems. In this work, we explore the challenges of migrating serverless, FaaS-based applications across cloud providers. To achieve this, we conduct an experiment in which we implement four prevalent yet intentionally simple serverless use cases and manually migrate them across three popular commercial cloud providers. The results show that even when migrating simple use cases, developers encounter multiple aspects of a lock-in problem. Moreover, we present a categorization of the problems and discuss the feasibility of possible solutions.

CCS CONCEPTS

• **Software and its engineering** → **Cloud computing**; • **Applied computing** → *Event-driven architectures.*

KEYWORDS

Serverless; Function-as-a-Service; FaaS; Portability; Migration

ACM Reference Format:
Vladimir Yussupov, Uwe Breitenbücher, Frank Leymann, and Christian Müller. 2019. Facing the Unplanned Migration of Serverless Applications: A Study on Portability Problems, Solutions, and Dead Ends. In *Proceedings of the IEEE/ACM 12th International Conference on Utility and Cloud Computing (UCC '19), December 2–5, 2019, Auckland, New Zealand*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3344341.3368813

UCC '19, December 2–5, 2019, Auckland, New Zealand

ACM ISBN 978-1-4503-6894-0/19/12…$15.00
https://doi.org/10.1145/3344341.3368813

1 INTRODUCTION

Cloud computing [20] becomes a necessity for businesses at any scale. The key benefits motivating its widespread adoption are elasticity, reduced maintenance efforts, and optimized costs [18]. Traditional cloud service models provide different levels of control over the underlying resources, from more flexible to more restricted, e.g., where only the underlying platform can be controlled. Such a broad spectrum of service models facilitates choosing which management responsibilities are eventually delegated to providers.

Serverless computing paradigm [5, 12] focuses on development of applications comprising provider-managed components. Hence the term *serverless* is misleading as it gives a wrong impression that servers are absent, while in reality they are only abstracted away. Function-as-a-Service (FaaS) offerings such as AWS Lambda [1] are often called serverless, since developers are able to deploy arbitrary, event-driven code snippets managed completely by providers, e.g., functions are automatically scaled if necessary; also including scaling idle instances to zero. However, these benefits come with a price: the reduced infrastructure control and, hence, the tighter-coupling with provider-specific internal mechanisms amplify the lock-in. Combining FaaS-based components with provider services introduces even more portability risks, e.g., identification of suitable service alternatives, adaptation of the configuration and business logic. Hence, migrating a serverless, FaaS-based application requires considerable efforts and appropriate technical expertise.

In this work, we investigate challenges encountered when migrating serverless, FaaS-based applications not built with portability in mind. To achieve this, we conducted an experiment in which we implement and manually migrate four prevalent serverless, FaaS-based use case applications across three well-known commercial cloud providers. The applications are intentionally simplified, to show that even the migration of simple serverless use cases faces multiple aspects of lock-in problem and to ease experiment's reproducibility. To achieve this, we (i) designed and implemented four typical serverless use cases for a baseline implementation provider and migrated them manually to two target providers. To report on our findings, we (ii) conduct a detailed analysis of arising problems and present a categorization of different *lock-in types* we faced. Moreover, we (iii) describe for all categories of encountered lock-ins possible solutions and report which problems result in a dead-end, meaning there is no acceptable cloud-native solution for this issue.

2 BACKGROUND

In this section, we describe the important fundamentals including serverless computing, FaaS, function orchestration and vendor lock-in. Furthermore, we discuss the motivation behind this work and outline the design of the conducted experiment.

2.1 Serverless and Function-as-a-Service

The overall intention behind the concept of serverless computing is to completely abstract away servers from software developers [12]. While actual servers still remain present, the duty of managing aspects such as provisioning and scaling resources is lifted away from the developer and done by the cloud provider instead. As a result, developers are able to focus solely on implementing the applications. Function-as-a-Service (FaaS) fits nicely into the world of serverless. In FaaS, applications are composed of multiple, short-lived and, typically, stateless functions that are triggered based on specific events, e.g, a database insert operation or whenever a message is published to a queue's topic. Among the main benefits of this service model are auto-scaling guaranteed by the provider and an optimized cost model, where instead of paying for idle times cloud providers only charge tenants for the time functions have actually been executed [5, 11, 12]. This means that idle instances are scaled to zero, which is not the case, e.g., for PaaS offerings. However, FaaS also comes with drawbacks, such as the so-called cold start problem or limited function execution time [16].

The lack of long-running tasks support in FaaS [16] is a major drawback, especially for more complex use cases. One potential solution to this problem is the decomposition of a large function into a chain of smaller functions modeled as a workflow that can be orchestrated by a function orchestrator [14], e.g., Microsoft Azure Durable Functions [21]. The main task of a function orchestrator is to execute workflows based on FaaS-hosted functions. While function orchestrator feature sets differ among cloud providers, the majority supports control flow structures like chaining, branching, and parallel execution of functions, which facilitate modeling of long-running tasks and make FaaS offerings more powerful.

2.2 Lock-in Problem

The inevitability of locking into requirements and features of a specific product is a well-known problem which can be encountered in multiple different migration contexts, e.g., switching between various competing mainframe vendors [15] or cloud computing offerings [8, 25, 27]. In the cloud, the lock-in is a common problem that complicates the migration of cloud applications from one provider to another [25]. Basically, every provider-specific offering comes with varying sets of features and requirements which have to be fulfilled to actually start using it. This includes specific data formats, remote APIs, or even custom configuration DSLs and programming languages [25]. As a result, by choosing a specific provider, cloud users become tightly-coupled with the corresponding services and features, which makes it significantly harder costs- and efforts-wise to switch cloud providers in cases when it is needed, e.g., organizational restructuring or costs optimization [25, 30]. Moreover, cloud applications are typically not built with portability in mind [25], and by choosing the cloud provider, applications become (un-)willingly restricted to the features of chosen provider.

2.3 Why Porting Serverless Applications?

The issues of portability and interoperability of applications in the cloud are well-known, with multiple papers describing why moving applications across clouds is important and beneficial and proposing ways to simplify this process [10, 26, 30]. Basically, there are numerous potential reasons to switch provider for already deployed applications including optimization of resources utilization and costs, depreciation of the service quality or provider's bankruptcy, changes of technology, terminated contracts, or legal issues [26]. While the lock-in problem is not novel, the fast pace of cloud service models' evolution and constantly-increasing number of provider-specific services makes it harder to understand which issues to expect when porting applications across clouds. With serverless and FaaS this problem is further exacerbated, since all application components are intended to be provider-managed, which results even in a stronger degree of lock-in. For example, the event-driven nature of FaaS imposes additional requirements on the way functions have to be developed and configured, e.g., which events can trigger them, how to establish bindings between event sources and actual functions, etc. Moreover, function orchestration engines that allow composing FaaS-hosted functions often work differently and support different sets of features, which further complicates migrating serverless applications. In addition, established services, e.g., databases, used as serverless components have similar problems discussed in multiple research publications [8, 22].

3 EXPERIMENT DESIGN

In this work, we explore and categorize the different dimensions of the lock-in problem for serverless applications and answer the following research question: *"Which technical challenges are commonly encountered when migrating serverless, FaaS-based applications that are not built with portability in mind across commercial cloud providers?"* To answer this question, we design and conduct an experiment comprising these steps: (i) select four common and intentionally-simple serverless, FaaS-based use cases, (ii) choose a baseline implementation provider, (iii) manually migrate the baseline implementations to another two commercial providers, (iv) analyze and categorize different lock-in types, possible solutions and dead ends. When designing the use cases, we favored two decisions: maximize components' heterogeneity and keep the implementations simple. The main reason for choosing components to be simple is to demonstrate that even such simplified use cases introduce multiple portability issues. To select four use cases, we first analyzed existing academic and gray literature [5, 11–13, 16] and identified commonly-described use cases such as event processing and API composition. It is important to note that we did not intend to gather a complete set of possible use cases, but rather a representative set which will allow us to design uncomplicated yet heterogeneous serverless, FaaS-based applications. Moreover, to emulate the real world scenarios requiring an unplanned migration, these use cases are implemented without optimizing them for portability and interoperability. The final list of use cases includes: one simple and one more advanced event processing application, serverlesss API, and a function workflow. We elaborate on the use cases in Section 4. For the baseline implementation we use AWS, and the two other providers are Microsoft Azure and IBM Cloud.

4 USE CASES

In this section, we describe the chosen use cases typically associated with the serverless and FaaS [5, 11], which we use in the migration experiment as described in Section 3. The components which constitute the use cases are chosen to be heterogeneous to cover different event trigger types such as relational and NoSQL databases, message queues, and API Gateways. Moreover, as function orchestration becomes an important part of FaaS [6, 14], we include the matrix multiplication use case implemented by means of function workflows. The chosen set of use cases is not planned to be complete, and represents typical yet heterogeneous serverless, FaaS-based scenarios.

4.1 Thumbnail Generation

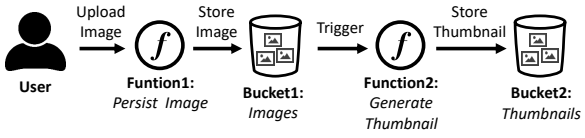

Figure 1. The components of a thumbnail generation application

The first use case shown in Figure 1 is a straightforward and most frequently-described example of an event-driven FaaS-based application: thumbnail generation. In this common use case, whenever a user uploads an image to a bucket in an object storage, the FaaS-hosted function is triggered to generate a thumbnail of this image and store it in a separate bucket. In addition, for facilitating the image uploading process, a separate function is responsible for uploading an image to a designated bucket.

4.2 Serverless API

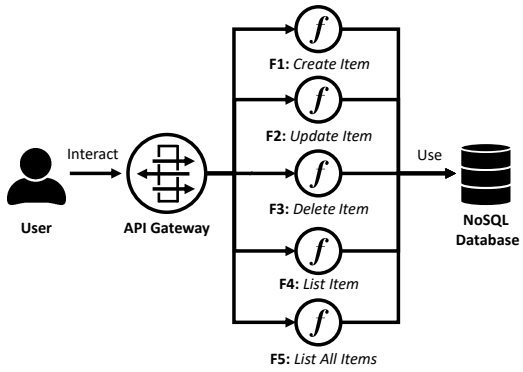

Figure 2. A serverless API of a ToDo Application

Another common use cases involving FaaS API composition and aggregation [5]. Essentially, the idea is to expose FaaS-hosted functions representing, e.g., a composition of several API calls, via an API Gateway as a single of point communication with clients. For the sake of simplicity, we implement a ToDo list application that allows creating and maintaining the list of to-be-done tasks persisted in a NoSQL database. The structure of the application is shown in Figure 2. Here, we focus not on the complexity of the functions, but on their integration with the API gateway and the storage service, i.e., a NoSQL database in this example.

4.3 Event Processing

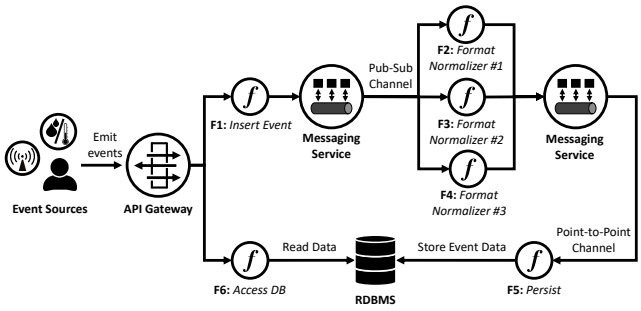

Figure 3. A messaging-based event processing application

The third use case depicted in Figure 3 is a more complex serverless and event-driven FaaS-based application motivated by IoT scenarios, where the data obtained from multiple sensors have to be aggregated for further processing. Typically, message queues and streaming platforms play an important role in such scenarios. The overall data flow of this application is as follows: Firstly, events emitted by various sources are processed by a function exposed via an API Gateway. Then, a publish-subscribe channel is used to normalize various event formats using corresponding *normalizer functions*. Afterwards, the normalized events are passed to a function that persists events to a RDBMS by means of a point-to-point message queue channel. Different types of channels are used to enable more possible provider-specific service combinations, e.g., AWS SNS for pub-sub and AWS SQS for a point-to-point channel.

4.4 Function orchestration

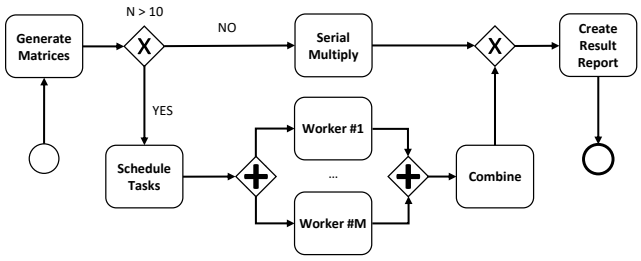

Figure 4. Matrix multiplication using function orchestration in BPMN

Typically, function orchestrators are available as standalone services that are offered to complement FaaS offerings, e.g., AWS Step Functions or Microsoft Azure Durable Functions. Function orchestrators allow defining function-based workflows, often in custom DSLs, that are highly convenient in more complex serverless application scenarios [6, 14]. For an orchestration use case, we model a matrix multiplication using a simple function workflow as specified in the BPMN [24] diagram depicted in Figure 4. The modeled workflow comprises exclusive and parallel gateways to perform matrix multiplication in parallel only for cases when matrices dimensionality is larger than ten. This condition is introduced to increase the number of distinct control flow constructs, which allows analyzing the aspects of orchestrators' workflow portability in more detail.

5 MIGRATING THE USE CASES

In this section, we elaborate on the implementation and migration of our intentionally-simple use case applications described in Section 4. We first discuss the baseline implementation in Amazon Web Services (AWS) for each of the use case applications. To emulate real world scenarios, we implement the use cases in different programming languages without optimizing them for portability and interoperability. Afterwards, we show how these baseline implementations are migrated to the chosen target providers, namely Microsoft Azure and IBM Cloud. For every use case we first provide descriptions of technical details for each target provider followed by a summary table with modifications needed for migrating the baseline implementation to each target provider. In every case, the changes and adaptation of function and configuration code was necessary. When describing the implementation details, we omit dry technical information such as the number of lines of code, and focus rather on important migration aspects and challenges. The source code of all implementations is publicly available on GitHub[1].

5.1 Thumbnail Generation

As described in Section 4, creating thumbnails using FaaS is a simple event-driven application that typically relies on FaaS-hosted thumbnail-generating function triggered by certain database events, e.g., emitted when images are inserted into an object storage. We describe the technical details related to every provider below.

Implementation for Amazon Web Services. For AWS Lambda in the thumbnail generation use case we chose Java programming language and Apache Maven for dependency management. This use case comprises two functions: one for generating thumbnails and one for storing an image into the database. For the database we use AWS S3 object storage service and two buckets for storing uploaded images and thumbnails separately. The trigger for a thumbnail generating function is configured for the bucket storing original, user-uploaded images. A common way to develop Java functions in AWS Lambda is to implement a *Requesthandler* interface and define function's inputs and outputs using generics. In addition, Java AWS provides a separate Java library[2] to work with S3 events. The choice of technologies and component types was not influenced by the decision to make application portable beforehand, instead, we picked rather a generic set of components, i.e., Java functions and AWS S3 buckets.

Migrating to Microsoft Azure. To migrate the thumbnail generation use case to Microsoft Azure, we use Azure Functions FaaS platform and Azure Blob Storage as an alternative to AWS S3. Azure Functions allows hosting functions written in Java making it possible to port the Java code. However, some modifications were needed due to provider-specific features, e.g., trigger configurations and handling event payload. For example, in Azure Functions, triggers are configured by means of Java annotations directly in the source code, whereas in AWS the triggers are defined externally. In addition, in AWS S3 events contain references to the actual data, i.e., images have to be accessed from the business logic itself, whereas Microsoft Azure handles this task transparently resulting in a less amount of code to be written. Essentially, Azure Blob storage is similar to AWS S3, with naming being one of the main differences: while in AWS buckets must have a globally-unique name, in Azure the name must only be unique withing a storage account. This simplifies blob container naming in Azure as only the storage account must be globally-unique.

Migrating to IBM Cloud. Several interesting issues were encountered when migrating the thumbnail generation use case to IBM Cloud. FaaS offering from IBM internally uses Apache Openwhisk, an open source FaaS platform, which has no limitations in terms of supported programming languages. As a result, the migration of Java functions was also possible, however, some modifications were needed. For example, Openwhisk does not support using custom objects as function's inputs and outputs, requiring to process input and output dictionaries represented as JSON objects in the business logic. IBM Object storage was used as an alternative to AWS S3. At the time the use case was implemented, a trigger required for binding the storage with the function was available only in one region and considered to be an experimental feature[3].

Table 1 shows a high-level overview of which changes were needed when migrating thumbnail generation application from AWS to Microsoft Azure and IBM Cloud. Note that the reported modification requirements are unidirectional, i.e., from baseline to target provider, and it is not necessarily true that the opposite direction would have the identical requirements. For example, if the baseline provider allows developing functions using only a subset of languages supported by the target provider, modification will not be needed when porting the application from the baseline to target provider. Conversely, porting the application *from* the target to baseline provider might require changing the language.

Table 1. Summary of changes needed to migrate the thumbnail generation use case from AWS to Microsoft Azure and IBM Cloud

Modification	Microsoft Azure	IBM Cloud
Change implementation language[1]	No	No
Adapt function code	Yes	Yes
Change configurations	Yes	Yes
Change application's architecture	No	No[2]

[1] Use case functions' code is implemented in Java
[2] Implementation relies on the experimental trigger available only in one region

5.2 Serverless API

Employing serverless, FaaS-based API can be attractive in various scenarios, e.g., composing multiple third-party API calls using a single FaaS function. To analyze the portability aspects for this use case, we implement a simplified FaaS-based API of a simple ToDo application that uses a NoSQL database in the backend as described in Section 4. As a next step, we provide more details for every provider in the respective subsections.

[1] https://github.com/iaas-splab/faas-migration
https://github.com/iaas-splab/faas-migration-go
[2] https://mvnrepository.com/artifact/com.amazonaws/aws-lambda-java-events

[3] https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-pkg_obstorage#pkg_obstorage_ev_ch

Implementation for Amazon Web Services. FaaS-hosted functions for this use case are implemented in Go, a statically-typed, compiled language introduced by Google, which gains popularity in cloud-native development community. To write functions in Go, AWS provides an SDK for interacting with AWS services and a package containing code for function invocation and event structures. For the NoSQL storage component, we use AWS DynamoDB database and the functions are exposed via AWS API Gateway. As mentioned previously, the choice of the programming language and architecture components was made without an intention to optimize the application for portability, but rather to emulate realistic scenarios in which the choice is defined by the organization requirements, team skills, etc. As a result, the fact that Go is not supported by Azure Functions was unexpected as the AWS implementation was built without checking other provider beforehand.

Migrating to Microsoft Azure. Unlike other chosen providers, Microsoft Azure Functions does not support Go, which resulted in all functions being reimplemented from scratch, in this case in C#. The development workflow for C# in Azure Functions is similar to Java, since the configuration of triggers and interaction with the storage is done directly in the source code using attributes, i.e., C# analog of Java annotations. As an alternative to AWS DynamoDB in Azure, we use Azure Table Storage, which suits nicely the given use case's requirements. It is worth mentioning that there are other suitable Azure offerings, e.g., Azure CosmosDB, making the choice of alternatives flexible, but also more complicated. Finally, API Management is used to combine Azure HTTP-functions.

Migrating to IBM Cloud. Since Openwhisk, and, therefore, IBM Cloud Functions platform supports Go, a reimplementation of the baseline implementations was not needed. Moreover, due to the nature of function's inputs and outputs processing in Openwhisk there is no need to use specific dependencies for implementing function in Go, making it generally simpler. To mimic AWS DynamoDB, we use IBM Cloudant database and to simplify communications with it, functions rely on the Cloudant-specific database driver. This dependency is substitutable, as IBM Cloudant also provides an API compliant with Apache CouchDB, which allows using a CouchDB-specific driver. As an alternative to AWS API Gateway, an Openwhisk API is created for IBM Cloud Functions manually, due to the small amount of functions. However, it is also possible to define an OpenAPI specification, which is helpful, as AWS API Gateway also allows defining APIs using OpenAPI specification. Table 2 shows an overview of required modifications.

Table 2. Summary of changes needed to migrate the serverless API use case from AWS to Microsoft Azure and IBM Cloud

Modification	Microsoft Azure	IBM Cloud
Change implementation language[1]	Yes[2]	No
Adapt function code	Yes	Yes
Change configurations	Yes	Yes
Change application's architecture	No	No

[1] Use case functions' code is implemented in Go
[2] All code had to be reimplemented from scratch, C# was used

5.3 Event Processing

The autoscaling feature of FaaS makes it attractive for data analytics scenarios. To further explore the portability aspects of event-driven serverless applications, we implement the application that processes events collected from different sources as described in Section 4.

Implementation for Amazon Web Services. This use case is implemented using JavaScript, a de-facto standard for FaaS-based development. As JavaScript is an interpreted and weakly-typed language, writing functions code is typically simpler. Event data can be accessed by specifying desired attribute names from the event's structure. The libraries for interacting with other AWS services do not need to be included into function packages, since NodeJS runtime provided by AWS Lambda already has them preinstalled. For message queue services we use two distinct offerings, namely AWS SNS and AWS SQS. The former is used for a publish-subscribe channel of the application, whereas the latter is used for implementing a point-to-point channel. For a relational database component we use Amazon Aurora with MySQL interface. Event insertion and database access functions are exposed via AWS API Gateway.

Migrating to Microsoft Azure. Unlike Java and C#, inline definition of triggers for JavaScript in Azure Functions is not supported. Instead, triggers have to be defined externally using a separate JSON file per function. As a suitable alternative for the relational database component we chose Azure Database for MySQL. The migration itself was mostly about changing the endpoints and credentials. Next, AWS SNS and AWS SQS components were ported to Azure Service Bus, which allows defining both publish-subscribe and point-to-point channels. Here, two subscriptions per function for reading from and writing to the pub-sub topic had to be created.

Migrating to IBM Cloud. One issue with IBM Cloud was the search of suitable alternative for publish-subscribe and point-to-point messaging services. While allowing to implement point-to-point channels, the message queue service IBM MQ does not provide a trigger for invoking functions based on the message arrival events. As a result, this component of the application was moved to a pub-sub channel, which can be implemented using IBM Event Streams service. Next, as a substitute for the relational database component we use IBM Compose for MySQL service, which can be considered a suitable alternative for AWS Aurora. While development of JavaScript functions was ordinary, the libraries for publishing to Kafka used in IBM Event Streams service were required. Table 3 shows an overview of required modifications.

Table 3. Summary of changes needed to migrate the event processing use case from AWS to Microsoft Azure and IBM Cloud

Modification	Microsoft Azure	IBM Cloud
Change implementation language[1]	No	No
Adapt function code	Yes	Yes
Change configurations	Yes	Yes
Change application's architecture	No	Yes[2]

[1] Use case functions' code is implemented in JavaScript
[2] Point-to-point channel had to be replaced with a publish-subscribe channel

5.4 Function Orchestration

The last use case implementation targeted the function orchestration application in a form of a workflow that defines matrix multiplication as described in Section 4. To implement this use case we used respective orchestration service offerings that allow defining function workflows for FaaS-hosted functions.

Implementation for Amazon Web Services. To implement functions for the function orchestration use case application we use C# programming language. The function workflow is defined and orchestrated using AWS Step Functions service offering. The definition of a workflow relies on the custom DSL provided by the service. Since AWS Step Function has a maximum allowed payload size of around 32000 characters, for multiplication of large matrices, e.g., with a dimensionality of several hundreds, we had to implement a caching mechanism for temporary storage of function's inputs and outputs. The execution of the workflow was exposed via API Gateway, allowing to trigger it using an HTTP call.

Migrating to Microsoft Azure. As a suitable orchestrator alternative from Microsoft, we use Azure Durable Functions. Here, the workflow is defined as code using one of the supported programming languages in a form of an orchestrator function. Unlike AWS Step Functions, this service does not set limits on the payload size and also caches large payloads in the background. Hence, the simple caching implemented for AWS was redundant here. Additionally, a function which starts the workflow is needed, which, in our case, was implemented using a standard HTTP-triggered function.

Migrating to IBM Cloud. IBM Composer, the suitable function orchestration service from IBM, required several modifications to successfully migrate our use case. Firstly, the workflow definition had to be rewritten in JavaScript, as IBM Composer only allows defining function workflows in this programming language. Moreover, IBM Composer also limits the payload size to 5Mb, which resulted in a need to implement a caching mechanism similar to AWS Step Functions implementation. Additionally, IBM Database for Redis service had to be used since IBM Composer dependency on Redis for caching orchestrations of parallel function executions.

Table 4 shows an overview of modifications required to migrate the baseline implementation of the function orchestration use case in AWS Step Functions to Microsoft Azure Durable Functions and IBM Composer. Of course, the FaaS-hosted functions were developed for the respective FaaS platforms, i.e., AWS Lambda, Azure Functions, and IBM Cloud Functions.

Table 4. Summary of changes needed to migrate the function orchestration use case from AWS to Microsoft Azure and IBM Cloud

Modification	Microsoft Azure	IBM Cloud
Change implementation language[1]	No	No
Adapt function code	Yes	Yes
Adapt workflow definition	Yes	Yes
Change configurations	Yes	Yes
Change application's architecture	No	No

[1] Use case functions' code is implemented in C#

5.5 Application Component Mappings

As can be seen from the aforementioned implementation details, every provider-specific components needs to be replaced with a suitable alternative from the target provider's portfolio. In Table 5 we provide an overview of possible component replacement that were used for use cases migration. In many cases, the choice of suitable alternatives is not limited with only one option, e.g., decide between Azure CosmosDB and Azure Table Storage, and making it more flexible, but also complicated.

Table 5. Chosen mappings of application components use cases migration

Component	Amazon Web Services	Microsoft Azure	IBM Cloud
Function	AWS Lambda	Azure Functions	IBM Cloud Functions
Object storage	AWS S3	Azure Blob Storage	IBM Object Storage
NoSQL database	AWS DynamoDB	Azure Table Storage	IBM Cloudant
API Gateway	AWS API Gateway	API Management	Openwhisk API
Relational database	Amazon Aurora	Azure Database for MySQL	IBM Compose for MySQL
Messaging: pub-sub channel	AWS SNS	Azure Service Bus	IBM Event Streams
Messaging: point-to-point channel	AWS SQS	Azure Service Bus	IBM Event Streams
Function orchestrator	AWS Step Functions	Azure Durable Functions	IBM Composer

6 LOCK-IN CATEGORIES, PROBLEMS, SOLUTIONS, AND DEAD-ENDS

In this section, we analyze the results of the conducted experiment as described in previous sections. As shown in our analysis, to a large extent, the nature of serverless computing and FaaS-based applications amplifies the lock-in degree. As expected, during the implementation and manual migration of the use case applications described previously we encountered multiple compatibility issues connected with the lock-in problem. More specifically, based on the encountered problems, we (i) derive different categories of lock-in, (ii) present possible solutions to these problems, and (iii) also explain where potential dead ends may occur. Here, by dead end we mean a solution that forces paradigm shift, i.e., when it is no longer possible to rely on existing cloud-native service alternatives and a new solution has to be build from scratch. In general, we faced a broad spectrum of migration problems, from easily-solvable to complete dead ends that require reimplementing specific parts of the application. In the following, we discuss different lock-in categories with respect to the compatibility scope, e.g., whether the lock-in is related to compatibility of distinct application components, the entire application, or application-related tooling, and outline the possible solutions and dead ends.

6.1 Compatibility of Application Components

One of the main problems is to identify compatible services for every component of a to-be-migrated serverless application. However, the term *compatible* here is rather vague, since often service's compatibility is determined by a degree of support for a set of required features. For instance, functions that are hosted on FaaS platforms typically (i) are implemented in a specific programming language, and (ii) are triggered by certain event types emitted from specific event sources. As a result, the target FaaS platform can be considered compatible if it supports the given combination of programming language, event types and sources, and provides built-in integration mechanisms for binding functions and event sources. Moreover, even if all required features are supported by the target provider's service alternative, various additional problems might occur, e.g., different implementation and configuration requirements might further complicate the migration process. In general, FaaS-based components play an important role in serverless applications, as they provide means to host the application's business logic. However, the underlying FaaS offerings vary significantly in their feature sets, resulting in multiple compatibility issues encountered when migrating all use cases described in Section 4. Additionally, other types of serverless components, e.g., databases, API Gateways, or messaging solutions offered as a service, might also be deemed incompatible due to various reasons.

6.1.1 ***Feature Set Lock-in.*** Identifying suitable service alternatives and overcoming feature-specific lock-ins are among the major problems encountered when migrating serverless applications.

Problem: *Each component of a deployed serverless application implicitly depends on a specific feature set of the underlying service.*

Examples: Multiple examples can illustrate this problem. For instance, one feature set lock-in example was encountered during migration of the event processing use case described in Section 4.3 from AWS to IBM Cloud. While IBM provides a separate message queue service called IBM MQ which can be used for implementing a point-to-point messaging, there is no built-in trigger for invoking a function hosted on IBM Cloud Functions based on the message receipt event. Since one feature from the required feature set is not supported, i.e., a specific trigger cannot be defined, the IBM MQ service cannot be considered a suitable alternative. It was possible to overcome this problem by turning a point-to-point channel into a pub-sub channel, however, for more complex real-world applications this might become a serious obstacle.

Another example of a feature set lock-in we encountered is the unsupported language for the serverless API use case described in Section 4.2. While initially this application was implemented in Go, it was not possible to migrate functions as-is to Azure Functions because Go is not supported by the platform. As a result, the only possible solution was to reimplement all functions.

Required feature set can also include management-related functionalities, e.g., in function orchestration use case described in Section 4.4. For example, AWS Step Functions provides a web UI for observing the execution state in detail, while IBM Composer allows exploring the execution state only using a CLI with a smaller set of capabilities. Depending on project requirements, such feature support discrepancy might also become a migration barrier. Migrating serverless applications to private clouds is even more complicated, since the service alternatives and their integration, e.g., with the chosen open source FaaS platforms, consumes more time and efforts, and can even be infeasible.

Possible solutions and dead ends: As we have seen previously, the strong feature set lock-in and absence of suitable alternatives might require reimplementing application's components. Feasibility of this decision depends on multiple factors including the size of the component, integration requirements, etc. Additionally, solutions like developing your own component's substitute or integrating with external third-party alternatives are likely to be infeasible too, e.g., implementation efforts, increased service costs. As a result, lack of compatible alternatives might be a migration dead end if implementation efforts are too costly.

6.1.2 ***Implementation Requirements Lock-in.*** This category is related to provider-specific requirements imposed on the way components must be implemented. Such kind of lock-in becomes an obvious problem for FaaS-based application components. For example, different FaaS platforms specify various requirements on the way functions have to be implemented, e.g., unlike IBM, AWS typically requires implementing a specific interface for Java functions and declare inputs and outputs using generics as shown in Listing 1. As such requirements must be fulfilled to use the target service, various dependency types are introduced in the respective component, which might become a serious portability issue.

Listing 1. Object storage-triggered Java functions on AWS and IBM

```
public class AWSGenerator implements
     RequestHandler<S3Event, Void> {
   @Override
   public Void handleRequest(S3Event input, Context c) {
      // code continues
   }
}

public class IBMGenerator {
   public static JsonObject main(JsonObject args) {
      // code continues
   }
}
```

Problem: *Fulfillment of provider-specific implementation requirements introduces multiple types of dependencies that must be taken into consideration as they might affect application's portability.*

Example: Implementation and packaging guidelines. In case of FaaS functions developed for the use cases from Section 4, the business logic typically had to be wrapped into a more or less fixed code skeleton (sometimes with a certain degree of flexibility). For example, function handlers need to have a specific name and structure, a specific interface must be implemented, or a function's signature must conform to a certain format. By introducing such wrapping around the actual business logic, functions become dependent on provider's requirements. Moreover, functions are packaged and deployed differently, e.g., AWS allows having multiple functions in one package whereas Azure allows only one function per package.

Such problems can also be encountered for non-FaaS components. For instance, function orchestration workflows from the use case described in Section 4.4 are defined and triggered differently,

e.g., AWS Step Functions allows specifying a workflow using a custom DSL and does not need an orchestrating function for running the workflow, whereas Azure Durable Functions allows defining workflows in code and need an orchestrating function to run them. As a result, specification and packaging dependencies contribute to the degree of lock-in into vendor's requirements.

Example: Format and data types dependencies. One interesting example of a data type dependency we encountered in all use cases from Section 4 is also related to FaaS-hosted functions. On the conceptual level, event types always look similar, e.g., database insert event, which is definitely not the case for real implementations. Every provider has a specific way of passing events to functions, also, depending on if the chosen programming language is strongly-typed. For example, AWS provides a separate library for working with service-specific events, e.g., AWS S3 events. As a consequence, the function code becomes coupled with specific data types, which in most cases are not compatible with other providers. Interestingly, event if the language is weakly-typed, e.g., JavaScript, the event handling code is still coupled with the format required by the provider. Since events in JavaScript are objects consisting of properties and their values, the event processing logic must still access property names defined by providers, which inevitably couples the function with provider's formats and data types similar to an example shown in Listing 2.

Listing 2. Excerpts from the 'ingest' function in the event processing use case that store an event as a deserialized JSON object in the 'message' variable.

```
// AWS JavaScript function definition
module.exports.handleIngest = async (event) => {
   let message = JSON.parse(event.body);
   // code continues
}

// Azure Functions JavaScript function definition
module.exports.handler = async function (context, req) {
   let message = req.body;
   // code continues
}

// IBM Cloud Functions JavaScript function definition
exports.main = async function(args) {
   let message = args;
   // code continues
}
```

Example: Library dependencies. Obviously, for cases when the component's code must use custom data types, the library dependency is introduced. Additionally, usage of specific libraries might result in a so-called version lock-in [17]. We encountered a version incompatibility issue when implementing in C# for Azure Functions for the serverless API and function orchestration use cases described in Sections 4.2 and 4.4. There, the triggers were not identified correctly because storage account library version was incompatible with the core library, which was resolved by reverting it to an earlier version. Library dependencies are introduced not only for processing custom data types, but also, e.g., for interacting with provider-specific APIs. Another problem happened when migrating the serverless API use case described in Section 4.2 to IBM Cloud. While Go executables mostly do not rely on dynamically linked libraries, for network communication the dynamically-linked implementation of the *net* package is used by default. Some minimalistic Linux images used for function containers might lack these dependencies, eventually preventing executables to run, which happened in our case too. This library dependency problem was resolved by disabling a CGO package which allows using C code within Go before running the build.

Example: Service interaction dependencies. Another interesting dependency is related to the provider-specific service interaction requirements. One good example we encountered is the object storage interaction, which required different actions across providers in the thumbnail generation use case described in Section 4.1. More specifically, the bucket triggers a function whenever an image is uploaded and the function must generate a thumbnail from this image. The caveat here is how to access the image from code, e.g., AWS events provide a reference to the corresponding bucket and image, and the function's code needs to include the image retrieval logic. Conversely, when Azure Object Storage's event triggers a function, the image can be directly accessed from the event data, which changes the way function's code has to be implemented. While this does not introduce big problems for this simple use case, such dependencies might become a problem for more complex scenarios.

Possible solutions and dead ends: Depending on the structure of a given application, tackling problems introduced by locking into implementation requirements might become a serious barrier. While aforementioned dependency types are quite heterogeneous, many of them can be relaxed by employing a stronger separation of concerns, i.e., by modularizing the actual business logic into separate libraries and writing the provider-specific code around them. We used this approach in several use cases, and in particular for serverless API and function orchestration applications described in Sections 4.2 and 4.4. The business logic was packed into a vendor-agnostic library which was used in vendor-specific code that handles inputs and outputs as well as interacts with remote services. As a consequence, migrating such modularized functions was significantly easier.

In theory, it is also possible to smoothen the event format discrepancies by using a vendor-agnostic event formats such as CloudEvents, and performing vendor-specific format transformation outside of the vendor-agnostic business logic package. Modularization, however, does not help significantly with overcoming service interaction and library dependencies, since the provider-specific code will still need to be written. One possible direction to simplify such problems is to facilitate boilerplate code generation for various cases and combinations, similar to how standard Maven archetypes are introduced by providers for FaaS development.

6.1.3 ***Configuration Requirements Lock-in.*** Similar to implementation requirements, a very crucial part is the way services are configured as parts of the application. This dimension includes both service- and application-level requirements.

Problem: *Fulfillment of provider's configuration requirements introduces tight-coupling which affects application's portability.*

Examples: A good illustration of a FaaS-specific configuration encountered for all use cases (see Section 4) is the definition of triggers that link together provider-specific services and functions. In AWS

Lambda, the triggers are defined externally in several possible ways including web UI, or establishing these binding by means of a deployment model defined using AWS Cloud Formation's DSL or AWS SAM template.

Listing 3. YAML Trigger definition for AWS Lambda in Serverless framework

```
functions:
  upload:
    handler: <PACKAGE_PATH>
    events:
      - http:
          path: upload
          method: post
```

In contrast to AWS Lambda's approach, Azure Functions platform allows configuring function triggers directly in the function's code, e.g., by means of Java annotations as shown in Listings 3 and 4. Such configuration binding implies certain code modifications and have to be taken care of when migrating functions. Application-level configuration examples include security-related configurations, naming conventions, e.g., AWS S3 bucket names must be globally-unique, while Azure Object Storage names must be unique only within the scope of a storage account.

Listing 4. Trigger definition for Azure Functions in Java using annotations

```
@FunctionName("Upload-Image")
@StorageAccount(Config.STORAGE_ACCOUNT_NAME)
public HttpResponseMessage upload(
   @HttpTrigger(name = "req", methods = {HttpMethod.POST},
        authLevel = AuthorizationLevel.ANONYMOUS)
        HttpRequestMessage<String> request,
   @BindingName("name") String fileName,
   @BlobOutput(name = "out", path = "input/{name}")
        OutputBinding<byte[]> blobOutput) {
     // code continues
}
```

Possible solutions and dead ends: To some extent, this problem and possible solutions are similar to implementation requirements, i.e., using modularization where possible, trying to use provider-agnostic formats, e.g., for deployment modeling, etc. However, in the majority of the cases it is not possible to avoid locking into provider-specific configurations as it is the only way to start using desired services. One possible direction here is to automate the process of mapping configurations. Obviously, an increase in number of standardization initiatives related to various aspects of implementation and configuration requirements and their successful adoption could be a step towards more portable serverless applications. CloudEvents, a common events format endorsed by CNCF, is one example of such initiatives. Other examples could be standardized trigger definitions, function descriptors, usage of provider-agnostic deployment modeling languages together with the third-party middleware, etc.

6.1.4 ***Service Limitations Lock-in.*** Often, it is the case to couple application's components implementation with existing limitations of the chosen offering, e.g., adapting the business logic to fit into allowed execution time or the impossibility to assign a desired amount of computer resources.

Problem: *When using a specific service offering, applications become dependent on service-specific limitations, e.g., resource consumption limits or allowed data transfer rate.*
Examples: For example, AWS Step Functions and IBM Composer set a maximum allowed limit for function input size, whereas Azure Durable Functions do not have any limits. This difference might require implementing functions differently, e.g., in the function orchestration use case described in Section 4.4 we had to implement inputs/outputs caching for orchestrators that impose such limits, whereas implementing the caching mechanism for providers without input-output limits is not required. Other examples of such limitations might be different limits imposed by providers on function execution time or compute resources.
Possible solutions and dead ends: Locking into service-specific limitations such as particular execution time limits might also be a dead end, since the migrated component will need to be reimplemented in order to comply with new requirements. Here, the feasibility of actions is determined by the concrete use case. For example, if a FaaS-hosted function is developed according to the common guidelines to be stateless and short-lived, then execution time limits are not supposed to affect its portability. Conversely, if the function strongly depends on a particular setting of a platform, it will likely require modifications for hosting it on another FaaS platform, or even has to be reimplemented. In some cases, reimplementing architecture's components using different architectural style can help solving the migration issues. However, the decision whether such solution is acceptable or not, strongly relies on multiple aspects including project's requirements, costs, etc.

6.2 Tooling Compatibility

The compatibility of the application-related tooling is another important aspect of serverless applications' migration.
Problem: *Locking into specific tooling, e.g., which facilitates the development, deployment, or observability of an application, might complicate its portability or even make it not directly possible.*
Examples: For example, a common way is to automate the deployment of applications using deployment modeling and orchestration technologies such as AWS Cloud Formation, Terraform, or Serverless framework. Such technologies can be either provider-specific like AWS Cloud Formation, or support multiple target providers such as Serverless, which becomes increasingly popular for describing the deployment and configuration of serverless applications. However, while supporting multiple providers, the deployment models defined using Serverless are not provider-agnostic, due to the description of provider-specific services, event types, etc, as was encountered when using Serverless framework for the use cases described in Section 4. As a result, these models have to be specified separately for target providers. Similar problems might occur when the development process also relies on provider-specific monitoring and testing solutions, custom CI/CD pipelines that will need to be modified or completely replaced with some compatible alternatives.
Possible solutions and dead ends: Unfortunately, for this problem the choice of possible solutions depends on the actual tool or a set of tools that are used. In case of deployment automation, provider-agnosticism is a well-known problem [23], and one of the possible solutions is to use vendor-agnostic cloud modeling

languages such as TOSCA [19]. However, the major risk is still to introduce provider-specific parts in these deployment models. For example, Serverless framework also uses a custom DSL to specify deployment models, but essentially these models are tightly-coupled with specific providers and cannot be reused directly for migration use cases as they rely on custom events and services. One possible way to relax the lock-in is to use modularization of deployment models where possible, e.g., by using inheritance and include statements if the language supports such constructs, as well as providing means to generate and transform boilerplate parts of the models. Similar directions can be followed for other tools, e.g., switching to provider-agnostic monitoring solutions if possible, but in many cases such change might require changing multiple components, e.g., adding new or integrating existing log emitting side cars for monitoring with custom, provider-agnostic solutions.

6.3 Architecture Compatibility

Surprisingly, this problem is rather about maintaining and not tackling one type of the lock-in, called architecture lock-in [17]. The main question here is whether the migrated application needs to be compatible with the original architectural style, i.e. remain serverless after the migration.

Problem: *The architectural style of a migrated application changed after the migration is performed, making the application potentially incompatible with the originally used architectural style.*

Examples: In case of the service incompatibility problem we encountered with the event processing use case described in Section 4.3, after substituting the point-to-point channel with a pub-sub channel we eventually obtained a more or less compatible architecture, yet with a completely changed semantics for one of the components. For migrating serverless applications, the discrepancy between source and target components, or their incompatibility might cause the switch from serverless to *serverful* architecture where provider-managed components were eventually substituted, e.g., with manually integrated PaaS or IaaS-hosted components.

Possible solutions and dead ends: Essentially, deciding if the change of architectural style after migration is acceptable or not is related to organization and project requirements as well as how efficient the end solution is. For example, if the resulting solution requires, e.g., implementing and integrating new triggers or reimplementing entire components using IaaS, it becomes a dead end.

7 RELATED WORK

To the best of our knowledge, there are no existing publications trying to explore and analyze the lock-in aspects encountered when migrating serverless, FaaS-based applications.

Multiple publications focus on portability and interoperability aspects of cloud applications and vendor lock-in problems. Silva et al. [28] conduct a systematic study that analyzes and classifies cloud lock-in solutions existing in research literature. Opara-Martins et al. [25] discuss the issues associated with portability and interoperability, focusing on the vendor lock-in problem. Authors describe various types of challenges also including brief description of such technical challenges as integration and data portability. However, the overall discussion stays on the high-level and does not contain the details about serverless and FaaS. Lipton [19] discusses how TOSCA, a provider-agnostic cloud modeling language [7] standardized by OASIS, can help avoiding locking into specific cloud providers. Miranda et al. [22] propose an approach using software adaptation techniques to tackle the vendor lock-in problem. Authors describe various types of application's component matching, e.g., component-to-component mismatch where migration of components to another cloud provider introduces a need to adapt the communication between them. While being related, the paper discusses component mismatch types on the high-level with the focus on traditional cloud service models, and without investigating the specifics of serverless and FaaS. Hohpe [17] discusses various types of lock-in, also including the vendor lock-in problem. In this article, the lock-in is described as a problem that is not always beneficial to tackle, since successfully tackling one lock-in type might introduce another lock-in. The described classification of lock-ins covers various aspects, from vendor and architecture lock-in to more organizational lock-in types such as legal or skills lock-in. In our work we refer to some of the lock-in types described in this article, e.g., architecture and version lock-ins, however, we mostly focus on various dimensions of vendor lock-in in the context of serverless and FaaS-based applications and connections between other lock-in types with the problems we encountered in our experiment.

Cloud migration is a well-established topic addressed by multiple researchers. Andrikopoulos et al. [2–4] elaborate on the challenges of migrating applications to the cloud and discuss the aspects of decision support for migrating existing applications. Multiple dimensions of requirements including elasticity, multi-tenancy, costs, security and data confidentiality, quality of service, are analyzed to form a holistic migration decision support framework. Binz et al. [9] introduce the framework for migrating applications to the cloud and across cloud providers based on search for suitable hosting alternatives for a given application model. Strauch et al. [29] present a provider- and technology-agnostic multi-step methodology for migrating application's database layer to the cloud. Multiple described migration challenges remain relevant when porting serverless applications across providers, e.g., data exchange format differences, security and cost aspects. However, the discussed approaches focus more on the legacy-to-cloud migration and do not elaborate on lock-in aspects in the domain of serverless computing. In this work, we focus solely on analyzing and structuring lock-in aspects of cloud-native, serverless applications, without considering other dimensions. The existing work on cloud migration can serve as a baseline for facilitating the decision support on portability of serverless applications. In addition, several discussed challenges become more restricted in the context of serverless applications influencing the overall portability strategy, e.g., elasticity w.r.t. provider-managed components or choice of the cloud service model for application components.

8 CONCLUSION AND FUTURE WORK

In this paper, we designed and conducted an experiment in which we implement and manually migrate four common serverless, FaaS-based application use cases across three well-known commercial cloud providers. We intentionally use simplified applications to demonstrate that even in such scenarios, multiple lock-in problems are encountered. We analyze several categories of encountered

lock-in problems and outline possible solutions, which might help relaxing the degrees of lock-in as well as discuss when these solutions become dead ends.

Essentially, the nature of serverless computing, and FaaS, amplify the lock-in problems since all components are intended to be managed by providers, resulting in multiple dimensions of a vendor lock-in problem. We can highlight the following findings:

- portability of components is affected by multiple lock-in types, e.g., feature set lock-in, implementation and configuration requirements lock-ins, or service limitations lock-in
- implementation requirements lock-in is introduced due to multiple dependency types, including implementation and packaging guidelines, format and data type dependencies, library dependencies, etc.
- some lock-in types can be relaxed by modularizing the business logic and parts of configurations where possible, using provider-neutral formats, etc.
- lack of compatible component alternatives might become a migration dead end
- the decision to comply with the serverless paradigm might require maintaining the architecture lock-in, meaning that reimplementation of components as non-serverless forces the paradigm shift
- the lock-in into tooling might also affect serverless application's portability

The encountered problems are amplified even more when the migration scenarios involve on-premise hosting of components as a target. For instance, FaaS offerings from commercial providers are so strong not only because of FaaS platforms themselves, but also because of the reach network of services with built-in integration. In contrast, self-hosted solutions require not only identifying the suitable alternative for FaaS platforms and services, but also in most of the cases spending a lot of efforts on integrating them together. In future work, we plan to investigate the decision support process on whether the portability of a given serverless, FaaS-based application is possible and how to automate its migration process.

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 reviewers, whose insightful feedback helped us to improve this paper.

REFERENCES

[1] Amazon Web Services, Inc. 2019. AWS Lambda. https://aws.amazon.com/lambda
[2] Vasilios Andrikopoulos, Tobias Binz, Frank Leymann, and Steve Strauch. 2013. How to Adapt Applications for the Cloud Environment. *Computing* 95 (2013), 493–535. https://doi.org/10.1007/s00607-012-0248-2
[3] Vasilios Andrikopoulos, Alexander Darsow, Dimka Karastoyanova, and Frank Leymann. 2014. CloudDSF–the cloud decision support framework for application migration. In *European Conference on Service-Oriented and Cloud Computing*. Springer, 1–16.
[4] Vasilios Andrikopoulos, Zhe Song, and Frank Leymann. 2013. 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*. IEEE Computer Society, 565–572. https://doi.org/10.1109/CLOUD.2013.128
[5] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. 2017. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer, 1–20.
[6] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. 2017. The Serverless Trilemma: Function Composition for Serverless Computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017)*. ACM, 89–103.
[7] Alexander Bergmayr, Uwe Breitenbücher, Nicolas Ferry, Alessandro Rossini, Arnor Solberg, Manuel Wimmer, Gerti Kappel, and Frank Leymann. 2018. A Systematic Review of Cloud Modeling Languages. *ACM Comput. Surv.* 51, 1, Article 22 (Feb. 2018), 38 pages. https://doi.org/10.1145/3150227
[8] Alexandre Beslic, Reda Bendraou, Julien Sopenal, and Jean-Yves Rigolet. 2013. Towards a solution avoiding Vendor Lock-in to enable Migration Between Cloud Platforms. In *MDHPCL@ MoDELS*. Citeseer, 5–14.
[9] Tobias Binz, Frank Leymann, and David Schumm. 2011. CMotion: A Framework for Migration of Applications into and between Clouds. In *2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE, 1–4.
[10] Jean Bozman and Gary Chen. 2010. Cloud computing: The need for portability and interoperability. *IDC Executive Insights* (2010).
[11] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 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* (2019).
[12] Cloud Native Computing Foundation (CNCF). 2018. CNCF Serverless Whitepaper v1.0. https://github.com/cncf/wg-serverless/tree/master/whitepapers/serverless-overview
[13] Geoffrey C Fox, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2017. Status of serverless computing and function-as-a-service (faas) in industry and research. *arXiv preprint arXiv:1708.08028* (2017).
[14] P. García López, M. Sánchez-Artigas, G. París, D. Barcelona Pons, Á. Ruiz Ollobarren, and D. Arroyo Pinto. 2018. Comparison of FaaS Orchestration Systems. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. 148–153. https://doi.org/10.1109/UCC-Companion.2018.00049
[15] Shane M Greenstein. 1997. Lock-in and the costs of switching mainframe computer vendors: What do buyers see? *Industrial and Corporate Change* 6, 2 (1997), 247–273.
[16] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651* (2018).
[17] Gregor Hohpe. 2019. Don't get locked up into avoiding lock-in. https://martinfowler.com/articles/oss-lockin.html
[18] Yury Izrailevsky and Charlie Bell. 2018. Cloud Reliability. *IEEE Cloud Computing* 5, 3 (2018), 39–44.
[19] Paul Lipton. 2012. Escaping vendor lock-in with tosca, an emerging cloud standard for portability. *CA Labs Research* 49 (2012).
[20] Peter M. Mell and Timothy Grance. 2011. *SP 800-145. The NIST Definition of Cloud Computing*. Technical Report. Gaithersburg, MD, United States.
[21] Microsoft. 2019. Microsoft Azure Durable Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-concepts
[22] Javier Miranda, Juan Manuel Murillo, Joaquín Guillén, and Carlos Canal. 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*. ACM, 12–19.
[23] OASIS. 2015. *TOSCA Simple Profile in YAML Version 1.0.* Organization for the Advancement of Structured Information Standards (OASIS).
[24] OMG. 2011. *Business Process Model and Notation (BPMN) Version 2.0.* Object Management Group (OMG).
[25] Justice Opara-Martins, Reza Sahandi, and Feng Tian. 2014. Critical review of vendor lock-in and its impact on adoption of cloud computing. In *International Conference on Information Society (i-Society 2014)*. IEEE, 92–97.
[26] Dana Petcu. 2011. Portability and interoperability between clouds: challenges and case study. In *European Conference on a Service-Based Internet*. Springer, 62–74.
[27] Benjamin Satzger, Waldemar Hummer, Christian Inzinger, Philipp Leitner, and Schahram Dustdar. 2013. Winds of change: From vendor lock-in to the meta cloud. *IEEE internet computing* 17, 1 (2013), 69–73.
[28] Gabriel Costa Silva, Louis M Rose, and Radu Calinescu. 2013. A systematic review of cloud lock-in solutions. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, Vol. 2. IEEE, 363–368.
[29] Steve Strauch, Vasilios Andrikopoulos, Dimka Karastoyanova, and Karolina Vukojevic-Haupt. 2015. Migrating eScience applications to the cloud: methodology and evaluation. *Cloud Computing with e-Science Applications* (2015), 89–114.
[30] Kostas Stravoskoufos, Alexandros Preventis, Stelios Sotiriadis, and Euripides GM Petrakis. 2014. A Survey on Approaches for Interoperability and Portability of Cloud Computing Services. In *CLOSER*. 112–117.