Cloud Function Lifecycle Considerations for Portability in Function as a Service

Robin Hartauer, Johannes Manner[a] and Guido Wirtz[b]
Distributed Systems Group, University of Bamberg, Germany
robin-christoph.hartauer@stud.uni-bamberg.de,

Keywords: Function as a Service, Serverless Computing, Portability, Function Lifecycle.

Abstract: Portability is an important property to assess the quality of software. In cloud environments, where functions and other services are hosted by providers with proprietary interfaces, vendor lock-in is a typical problem. In this paper, we investigated the portability situation for public Function as a Service (FaaS) offerings based on six metrics. We designed our research to address the software lifecycle of a cloud function during implementation, packaging and deployment. For a small use case, we derived a portability-optimized implementation. Via this empirical investigation and a prototypical migration from AWS Lambda to Azure Function and from AWS Lambda to Google Cloud Function respectively, we were able to reduce writing source code in the latter case by a factor of 17 measured on a Lines of Code (LOC) basis. We found that the default *zip* packaging option is still the favored approach at Function as a Service (FaaS) platforms. For deploying our functions to the cloud service provider, we used Infrastructure as Code (IaC) tools. For cloud function only deployments the Serverless Framework is the best option whereas Terraform supports developers for mixed deployments where cloud functions and dependent services like databases are deployed at once.

1 INTRODUCTION

In 1976 already, BOEHM and others stated that portability of an application is one of eleven properties for describing the quality of software (Boehm et al., 1976). An ISO/IEC/IEEE standard for *systems and software engineering vocabulary* defined portability as "the ease with which a system or component can be transferred from one hardware or software environment [...] with little or no modification" (ISO/IEC/IEEE, 2010, p. 261). This notion of portability is the same for applications hosted in the cloud. It is especially important that users are able to switch between vendors based on their quality of services and their performance (Gonidis et al., 2013).

Efforts to enable portability are already present in the community. One example are cloud function triggers. These triggers are events from different sources like databases. When a new entry is created, an event is created which triggers the execution of the cloud function. The structure of these events are proprietary and specified by the service provider, in our case a public cloud service provider. CloudEvents[1], a Cloud Native Computing Foundation (CNCF) incubator project, tries to solve the interoperability[2] issue. This standard tries to build a foundation for communication across vendors when invoking cloud functions. We will not address the interoperability issue in this work since we agree with KOLB (Kolb, 2019) that interoperability and portability are no synonyms despite their interchangeable usage in research and industry. FISCHER and others (Fischer et al., 2013) support this assessment. They see interoperability from a communication perspective compared to a deployment perspective on portability.

Since portability for Platform as a Service (PaaS) has already been investigated in detail (Kolb, 2019), we want to suggest first steps towards portability in FaaS as well as mitigation strategies to avoid a vendor lock-in. However, in practice, a single function is rarely used on its own. Dependent services like

[a] https://orcid.org/0000-0002-7298-3574
[b] https://orcid.org/0000-0002-0438-8482

[1] https://cloudevents.io/

[2] "the capability to communicate, execute programs, and transfer data among various functional units in a manner that requires the user to have little or no knowledge of the unique characteristics of those units" (ISO/IEC/IEEE, 2010, p. 186)

Hartauer, R., Manner, J. and Wirtz, G.
Cloud Function Lifecycle Considerations for Portability in Function as a Service.
DOI: 10.5220/0010999000003200
In *Proceedings of the 12th International Conference on Cloud Computing and Services Science (CLOSER 2022)*, pages 133-140
ISBN: 978-989-758-570-8; ISSN: 2184-5042

databases, gateways, messaging systems are needed to build cloud applications as well. These services increase the risk for an ecosystem lock-in at the selected vendor. This was already identified as a risk for building cloud (Armbrust et al., 2010) and especially *serverless* applications (Baldini et al., 2017) by having the fast changing "API jungle" (van Eyk et al., 2018) in mind.

Additionally we want to extend our scope in order to consider on the software lifecycle rather than the development of a single function only. Initial development, evolution, service deployment and operation, closedown and phaseout are the five steps of the software lifecycle as defined by RAJLICH and BENNETT (Rajlich and Bennett, 2000). Since portability considerations only influence the first three phases, we will exclude the latter two. We combine the first two phases in implementation aspects and split servicing into packaging and deployment. Each of our research questions is related to one of these three steps:

- **RQ1:** (1) How can the portability of a serverless application be assessed based on code properties and metrics? (2) How can the challenges of portability be solved when considering dependencies on other services like in-function database calls?
- **RQ2:** (1) Which packaging options exist for cloud function provisioning? (2) Does the chosen packaging option increase or decrease portability for a cloud function?
- **RQ3:** (1) Which deployment options exist for cloud functions? (2) Does the chosen deployment option increase or decrease portability for a cloud function? (3) How can Infrastructure as Code solutions support developers to provide cloud functions with and without dependent services on different platforms?

To answer our research questions, we state related work in Section 2. For deployment, we chose the three public cloud providers Amazon Web Services (AWS), Azure Cloud (AZC) and Google Cloud Platform (GCP). As a baseline, we use AWS and migrate our functions and the dependent services to AZC and GCP as suggested by (Yussupov et al., 2019). Unlike their approach, we already had the migration in mind and looked at potential portability weaknesses during experiment design. We discuss our experimental setup in Section 3, where we list the technologies and tools used. In Sections 4, 5 and 6, we discuss the research questions in order. At the beginning of each of these Sections, we describe the situation and the challenges we faced. We then propose different implementation solutions and define dimensions on which to compare the solutions. In Section 7, we discuss our results and point out some threats to validity introduced by our methodology and experiment. Future work concludes the paper.

2 RELATED WORK

Some authors identified the need to have a uniform interface to finally solve portability issues in FaaS (Jonas et al., 2019). They argue that an abstraction introduced by Knative[3] via containerization and Kubernetes could be a solution which is also pushed by Google's cloud platform. Another interesting work was done by (Yussupov et al., 2019) who implemented four typical FaaS use cases on AWS. Then they migrated these applications to AZC and IBM cloud where they faced different types of lock-ins. To categorize these lock-ins, they used a scheme according to HOPHE[4]. The assessment was based on a binary (yes/no) decision to determine whether some aspects need to be changed. These include, for example, the programming language if the target platform of the migration does not offer the initial language.

Furthermore, they included a mapping of typical application components at different providers which helps to find the corresponding service at the target platform. This experience report on the challenges of an "unplanned migration" (Yussupov et al., 2019) is an interesting contribution, but lacks details like the Lines of Code (LOC) metric, and the comparison between the providers. In addition, YUSSUPOV and others (Yussupov et al., 2020) asked the question whether and to which extent an application is portable to another platform. To investigate this, they transformed a serverless application to a provider agnostic model using established frameworks like the Cloud-Application-Management Framework (CAMF) (Antoniades et al., 2015) or Topology and Orchestration Specification for Cloud Applications (TOSCA) (Binz et al., 2013).

In edge use cases, portability is even more important since the hardware used is more heterogeneous. Early works deal with these issues by suggesting a FaaS platform based on WebAssembly[5] (Hall and Ramachandran, 2019) or using Linux containers, i.e. cgroups and namespaces capabilities to allow for a common interface for edge and cloud elements to enable also shifting of components (Wolski et al., 2019).

[3]https://knative.dev/docs/

[4]https://martinfowler.com/articles/oss-lockin.html

[5]https://webassembly.org/

3 EXPERIMENT SETUP

To demonstrate the challenges of migrating source code of a cloud function from one platform to another, we created a small prototype[6]. We implemented a function of a typical user management service consisting of two parts: First, the function can create users and store them in a database. Secondly, it can retrieve the user by checking the credentials inserted via a login. We keep our function simple since the focus of this work lies on looking at interfaces and the lifecycle, but nevertheless this single, simple function already reveals the most important challenges. During implementation we focused on the portability challenges described above. For a better comparison of the lifecycle tasks, we investigated each task in isolation. We developed the implementations for each platform in parallel and used the gained knowledge to understand similarities and differences. This enabled us to quantify code changes and find best practices for every task. For the implementation we used NodeJS 14[7], Typescript[8], ExpressJs[9], the Serverless Framework[10] and Terraform[11].

4 CLOUD FUNCTION IMPLEMENTATION

4.1 Challenges, Properties and Metrics

During the implementation phase of our function, we found several aspects to consider when tackling portability problems. We later relate these properties, see Table 1, to metrics, see Table 2, and give an assessment of the portability. The first property (P1) is the handler implementation, which is different for each platform and language. Listing 1 shows examples for JavaScript handler interfaces for the three providers. There, we see a difference in the order of parameters and the data they contain. Google Cloud Functions (GCF) uses the ExpressJS abstraction. AWS Lambda and Azure Functions (AZF) have custom formats for their handler abstraction. Additionally, the return values of the different platforms are handled differently. At AWS, we have to configure an additional gateway (Amazon API Gateway) to handle the incoming and outgoing traffic. This gateway also has the option to add custom transformation rules, handle security related features etc. We are aware that, as YUSSUPOV and others (Yussupov et al., 2019) already stated, such a use of proprietary features hinders the migration. Therefore, we did not use these additional request and response capabilities at AWS.

Comparing the logging capabilities shows that the providers use different logging frameworks. This is the second property we want to consider (P2). AWS Lambda and GCF use the JavaScript default `console.log()` statement which writes the results to the corresponding monitoring/logging service like AWS CloudWatch. The logging data is stored on a function and request basis. The same behavior can be seen on Azure, when using `context.log()`.

Listing 1: Handler Interfaces for included Providers in the Experiment.

```
// GCF handler
exports.helloHttp = (req, res) => {
  console.log('Hello CLOSER');
  res.send('Hello CLOSER');
};

// AWS Lambda handler
exports.handler =
 async function(event, context) {
  console.log('Hello CLOSER');
  return context.logStreamName;
};

// AZF handler
module.exports =
 async function (context, req) {
  context.log('Hello CLOSER');
  return { body: "Hello CLOSER"};
}
```

The third and last property (P3) is the access to other services in the provider's ecosystem. For every investigated platform, custom SDKs and APIs complicate portability. The three properties and their manifestations at the three providers are summarized in Table 1.

Table 1: Implementation Aspect assessed at investigated Providers.

	AWS	AZF	GCF
P1	custom format		ExpressJs
P2	console.log	context.log	console.log
P3	native SDKs and APIs		

When migrating the function from one provider to another, the number of source code changes are one metric (`M1`) to consider as suggested by (Lenhard and

[6] https://gitlab.com/rrobin/masterarbeit_hartauer
[7] https://nodejs.org/en/
[8] https://www.typescriptlang.org/
[9] https://expressjs.com/
[10] https://www.serverless.com/
[11] https://www.terraform.io/

Table 2: Metrics for a Portability Assessment of the Cloud Function Lifecycle at different FaaS Providers.

	Metric Description	Section
M1	Source code changes based on LOC metric	4.3
M2	Number of different locations where source code has to be changed	4.3
M3	Number of steps in the packaging process	5
M4	Portability assessment of the deployment configuration	6.2
M5	Number of platform dependent configuration parameters	6.2
M6	Portability assessment of the deployment process and involved tooling	6.2

Wirtz, 2013). We measure `M1` based on the LOC metric which is easily quantifiable. In contrast, soft facts like experience of developers, tool support etc. are hard to quantify. Therefore, we include only the LOC metric for an unbiased comparison. The number of different locations where code needs to be altered is another metric (`M2`) for assessing how error-prone the migration might be.

The properties (P1-P3) and the metrics (`M1` and `M2`) give an answer to question **RQ1.1**. In the following, we propose implementation improvements for the three properties, which also positively influence the metrics `M1` and `M2` as shown in the evaluation at the end of this Section.

4.2 Prototypical Implementation

As stated before, we first implemented our cloud function on AWS Lambda with portability in mind as a baseline for our experiment. To harmonize the different function handler interfaces (P1), we adjusted the AWS Lambda and AZF handler interface and added another layer to conform to the request and response handling based on the ExpressJS[12] framework as already done by GCF. We used a transformation package to transform the incoming request to meet the request/response interface. After this transformation we were able to use the generic ExpressJS request handling. This enables a separation of business logic and provider dependent logic through our middleware layer. In addition, it allows us to test the business functionality independently from the provider since we can start the standalone ExpressJS application.

To handle the different logging-mechanisms (P2), we used a logging interceptor. Because of this, every `console.log()` statement will be translated to the corresponding provider statement. This interceptor was only used for AZF since the other platforms already used the JavaScript standard.

To solve the last problem (P3) with different vendor-specific services - in our case databases - we used the *Factory-Pattern* (Ellis et al., 2007). This pattern returns a provider specific object for database access when a user requests it for the corresponding ecosystem. This improves our metric `M2` since the internal interface used by the business logic is not affected by the migration and therefore changes are centralized. Only the factory method has to be extended to work with the new database service when migrating to a new platform. For the database services, we used Amazon DynamoDb, Azure Cosmos Db and Google Firestore since they are all document oriented databases. The corresponding database interface implementation is selected based on an environment variable at runtime. These implementation optimizations answer **RQ1.2**.

[12]https://expressjs.com/

4.3 Implementation Evaluation

To evaluate the improvements suggested in the previous Section, we used the LOC metric. We evaluate `M1` in Table 3 by looking at the LOC measure for the migration from AWS to AZC as well as from AWS to GCP.

Table 3: Source Code Changes by Migrating the Function of our Use Case.

	Unoptimized		Optimized	
AWS-AZC	+86	-71	+80	-9
AWS-GCP	+106	-72	+62	-10

In the *unoptimized* case none of the aforementioned improvements (handler wrapper, log interceptor, database factory) are implemented. Both unoptimized migrations show a lot of code changes, added as well as removed LOC. This high number of changes is due to platform dependent code. For the *optimized* case, we see a lot of LOC additions in since it features all the changes described in Section 4.2. When we assume, that these middleware components are in place already, the code additions for AWS to AZC would be only +20/-6 LOC and +6/-8 for the migration of AWS to GCP respectively.

`M2` is hard to assess quantitatively for the unoptimized case since some platform specific features cannot be easily compared with another. The need to have resource groups on AZC or releasing the API

on GCP showcases this problem. Furthermore, using native database accesses, which are spread in the code, lead to a situation where we have to adjust every database related call during migration. It is important to reduce the number of code location changes as introduced by our improvements and shown due to the results in Table 3. When assessing M2 in the optimized case, we only have three code locations to change namely the abstract factory, the handler harmonization and the new implementation for the corresponding database service. The fewer locations to change, the less error-prone is the migration.

The presented results might be slightly different, when starting with GCF as baseline. However, this does not affect our findings about portability of functions using our approach.

5 CLOUD FUNCTION PACKAGING

After the implementation of our cloud function, the next step in the lifecycle is packaging. In our second research question (**RQ2.1**), we ask: *Which options are offered for packaging an application?* The FaaS providers started their offerings by accepting zip archives with the source code and the dependencies needed to run the function.

One packaging alternative is a OCI[13] compliant image. AWS Lambda and AZF provide this option, whereas GCF is currently not capable to run functions based on a custom image. This constrains the portability of a containerized solution to GCF. However, GCP offers another service called *Google Cloud Run (GCR)* where a user can run arbitrary images in a *serverless* fashion. For comparison, we use this equivalent offering for discussing the packaging option for our function at GCP.

Containerization is popular especially due to Docker and frameworks like Kubernetes. The claim "Build, Ship, Run, Any App Anywhere"[14] and its practical implications are the reason for the wide adoption of this technology. Since containers allow an abstraction from the platform and operating system they are running on, their declarative description (in the Docker universe the Dockerfiles) faces the same problems as function handlers. Due to the design of the platforms, e.g. when using optimized virtual machine monitors like Firecracker[15], the platform providers restrict the number of potential base images and publish a set of valid ones. The base image problem at AWS Lambda and AZF is different to the offering of GCR. GCR accepts all images by running the functions on a Knative hosting as already mentioned in related work (Jonas et al., 2019).

Table 4: Assessment for M3, process portable (●) to not portable (○) and (-) for not available.

	AWS	AZF	GCF	GCR
zip	●	●	●	-
image	◐	○	-	●

To assess the packaging of a cloud function, we focus on the metric M3 which tries to quantify the number of steps in the packaging process. Table 4 shows an assessment for M3 and gives an answer to **RQ2.2** which addresses the question, if the chosen packaging option increases or decreases portability.

Zip packaging option enables an easier portability compared to the *image* option when the implementation challenges are addressed, as discussed in the previous Section. Our metric M3 is zero in this case since only a custom zip tool has to be used by the developer. There are differences for deployment, when choosing zip or image packaging, but we only consider the packaging process here.

Image packaging on the other side is not that portable. AWS Lambda has a set of base images configured which support the same runtimes available for zip packaging. For AWS Lambda, M3 is only affected by exchanging the base image to a platform compliant one. For AZF, the user additionally has to configure some environmental parameters, at least AzureWebJobsScriptRoot and whether logging is enabled or not. For this reason, we assess the portability worse compared to AWS Lambda. In the GCP case, there is no image support for GCF. The corresponding alternative can use any image for starting containers. Therefore, portability is not hindered by choosing GCR.

Due to these aspects and in order to use only FaaS offerings from the corresponding providers, we use the *zip* packaging option for deployment in the next Section.

6 CLOUD FUNCTION DEPLOYMENT

6.1 Deployment Options

Before comparing different deployment solutions, we want to describe the current suitable deployment options for our use case, see **RQ3.1**. However, this list

[13] https://opencontainers.org/

[14] https://www.docker.com/sites/default/files/Infographic_OneDocker_09.20.2016.pdf

[15] https://firecracker-microvm.github.io/

of options is not necessarily comprehensive. First of all, there are provider related deployment tools as listed in Table 5. These tools have their own syntax and semantics for deploying functions and dependent services like our database to the corresponding ecosystem. Differences in the used configuration syntax further complicate the migration from one provider to another. Currently, the Google DeploymentManager does not offer the deployment of cloud functions, but offers a database deployment. Therefore, we had to use the gcloud CLI feature and the DeploymentManager to deploy the function and the corresponding database.

Table 5: Selection of Provider Deployment Tools and agnostic Solutions.

	Platform	Config
AWS CloudFormation	AWS	JSON, YAML
Azure ResourceManager	AZC	JSON
Google DeploymentManager + gcloud CLI	GCP	Jinja, Python, Shell
Serverless Framework	independent	YAML
Terraform	independent	HCL, JSON

Besides these three tools (AWS CloudFormation, Azure ResourceManager and Google DeploymentManager), there are provider independent tools. The Serverless Framework with more than 40,000 GitHub stars is a widely used option for deploying cloud functions. It specifies an abstraction for the configuration of the functions and transforms this provider independent format for deployment in the provider dependent formats. Since it is specialized for cloud functions, dependent services like our database cannot be deployed via the Serverless Framework. Furthermore, it addresses only a single provider at a time. For multi-cloud deployments, multiple independent deployments need to be started.

In our use case scenario, the most generic Infrastructure as Code (IaC) solution is Terraform. Despite using a custom HCL syntax for describing components, the tool is capable to deploy cloud functions and dependent services as well as components to different providers within the same deployment process.

6.2 Deployment Metrics

For the deployment assessment we use some metrics introduced by (Kolb et al., 2015) for their use case of a cloud to cloud deployment. We do not state the number of LOC changes here, since the different configuration syntaxes are more or less verbose and, therefore, hardly comparable. Instead we rate the metrics on a scale from not portable (○) to portable (●) as already done in Table 4. The first metric we want to consider is whether the configuration of the deployment is easily transferable based on the used syntax (`M4`). It works on a more general level and assesses whether changes occur in the tooling and its syntax and semantics. Such changes hinder portability since another tool increases the complexity and forces the developer to learn a new syntax. The next metric, `M5`, is the number of platform dependent configuration parameters which need to be adapted from one provider to another. For example a resource setting, i.e. the memory setting, is one of the most important options to choose. At AWS Lambda, the property is called *MemorySize*, at AZF a specification is not possible due to the dynamic allocation of resources and at GCF, the property is named *memory*. The last metric (`M6`) assesses the number of different steps during deployment when looking at the difference between the source and target platform.

Table 6: Portability Assessment of Selected Tools, not portable (○) to portable (●).

	M4	M5	M6
Native IaC Tools	○	○	○
Serverless Framework	◐/●	◐	◐/●
Terraform	●	○	●

Table 6 contains the information we need to answer **RQ3.2** and gives first insights into the portability assessment of our selected IaC tools. In summary, the providers' native IaC tools lead to a vendor lock-in and a low level of portability for all metrics, which is not surprising due to the challenges identified in this work. The situation is different for the provider independent IaC tools. The first metric investigated for the Serverless Framework is medium portable, since the cloud functions can be defined in one configuration format across platforms, but for dependent services like the database, we need the provider's native IaC tool which leads to a mixed evaluation. Terraform allows full portability with regard to this metric since the HCL syntax is used for functions and dependent services.

Since the Serverless Framework's scope is on cloud function deployment at different providers, they harmonize the configuration settings where possible. For example, they use *memorySize* as a key in

their deployment YAML for AWS Lambda and GCF, whereas there is no corresponding entry for AZF. We assess the portability of the configuration parameters as medium since there are some special settings present for different platforms, which are part of the framework adapters but cannot be ported that easily to other platforms. In these situations, workarounds need to be developed which hinder migrations. Terraform applies a different concept for the platform dependent parameters. Similar settings across different providers are not harmonized and stay therefore platform dependent, e.g. the storage location for the function is named *s3_bucket* for AWS Lambda, whereas GCF uses two properties *source_archive_bucket* and *source_archive_object*. Therefore the Terraform HCL is not reusable and has to be rewritten for every migration.

For the last metric `M6` we look at starting the deployment process and the commands and tools needed. For the Serverless Framework, the situation is similar to `M4`, when dependent services are deployed as well. For a cloud function only scenario, the commands and the tools needed are identical, hence the solution is fully portable in this case. For other scenarios with provider specific services, we need the provider tools as well and, therefore, only the Serverless Framework part is reusable for starting the deployment. As already discussed, Terraform is a provider agnostic tool and the command to deploy and the steps to execute are the same for all providers.

7 CONCLUSION

7.1 Discussion of the Results

Our research on cloud function lifecycle considerations for portability in FaaS revealed, that it is important to also consider other aspects besides implementation. Since cloud functions are seldom used in isolation, dependent services have to be considered at the same time.

We first concentrated on the source code to decouple the logic and the provider specific interfaces in Section 4. We especially focused on three aspects. We investigated the *Cloud Function Handler* interfaces where we suggested the use of a transformation package to transform the incoming request to meet a generic interface, in our case study the ExpressJS-Framework interface, as well as using a generic *log interceptor*. To solve the problem with different vendor-specific database services, we used the *Factory-Pattern*. We found that we are able to reduce the code changes to a minimum as stated in Table 3 for metric `M1`. We reduced writing new code measured on a LOC basis by a factor of 17 when migrating from AWS to GCP comparing the unoptimized case to the optimized case excluding the improved middleware.

For packaging and deploying of cloud functions, to the best of our knowledge, we are the first ones to consider this in an empirical evaluation. We used the default *zip* packaging option as well as build images from our functions. Due to some limitations, e.g. GCF is not supporting containers and proprietary images, we recommend zip packaging (see Table 4 with metric `M3`).

For deployment, we used the native provider IaC framework like AWS CloudFormation and provider independent frameworks like Terraform. We already answered the research questions on the pros and cons of the deployment tools in Table 6. Only **RQ3.3** is unanswered where we ask how IaC solutions support developers to provide cloud functions with and without dependent services. The answer is two-fold. For situations where only functions should be deployed, the Serverless Framework is favoured whereas for mixed deployments with other services from the same or a different provider's ecosystem, Terraform offers the better features. It is capable of combining the packaging and the deployment step. Due to its holistic approach, a consistent deployment and undeployment process can be guaranteed. The only downside when using platform independent tools is that improvements might not be available directly for independent services compared to native solutions.

7.2 Threats to Validity

The introduced properties and metrics used for our comparison are not complete in a sense that all facets of portability issues are addressed already. Nevertheless, since the body of knowledge in this area is small, only a few publications in the FaaS area address portability at all (Yussupov et al., 2019; Yussupov et al., 2020), this work empirically contributes to it. We are aware of some threats to validity which might compromise the results when reproducing the research:
Selected Providers for Migration - We only used a subset of public cloud provider offerings available. The LOC metrics will be different when using other providers or starting with another provider like AZF or GCF. The challenges remain the same but additional challenges might arise. Furthermore, including open source offerings using a Kubernetes abstraction might improve some drawbacks identified like the handler interface.

Selected Programming Language - In our experiment, we only used JavaScript for implementing our function. The main reason was that JavaScript is well supported for all providers used and does not introduce inconsistency in the language dimension. Therefore, as for the provider case, exact measures might differ when repeating the experiment with another language but the challenges like the native SDKs and handler interfaces are the same.

8 FUTURE WORK

Our ideas for future work are threefold. First, the identified challenges are similar to already known integration problems, where different data formats and interfaces need to be harmonized. Interesting ideas are described in an abstract way by HOHPE and WOOLF (Hohpe and Woolf, 2003). When using these abstract building blocks, the enterprise integration patterns, migrations would be a lot easier. Developers using these patterns understand their meaning and how to implement them. Furthermore, a shared open source tool box where these patterns are already implemented - like in our factory example for the databases - reduce migration efforts and finally the portability issue.

Based on the previous idea, established frameworks like CAMF or TOSCA could be used to describe applications and annotate the components with metadata to get portable functions and use the previously mentioned building blocks based on metadata specified at design time.

Lastly, open source platforms were not considered at all, when talking about portability. Since many open source platforms like OpenFaaS or Knative use a sort of Kubernetes abstraction, the question would be if this already solves the portability issues to some extent.

REFERENCES

Antoniades, D. et al. (2015). Enabling cloud application portability. In *Proc. of UCC*, pages 354–360.

Armbrust, M. et al. (2010). A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58.

Baldini, I. et al. (2017). Serverless Computing: Current Trends and Open Problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer Singapore.

Binz, T. et al. (2013). TOSCA: Portable automated deployment and management of cloud applications. In *Advanced Web Services*, pages 527–549. Springer New York.

Boehm, B. W. et al. (1976). Quantitative evaluation of software quality. In *Proc. of ICSE*.

Ellis, B. et al. (2007). The factory pattern in API design: A usability evaluation. In *Proc. of ICSE*.

Fischer, R. et al. (2013). Eine bestandsaufnahme von standardisierungspotentialen und -lücken im cloud computing. In *Proc. of WI*.

Gonidis, F. et al. (2013). Cloud application portability: An initial view. In *Proc. of BCI*.

Hall, A. and Ramachandran, U. (2019). An execution model for serverless functions at the edge. In *Proc. of IoTDI*.

Hohpe, G. and Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional.

ISO/IEC/IEEE (2010). *24765-2010 - ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary*.

Jonas, E. et al. (2019). Cloud Programming Simplified: A Berkeley View on Serverless Computing. Technical Report UCB/EECS-2019-3.

Kolb, S. (2019). *On the Portability of Applications in Platform as a Service*. Bamberg University Press.

Kolb, S. et al. (2015). Application migration effort in the cloud - the case of cloud platforms. In *Proc. of CLOUD*.

Lenhard, J. and Wirtz, G. (2013). Measuring the portability of executable service-oriented processes. In *Proc. of EDOC*.

Rajlich, V. and Bennett, K. (2000). A staged model for the software life cycle. *Computer*, 33(7):66–71.

van Eyk, E. et al. (2018). Serverless is More: From PaaS to Present Cloud Computing. *IEEE Internet Computing*, 22(5):8–17.

Wolski, R. et al. (2019). Cspot: Portable, multi-scale functions-as-a-service for iot. In *Proc. of SEC*.

Yussupov, V. et al. (2019). Facing the unplanned migration of serverless applications. In *Proc. of UCC*.

Yussupov, V. et al. (2020). SEAPORT: Assessing the portability of serverless applications. In *Proc. of CLOSER*.