AFCL Environment
Development and Scalable Execution of Portable Function Choreographies Across Multiple Serverless Cloud Platforms

Master Thesis

Supervisors:
Dr. Sashko Ristov, Univ.-Prof Dr. Thomas Fahringer

Stefan Pedratscher
Stefan.Pedratscher@student.uibk.ac.at

Innsbruck
15 April 2021
Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht.

----------------------------------
Datum                                  Unterschrift
Acknowledgments

Throughout the writing of this master thesis, I have received a great deal of support and assistance. I would like to thank my supervisor, Dr. Sashko Ristov, for his great support, encouragement, motivation and excellent cooperation during this scientific process. In addition, I would like to thank my co-supervisor Univ.-Prof Dr. Thomas Fahringer for the feedback and guidance during the project. Last but not least, I would like to thank my parents and sister who made this thesis even possible.
Abstract

Function-as-a-Service is an increasingly popular paradigm to run distributed applications in the form of lightweight function in their preferred cloud provider. Developers simply provide the code, while the resource provisioning, configuration, scaling, and overall management of the cloud functions is taken over by the cloud providers. However, more challenging is to build complex serverless applications in a form of a workflow, also known as function choreographies, where the developers connect functions through various control and data flow dependencies. The developer has two options to develop FCs, either by porting the existing monolithic applications to the serverless environment or by developing FCs from scratch. Both approaches have pros and cons. While the former reuses the existing code to orchestrate serverless functions, it usually does not fully exploit the serverless architecture. Moreover, the cohesive monoliths commonly have many interdependencies and migrating part of the code as serverless functions naively often cause failures. On the other hand, building FCs from scratch may benefit from better granularity, but may increase the development effort. In either way, developers are faced with many challenges and have to live with many trade-offs for non-functional requirements.

Developed FCs are usually non-portable and huge development effort is needed to port FCs from one to another provider. This operation can be time consuming and error-prone because all serverless functions of an FC have to be ported to another FaaS system and their orchestration within an FC has to be redeveloped as there is no portability service.

To alleviate the above-mentioned challenges, this thesis introduces the AFCL Environment, a collection of modules which aim to solve provider-lock-in, allow an easy way to port existing applications to the cloud systems and execute FCs across different cloud providers. The goal of the AFCL Environment is to facilitate FC development and at the same time to allow portable and scalable execution across multiple platforms. Evaluations of the different modules show that the system advances beyond state-of-the-art.
Preface

This master thesis introduces the *AFCL Environment*, a platform that simplifies development of serverless applications in a form of serverless workflows and is able to run them across multiple cloud providers.

Scientific contribution

This thesis follows the research openness. All outputs of the thesis are publicly available on several Github repositories for software practitioners. Furthermore, several datasets are also publicly accessible for tracing and data analysis [1, 2, 3].

The outputs of this research and development-based thesis are recognized by the research community. The research resulted in two published scientific papers in highly recognized journals and magazines.

Personal contribution in the published papers

In the first published paper [4] we introduced Abstract Function Choreography Language (AFCL), which allows serverless application developers to orchestrate functions in a workflow at a high-level of abstraction. This paper is co-authored with Sasko Ristov and Thomas Fahringer. I led the development of the *AFCLCore* Java API and AFCL schema together with the other two authors and contributed in experiment setup and evaluation.

Further on, in the second published paper [5] we introduced a novel approach to develop serverless functions by annotating NodeJS monoliths. This paper is co-authored with Sasko Ristov, Jakob Wallnöfer, and Thomas Fahringer. Jakob Wallnöfer is a master student. My contribution was twofold. Firstly, I reviewed the related work in the area of FaaSifiers with the focus on NodeJS monoliths. My investigation determined several weaknesses, which were considered in the development of DAF. Secondly, I set up the testbed and contributed in the evaluation of DAF compared with two other state-of-the-art tools.

Moreover, as a result of the research in this thesis, a portable and scalable tool *jFaaS* is introduced, which allows development of portable Java applications that orchestrate serverless functions across multiple FaaS systems.
**Content of the thesis**

The content of this master thesis is organized in six chapters. The chapters are organized in a top-down approach. After presenting the introduction and motivation in Chapter 1, Chapter 2 shows the overall output of the thesis, i.e., *AFCL Environment*. Afterwards, all modules of the *AFCL Environment* are elaborated in a separate chapter. The following paragraphs summarize the content of each chapter.

**Chapter 1: Introduction** introduces the motivation for this master thesis. It explains the challenges of the current serverless technologies and the need for a system that will simplify development and maximize the performance of serverless workflow applications. Finally, it summarizes the contributions of this master thesis.

**Chapter 2: AFCL Environment** describes the overall *AFCL Environment* prototype. It puts together all tools and modules that support the overall lifecycle of serverless workflow applications developed in NodeJS, Java, and AFCL. This includes converting monoliths as an FC, running portable serverless functions from a monolith, and development and portable execution of FCs across multiple FaaS systems.

**Chapter 3: Dependency-aware Monoliths FaaSification** presents *DAF*, the *Dependency-Aware FaaSifier*, which simplifies the development of serverless functions from an existing monolithic code. *DAF* allows application developers to annotate methods of a monolith that should be faaSified, and *DAF* builds the equivalent serverless functions automatically, with all dependent declarations and external packages that the method uses. Chapter 2 presents the evaluation of *DAF*, which is superior compared to the state-of-the-art approaches in terms of automation and language constructs coverage. The conducted research for this chapter resulted in a published scientific paper in an IEEE magazine [5].

**Chapter 4: Portable and Scalable Serverless Applications Development** describes our novel Java-based tool *jFaaS* to overcome portability and scalability challenges of individual FaaS providers. *jFaaS* allows developers to develop portable and scalable serverless applications that are able to invoke serverless functions on any provider and written in any programming language that is supported by the provider. This chapter presents the architecture of *jFaaS* and how developers may develop portable applications by simply providing locations and data inputs of the serverless functions that should be invoked across multiple FaaS providers, without developing low-level details of each FaaS provider library. Finally, Chapter 3 shows the evaluation of *jFaaS*, which achieved 2.6 times higher throughput compared to the execution on a single FaaS provider at a time as the state-of-the-art portability tools.

**Chapter 5: Portable Serverless Workflow Applications at a High-level of Abstraction** describes details about our novel language AFCL, which allows application developers to build function choreographies and orchestrate functions into serverless workflows by data- and control-flow. Further on, Chapter 4 introduces the AFCL specification, including a rich set of constructs to express advanced control-flow and data-flow constructs in AFCL. Later on, it presents how application developers can build serverless workflows from different domains. Finally, Chapter 4 presents the evaluation of the same
realistic applications encoded with AWS Step Functions, IBM Composer and AFCL as serverless workflows. The output of the research that was performed for this Chapter resulted in a published open access journal paper [4].

Chapter 6: Conclusion and Future Work concludes the work conducted in this thesis and presents the plans for the future work.
## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Preface</strong></td>
<td>v</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>2 AFCL Environment</strong></td>
<td>3</td>
</tr>
<tr>
<td>2.1 The AFCL Environment stakeholders</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Function development</td>
<td>4</td>
</tr>
<tr>
<td>2.3 FC development</td>
<td>4</td>
</tr>
<tr>
<td>2.4 FC execution</td>
<td>5</td>
</tr>
<tr>
<td><strong>3 DAF: Dependency-aware Monoliths Faasification</strong></td>
<td>6</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>6</td>
</tr>
<tr>
<td>3.2 Related Work</td>
<td>7</td>
</tr>
<tr>
<td>3.3 DAF Faasification</td>
<td>8</td>
</tr>
<tr>
<td>3.3.1 Monolith annotation</td>
<td>9</td>
</tr>
<tr>
<td>3.3.2 Faasification of the annotated methods</td>
<td>9</td>
</tr>
<tr>
<td>3.4 Evaluation</td>
<td>9</td>
</tr>
<tr>
<td>3.5 Discussion</td>
<td>11</td>
</tr>
<tr>
<td>3.5.1 DAF's limitations</td>
<td>11</td>
</tr>
<tr>
<td>3.5.2 Further Faasification challenges</td>
<td>11</td>
</tr>
<tr>
<td>3.6 Summary</td>
<td>12</td>
</tr>
<tr>
<td><strong>4 Portable and Scalable Serverless Applications Development</strong></td>
<td>13</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>13</td>
</tr>
<tr>
<td>4.2 Related Work</td>
<td>14</td>
</tr>
<tr>
<td>4.3 jFaaS software architecture and design</td>
<td>15</td>
</tr>
<tr>
<td>4.3.1 Overview of the jFaaS architecture</td>
<td>15</td>
</tr>
<tr>
<td>4.3.2 jFaaS software design</td>
<td>16</td>
</tr>
<tr>
<td>4.4 Evaluation</td>
<td>18</td>
</tr>
<tr>
<td>4.5 Summary</td>
<td>19</td>
</tr>
<tr>
<td><strong>5 Portable Serverless Workflow Applications at a High-level of Abstraction</strong></td>
<td>20</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>21</td>
</tr>
<tr>
<td>5.2 Related work</td>
<td>22</td>
</tr>
<tr>
<td>5.2.1 Task-based workflows</td>
<td>22</td>
</tr>
<tr>
<td>5.2.2 Dataflow-based workflows</td>
<td>22</td>
</tr>
<tr>
<td>5.2.3 Scientific FC systems</td>
<td>23</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Commercial FC systems</td>
</tr>
<tr>
<td>5.3</td>
<td>AFCL Specification</td>
</tr>
<tr>
<td>5.3.1</td>
<td>AFCL portable approach with a high-level of abstraction</td>
</tr>
<tr>
<td>5.3.2</td>
<td>General overview of AFCL</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Base functions</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Compound functions</td>
</tr>
<tr>
<td>5.3.5</td>
<td>Invocation type of a function</td>
</tr>
<tr>
<td>5.3.6</td>
<td>Data-flow in AFCL</td>
</tr>
<tr>
<td>5.3.7</td>
<td>Event-based invocation</td>
</tr>
<tr>
<td>5.4</td>
<td>Composing FCs with different FC systems</td>
</tr>
<tr>
<td>5.4.1</td>
<td>FC applications composed with AFCL</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Composing FCs with AWS_STEP and IBM_COMP</td>
</tr>
<tr>
<td>5.5</td>
<td>Evaluation</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Testing methodology</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Development effort evaluation</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Makespan evaluation</td>
</tr>
<tr>
<td>5.5.4</td>
<td>Economic cost evaluation</td>
</tr>
<tr>
<td>5.5.5</td>
<td>Discussion</td>
</tr>
<tr>
<td>5.5.6</td>
<td>Threats to validity</td>
</tr>
<tr>
<td>5.6</td>
<td>Summary</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion and Future work</td>
</tr>
</tbody>
</table>
1 Introduction

Serverless computing and especially the programming paradigm Function-as-a-Service (FaaS) is one of the fastest growing cloud technology in the recent years. In such FaaS system the developers simply create and upload functions to the cloud providers, which manage the underlying infrastructure. The system automatically tears down or sets up function containers depending on what is needed, while the user can fully focus on development. There is no need to provision, configure or manage servers and therefore many companies build their applications using serverless technologies. The first task to run an application in the serverless environment is to bring functions to the cloud providers. In general, it is very tedious to port an existing monolithic application to the serverless environment. The second step is to connect these cloud functions by control- and data-flow constructs to so called function choreographies (FCs) or serverless workflows.

Many big cloud providers introduced services to ease the life of developers. A very well known service is the execution system for FCs. All big companies introduce a service which aims to easily run workflows. Nevertheless, this results also in different, provider specific, approaches on how the users need to interact with these services. Each provider has its own language describing the function choreography, its own engine to run the workflow and it is not possible to run functions of different providers within a choreography for the current systems. Additionally, each provider introduces hard limits like the maximum amount of cloud functions which can run in parallel.

The master thesis tackles the above-mentioned challenges and advances beyond state-of-the-art in several areas. Firstly, it contributes in FaaSification of existing monoliths considering various dependencies in the cohesive monoliths. The Dependency-Aware FaaSifier (DAF), which is an output of this thesis, resolves code and package dependencies of each faasified method automatically. DAF automatically builds an equivalent serverless function for each annotated method, which includes all dependent declarations and external packages that the method uses. Afterwards, DAF replaces the monolith method code with an API call to the equivalent serverless function.

Secondly, we introduced a multi-FaaS toolkit jFaaS, which facilitates the development of multi-FaaS portable and scalable serverless Java applications. Developers can invoke functions across multiple FaaS providers at a time through a single object of a class, without rewriting, rebuilding, testing and redeploying the application. Instead, developers need simply to provide the function location as input parameter and the data input of the function they invoke.

Thirdly, we designed the Abstract Function Choreography Language (AFCL), which offers a rich set of control-flow and data-flow constructs to build serverless workflow applications at a high-level of abstraction. Further on, a Java API AFCLCore is developed.
to easily build, parse, write and validate serverless workflow applications. The Java API is integrated into several stand-alone tools that resulted from bachelor theses. Finally, \textit{AFCLCore}-based tools and other above mentioned tools were integrated into the novel \textit{AFCL Environment}, which is able to develop and run portable serverless applications across multiple serverless cloud platforms.


## 2 AFCL Environment

The result of this master thesis is the *AFCL Environment*, which facilitates the development of serverless functions and entire FCs and offers their portable and scalable execution across multiple FaaS systems (AWS Lambda, Google Cloud Functions, Microsoft Azure Functions, Alibaba Function Compute, and IBM Cloud Functions). It supports the development and execution of various types of FCs. The *AFCL Environment* is a middleware between the user and the target FaaS systems.

The *AFCL Environment* integrates several modules, as shown in Fig. 2.1. The following sections give a high-level overview of each module and various functionalities of how to develop and run entire FCs across multiple supported FaaS systems. Afterwards, the following three chapters present our motivation, weaknesses in the related work, low-level details of their design, and evaluation.

![AFCL Environment Diagram](image)

**Figure 2.1: The system architecture of the AFCL Environment.**

The *FC Editor* and the *Enactment Engine* are a result of two bachelor theses [6, 7]. Both modules are integrated in the *AFCL Environment* by integrating *AFCLCore* as a part of this master thesis. *FC Editor* uses *AFCLCore* to parse the loaded AFCL file and convert it into a graphical representation. Once the user updates the existing FC or builds a new one, the *AFCLCore* is used to validate and create the AFCL file. The *Enactment Engine* uses *AFCLCore* to parse the given AFCL file and transform it into

---

1[http://fceditor.dps.uibk.ac.at:8180/], accessed on 26.03.2021
2[https://github.com/sashkoristov/enactmentengine accessed on 26.03.2021]
Java objects. Further on, the *Enactment Engine* uses them to control the execution of the entire FC.

### 2.1 The AFCL Environment stakeholders

The *AFCL Environment* introduces three stakeholders to allow scalable and portable execution of FCs across multiple FaaS systems: an *FC developer* (*FCDev*), a *function developer* (*fDev*), and a *user*. Each stakeholder has a separate role in the *AFCL Environment*. *FCDev* is the stakeholder that builds the business logic of the functions choreographies by creating and connecting function types with control and data flow. For such function types, *fDevs* develop at least one implementation for each function type of the FC on at least one FaaS system. Finally, after FCs are fully developed, *users* may run them across multiple FaaS systems.

### 2.2 Function development

*fDev* is responsible to develop serverless functions, which are used later in FCs. *fDev* can create functions in two ways, either implement new functions manually or use the *Dependency Aware FaaSifier*, presented in Chapter 3, to automatically generate cloud functions out of a monolithic NodeJS application. *fDev* needs to supply the resource locations of the deployed cloud functions within the FC, so that FCs can invoke the specified functions.

### 2.3 FC development

The *AFCL Environment* offers the *FCDev* to develop FCs in three different ways (i) from existing NodeJS monolith applications, (ii) develop a new Java monolith with portable functions invocation, and (iii) portable and scalable FCs using AFCL. In either case, the functions may be developed in any programming language that is supported by the FaaS system where they are deployed.

**Develop FCs from existing monoliths.** *FCDev* may orchestrate FCs by automatic FaaSification from *DAF* and using the existing monolith as the orchestrator. In this case, the interface to the *user* is kept the same and function invocations are transparent to the *user*. The current prototype supports creation of AWS Lambda serverless functions and handles code and package dependencies.

**Develop new FCs with portable functions.** Secondly, *FCDev* may develop a new monolith and orchestrate and invoke functions in a portable way using *jFaaS*. The *FCDev* orchestrates functions with Java and invocation is developed in a portable way using the *invokeFunction* method of the *FaaSInvoker* interface. To achieve a high level of portability, *FCDev* can develop the FC such that all function locations may be input arguments. With this approach, FCs do not need to be built each time to change a single function.
Develop new portable and scalable FCs. Finally, FCDev can develop portable and scalable FCs in AFCL. The AFCL Environment offers the FC developer several prongs to compose AFCL FCs. One way is to create a workflow solely using the AFCLCore Java application programming interface. Another approach is to use the FC Editor, which provides a graphical user interface to build a workflow by dragging constructs on a web application. This tool is built on top of AFCLCore, which manages the validation and the creation of the AFCL file for the FC. Since the output of these tools is a simple .yaml file, it is of course also possible to manually create this file in a preferred editor of the FCDev choice.

2.4 FC execution

The AFCL Environment allows the users to run individual serverless functions using jFaaS. Further on, users may run the monoliths whose parts are offloaded as serverless functions in a transparent way. However, the main benefit of the AFCL Environment is the possibility of portable execution of scalable FCs that are developed by FCDev’s in AFCL. Since AFCL offers a rich set of control- and data-dependencies, including parallel loops and data distribution, users may run FCs with complex dependencies or a high degree of parallelism across all five supported public cloud providers. Optionally, users provide the JSON based input to the FC and the AFCL Environment returns the output of the successful finished FC.
Serverless computing is emerging as a popular paradigm for the deployment of applications over the edge-cloud continuum. However, porting an existing monolith to a serverless platform ("FaaSification") can be challenging because monoliths commonly have many interdependencies and are highly cohesive. Migrating monolith’s methods to serverless functions and linking them via API calls often causes failures in the serverless functions due to unfulfilled code and package dependencies.

This chapter introduces the Dependency-Aware FaaSifier (DAF), which resolves code and package dependencies of each faasified method automatically. DAF allows developers to annotate all methods that should be faasified. After the annotation, DAF proceeds with the automatic FaaSification in two steps. Firstly, DAF automatically builds an equivalent serverless function for each annotated method, which includes all dependent declarations and external packages that the method uses. Secondly, DAF replaces the monolith method code with an API call to the equivalent serverless function.

Experimental evaluation demonstrated that even for a simple monolith, the state-of-the-art faasifier Node2FaaS required numerous manual code changes to include external libraries/methods and adapt the algorithm, while DAF’s faasification was fully automatic.

3.1 Introduction

Serverless computing became a widely used cloud technology in recent years. Developers simply upload the code of their functions, while the underlying platform and infrastructure is managed by FaaS providers. While building a new serverless application from scratch is a straightforward process, faasifying an existing monolith raises two main challenges.

Firstly, the monolithic architectural style commonly assumes that all files and external dependencies are in the global scope and accessible from all methods of the application. Serverless architectural style, however, defines that functions are isolated within their own runtime environment, where external files and dependencies such as NPM packages may not be in the scope. Developers must carefully include all required packages and files, recursively, with each serverless function. For a real-world application, this manual process is a very costly operation. For example, GitHub projects have on average 200 package dependencies, reaching 1000 and above in some cases.

Secondly, monoliths comprise many inter-method (code) dependencies. For exam-
3.2 Related Work

Spillner et al. [11] respectively introduced three levels of Faasification (shallow, medium and deep), based on the atomic units of Faasification (methods, lines of code, or instructions). But, only a few works can be found in the research community regarding monolith shallow Faasification. The first attempt for Faasification of a single Python method is Lambada [12]. The same authors introduced Podilizer [13], which faasifies Java monoliths. However, Podilizer is still in its early stage with multiple restrictions on the input Java project.

PyWren [14] uses another approach for running Python code as serverless functions. Instead of deploying the function code, both the code and the data input are fetched from the storage by a single function (the executor). However, this approach requires additional developer effort in case there are multiple dependencies and generates additional runtime overhead to fetch the code.

Node2FaaS [15] converts methods of NodeJS monolith into serverless functions and replaces their bodies with an API call to the target FaaS system. However, we detected several weaknesses of node2faas. Firstly, it neither resolves code nor package dependencies. This means that Node2FaaS can faasify methods that are fully self-contained, and do not access to any outer functions or packages. The unfit monolith code would have to be re-structured first, which is usually not viable. Secondly, the generated serverless function has to be checked by hand for correctness. If Node2FaaS selects a method for outsourcing that is not self-contained, the resulting serverless function is invalid. Worse, this will only be evident at runtime, causing the entire hybrid monolith to fail.
3 DAF: Dependency-aware Monoliths FaaSification

3.3 DAF FaaSification

DAF (publicly available [16]) automatically faasifies even highly cohesive NodeJS monoliths. We have selected to support NodeJS, as it is the most popular programming language on GitHub by repository contributors [9]. Developers simply annotate methods that should be faasified, without changing the semantics of the monolithic. Based on the annotations, DAF outputs a hybrid application comprising the converted monolith and the equivalent AWS lambda serverless functions of each annotated method. Unlike the related work, DAF resolves all code and package dependencies.

Fig. 3.1 presents how DAF faasifies a NodeJS monolith. As an example, we use the widely known N Queens problem, which consists of finding all placements of N queens on an N · N chess board such that no queen is under attack. We developed the N Queens problem as a monolith that has two files [17]. As presented in Fig. 3.1 left, nqueens.js contains the method fraction(), which iterates over several queen placements in a specific range and calls acceptable() (declared in external.js) for each queens’ placement to check its validity. NodeJS scripts are invoked by the shell script workers.sh, which takes as an input the number of queens numberOfQueens, how many parallel workers theNumberOfWorkers to be started, and the path to the script which should be executed (nqueens.js). Each worker represents a UNIX process and calls fraction() for a specific range of queen placements.

*Figure 3.1: DAF FaaSification of N Queens. The method fraction() has both code and package dependencies (external.js and lodash). DAF bundles them recursively, and builds the serverless function λfraction(). Furthermore, it replaces the outsourced method body with an API call.*

Our implementation of N Queens contains two types of dependencies, code and package dependencies, which cannot be handled by the related work. N Queens contains
3.4 Evaluation

a package dependency because `fraction()` uses `range()` from the external package `lodash`. It also contains code dependency because `fraction()` calls `acceptable()`, which is defined in another file, `external.js`. Finally, another challenge is to resolve the return value.

3.3.1 Monolith annotation

Developers start FaaSification by adding annotations to the monolith’s source code (Step 1 in Fig. 3.1). The annotations instruct `DAF` which methods to faasify and how to generate serverless functions from them. In our example, we annotate `fraction()` to be extracted as the serverless function `\(\lambda\) fraction()`.

The annotation “// 1” denotes the start of a method to be faasified, while “// lend” denotes the end (Fig. 3.1 top). The // 1 header can be followed by additional directives. In our example, we name the deployment package with `name(\(\lambda\) fraction)`. This helps us to keep track of it, and allows for delta recompilation. We then specify `install(lodash)` to include the `lodash` package with the code. The `fraction()` method also depends on the local method `acceptable()` from `external.js`. We specify `require(./external.js as external)` to make it available in the equivalent serverless function `\(\lambda\) fraction()`. Internally, `DAF` bundles it with its dependencies using webpack, and adds it to the `\(\lambda\) fraction()`’s scope. Similarly, `require(lodash as _)` adds `lodash` to the scope. Finally, `return(solutions)` ensures that the variable `solutions` is returned from `\(\lambda\) fraction()` as JSON. The directives give much control over the generated serverless function, without changing the semantics of the original code.

3.3.2 FaaSification of the annotated methods

Based on the annotations, `DAF` starts to build deployment packages by linking all annotated package dependencies and method definitions. `DAF` parses the annotation header (“// 1”) and the enclosed method code until “// lend”. Then, `DAF` builds an equivalent serverless function `\(\lambda\) fraction()`, including code dependencies (`external.js`), package dependencies (`lodash`), and `context.succeed(solutions)`, as presented in the right part of Fig. 3.1. Additionally, `DAF` converts the original monolith by replacing the code body of `fraction()` with an API call to `\(\lambda\) fraction()`.

3.4 Evaluation

We conducted a set of three experiments (publicly available [17]) to evaluate the FaaSification of the state-of-the-art tool `Node2FaaS` and `DAF`. We evaluated whether `\(\lambda\) fraction()` runs successfully and its performance with 1, 10, 20, and 30 workers.

We used hybrid resources in AWS `eu-central-1`. We selected `t2.micro` used for `Node2FaaS` validation [15] and we configured `\(\lambda\) fraction()` with 256MB memory to provide better speedup [18]. We run each experiment with problem size \(N = 9\). In order to have statistically significant results, we repeated each experiment six times and considered the average of the last five executions to ignore the `\(\lambda\) fraction()` cold start.
After deploying \( \lambda \text{fraction}() \) on AWS with Node2FaaS, we had to make several manual changes to resolve dependencies and to invoke it. Firstly, AWS returned error message "_. is not defined". However, even after adding `const _ = require('lodash')`, \( \lambda \text{fraction}() \) failed again returning "Error: Cannot find module 'lodash'", which required manually adding the lodash package to \( \lambda \text{fraction}() \). We had to perform similar manual transformations for \( \lambda \text{acceptable}() \). Afterwards, \( \lambda \text{fraction}() \) failed again with the error message "result is not defined", alluding to a Node2FaaS code convention that the monolith did not meet. Since this error required changing the semantics of the monolith, we stopped with our manual trials. On the contrary, DAF converted the monolith into a valid hybrid and \( \lambda \text{fraction}() \) worked right away.

Since Node2FaaS \( \lambda \text{fraction}() \) always failed without manual code refactoring, we evaluated the performance of DAF against the monolithic version of \( N \text{ Queens} \) (see Fig. 3.2). We observed that the execution time of the monolith did not scale in strong scaling (average of 158 – 171 seconds) due to limited resources of t2.micro. On the contrary, the faasified solution of DAF scaled and reduced the execution time to 47 seconds using \( w = 30 \) workers, which is \( S_{rel} = 3.64 \) times faster compared to the monolith. However, Faasification should not be considered as the default solution for any application. For example, \( \lambda \text{fraction}() \) failed due to the duration limit of 900 seconds when running it with a single worker (\( w = 1 \)). On the other side, assignment of 1GB, the same as for the monolith, helped \( \lambda \text{fraction}() \) to finish, but in 242 seconds, which is 53.16\% longer than the execution time of the monolith.

![Figure 3.2: Execution time and relative speedup as a function of the number of workers of faasified with DAF vs. monolithic N Queens.](image)

To investigate the achieved speedup and efficiency of faasified \( N \text{ Queens} \) problem with DAF, we estimated the execution time of 968s (1GB/256MB·242s) for \( w = 1 \), assuming a linear speedup \([19]\). We observed significant speedup of up to 20.6 and efficiency of
3.5 Discussion

3.5.1 DAF’s limitations

DAF is limited to a single programming language because each programming language handles code and package dependencies differently to build deployment packages for serverless functions.

DAF does not consider FaaS provider specific constraints. For example, any function that writes to the file system would fail because AWS Lambda limits the function write access to the /tmp folder only. However, such provider-specific limitations have to be handled by the developer even for manual creation of serverless functions. Moreover, although DAF can handle an arbitrary number of code and package dependencies by using webpack, developers need to consider provider limitations in terms of package size.

3.5.2 Further FaaSification challenges

Other specific types of code dependencies are nested functions, recursive calls of the same function, or chained calls of several functions. In all of these cases, the caller function will be active until the callees terminate and return. Although DAF can properly faasify “callee” functions (e.g. acceptable() as λacceptable()), still, the execution of caller functions (λfraction()) will not be ideal. Namely, λfraction() will block and wait λacceptable() to finish before it continues. Regardless of waiting and inefficient usage of resources, FaaS providers will charge the user for usage time of resources of the caller

Figure 3.3: Speedup and Efficiency as a function of the number of workers of faasified with DAF vs. monolithic N Queens.

68.6% for \( w = 30 \) (Fig. 3.3).
functions, which may significantly increase the cost compared to the monolith [20]. Even worse, \texttt{lambda()} will probably often reach the duration limit of the FaaS provider.

Another challenge is how to decompose the dependencies from the monolithic architectural style into serverless to effectively benefit from the serverless technology. The resulting hybrid application should retain the interfaces without being affected by FaaSification. Clients and other services should still be able to interact with the functionality in a transparent way [21], while the clients should not be aware where the computing is performed and who is serving the requests (the monolith or lambdas) [22].

### 3.6 Summary

\textit{DAF} advanced the state-of-the-art in automated FaaSification by resolving both code and package dependencies of each faasified method of a NodeJS monolith. Our novel approach with annotations does not require to change the semantics of the monolith. After the annotation, \textit{DAF} automatically builds equivalent serverless functions (including all dependent declarations and external packages that the method uses) and replaces the monolith method code with an API call.
4 Portable and Scalable Serverless Applications Development

All well-known cloud providers offer Function-as-a-Service while supporting multiple programming languages. Still, users are locked into the FaaS provider platform and porting serverless applications to another provider requires huge development effort. Moreover, the application scalability is limited because FaaS providers usually restrict the number of concurrent functions within a single region.

To overcome these limits, we introduce \textit{jFaaS}, a multi-FaaS toolkit, which facilitates the development of multi-FaaS portable and scalable serverless Java applications. \textit{jFaaS} offers developers to invoke functions across multiple FaaS providers at a time through a single object of a class, without rewriting, rebuilding, testing and redeploying the application, but simply by providing the function location as input parameter, an environment variable, a record from a database, or in a file. The development of portable serverless applications is very simple because developers need to provide only security credentials for all providers, the location of the function and its data input. \textit{jFaaS} will then invoke specified functions in a transparent way. We developed a prototype of \textit{jFaaS} that supports two back-end FaaS providers, AWS Lambda and IBM Cloud Functions, including both portability and FaaS provider specific configuration. \textit{jFaaS} achieved 2.6 times higher throughput compared to the execution on a single FaaS provider at a time as the state-of-the-art portability tools.

4.1 Introduction

Serverless is one of the main programming models and fastest growing cloud technology. Although in its early stage of development, serverless computing became very popular among developers because they upload the code of their functions and leave the management of the underlying platform and infrastructure to FaaS providers [8]. Companies tend to deploy their applications using serverless technologies to reduce their costs and improve flexibility [20]. However, despite the low startup of functions within a few milliseconds, well-known FaaS providers limit the scalability, i.e., to 200 containers [23] and 1,000 concurrent invocation of functions [24].

Recent surveys reported that on average, companies use cloud services of five clouds providers, including both public and private clouds [25] [26]. Such multi-cloud serverless platforms may bring multiple benefits for companies in terms of availability, cost, data privacy, performance, and support. Moreover, companies tend to avoid vendor lock-in with multi-cloud platforms. However, the lack of a common standard burdens developers’
life as they have to spend huge effort to integrate various libraries for each FaaS provider individually. Once the company management decides to switch the serverless application to another FaaS provider or to deploy it across multiple FaaS providers (e.g. to switch to another pricing model), all these low-level development codes need to be redeveloped to include libraries of the target FaaS providers. This process is usually tedious and error-prone, and needs significant development expertise [27]. At the end, both the development a new serverless application across multiple FaaS providers or migrating an existing serverless application to another FaaS provider have to go under complex and costly testing phase.

4.2 Related Work

Several related works covered the portability of cloud applications. Apache jclouds [28] allows development of portable cloud applications on various Infrastructure-as-a-Service in a transparent way by simply switching the provider’s name in the application. However, it supports only a single provider at a time without support for serverless. Similar approach for serverless applications was introduced by the MPSC platform [29], which can switch from a provider to a provider during runtime based on the specified scheduling targets. However, this approach restricts the execution to a single FaaS provider only at a time, either on AWS Lambda, or on IBM, which limits the application scalability to only 1,000 concurrent functions. Instead, jFaaS offers developers to specify concurrent execution of functions on multiple FaaS providers at a time, which may significantly increase the throughput, without any latency to switch the execution on another FaaS provider.

FuncX [30] introduced another approach for portability and scalability by running functions within containers. While this approach facilitates the development of portable serverless applications, it does not support multiple allocations and generates high startup latency compared to widely used FaaS providers, especially for cold start.

Node2FaaS framework [31], on the other side, supports porting of serverless functions from the monolithic NodeJS code using Terraform [32]. Still, the generated serverless application is ported on a single FaaS provider, i.e., either only to AWS Lambda, or IBM, or Google Cloud Functions, as specified in the Terraform’s infrastructure as a code script, without an option for a multi-FaaS scenario as jFaaS.

To the best of our knowledge, no tool exists that facilitates development of portable serverless applications that are able to run individual functions in a scalable way across multiple regions of multiple FaaS providers simultaneously, which motivated us to build jFaaS.
4.3 \textit{jFaaS} software architecture and design

4.3.1 Overview of the \textit{jFaaS} architecture

Figure 4.1 presents the system architecture of \textit{jFaaS}, which offers developers two-pronged approach to support invocation of multiple functions across various regions of multiple FaaS providers. The first prong offers developers a high-level of portability, i.e., to invoke a function by simply providing security credentials, a function that should be invoked and its data input. \textit{jFaaS} will automatically invoke this and all subsequent functions, regardless of the FaaS provider. For example, invoke a function in IBM Tokyo, then another one on Amazon North Virginia and yet another one on IBM Frankfurt, without the need to rewrite and rebuild the application. For this high-level portability, \textit{jFaaS} offers two classes (i) \texttt{HTTPGetInvoker}, which invokes functions via a simple HTTP GET request, and (ii) \texttt{Gateway}, which invokes functions via the specific library of the target FaaS provider by parsing the function resource name to determine the provider and the specific function that should be invoked. Function invocations through these two classes are transparent to developers.

The second prong allows developers to configure specifics for each provider/region individually. For example, to setup the maximum time to wait within a request for a specific AWS Lambda function. The trade-off for provider specific configurations is the lower level of portability because developers need to rewrite and rebuild their serverless application each time they need to change the execution plan. Nevertheless, the development effort is minimal as developers need to create an object of another class that implements the same interface \texttt{FaaSInvoker}.

The current prototype of \textit{jFaaS}, which is available on GitHub, supports two FaaS providers AWS Lambda and IBM Cloud Functions. Unlike the other tools that allow portable execution in a single region of a single provider at a time, \textit{jFaaS} portability allows developers to invoke functions that are deployed in various regions of both providers simultaneously. This feature allows serverless applications higher throughput.
as it overcomes the limitations of individual FaaS provider (1,000 concurrent invocations).

### 4.3.2 jFaaS software design

In order to abstract FaaS providers and simplify portability, jFaaS contains the FaaS-Invoker interface, whose method `invokeFunction` accepts two input parameters, i.e., the function name (`function`), and its inputs (`functionInputs`). Further on, all other classes of jFaaS implement this interface.

#### Provider-specific classes

Listing 4.1 shows the implementation of the `LambdaInvoker` class, which allows developers to create a separate object for each AWS region and then invoke multiple functions in that region. `LambdaInvoker` contains two constructors to create an object, one for a standard (default) configuration and another one for custom configuration (`ClientConfiguration`). For example, to set up `setMaxErrorRetry`, i.e., the default number of retries if a function fails. Finally, the `invokeFunction()` method invokes a lambda function with the same two parameters from the `FaaSInvoker` interface, i.e., (i) function ARN (Amazon Resource Name) as `function` and (ii) data inputs to the function (`functionInputs`).

```java
public class LambdaInvoker implements FaaSInvoker {

    public LambdaInvoker(String awsAccessKey, String awsSecretKey, Regions region) {
        // ...
    }

    public LambdaInvoker(String awsAccessKey, String awsSecretKey, Regions region,
                          ClientConfiguration clientConfiguration) {
        // ...
    }

    @Override
    public JsonObject invokeFunction(String function, Map<String, Object>
                                      functionInputs) throws IOException {
        // ...
    }
}
```

Listing 4.1: Implementation of the `LambdaInvoker` class to invoke portable functions on AWS Lambda.

Similarly, the `OpenWhiskInvoker` class enables developers to invoke functions on IBM Cloud Functions and OpenWhisk. It has a constructor to create an object with the access key and also implements the same `FaaSInvoker` interface. Developer need to create a single object to invoke functions on IBM, regardless of the region.
4.3 \textit{jFaaS} software architecture and design

Classes for portability and scalability

Since well-known FaaS providers offer function invocations over HTTP, \textit{jFaaS} introduces the \texttt{HTTPGETRequest} class, which can be used to invoke portable functions via a simple HTTP GET request. It also implements the same \texttt{FaaSInvoker} interface. Internally, \texttt{HTTPGETRequest} builds the URL for the GET request.

The \texttt{Gateway} class offers a different approach for portability than \texttt{HTTPGetRequest}, i.e., it automatically creates an object of the specific provider Invoker class (e.g. \texttt{LambdaInvoker} for AWS Lambda functions) based on the function that should be invoked. The \texttt{Gateway} class keeps the object of an invoker alive if the subsequent function is invoked in the same provider (IBM) and in the same region (AWS). Otherwise, a new object of the corresponding invoker class is instantiated.

Listing 4.2 presents the implementation of the \texttt{Gateway} class. The constructor loads the credentials of each FaaS provider from \texttt{credentialsFile}. It also implements the \texttt{FaaSInvoker} interface. The current prototype implementation of \textit{jFaaS} and the \texttt{invokeFunction} method parses the function name to determine whether it contains "arn:" (for AWS) or "functions.cloud.ibm" (for IBM). Based on that, the object of the \texttt{Gateway} class creates a corresponding object of \texttt{LambdaInvoker} or \texttt{OpenWhiskInvoker} class, or uses and already existing object, created from previous function executions. Moreover, the region of the AWS Lambda function is determined using the \texttt{detectRegion} method and is also passed to the \texttt{LambdaInvoker} constructor. Once the object of the corresponding Invoker class is created, \texttt{invokeFunction} can be called with the function name and its inputs.

```java
public class Gateway implements FaaSInvoker {
    public Gateway(String credentialsFile) {
        // Load the corresponding credentials
    }
    @Override
    public JsonObject invokeFunction(String function, Map<String, Object> functionInputs) throws IOException {
        // use LambdaInvoker if function contains "arn:"  
        // use OpenWhiskInvoker if function contains "functions.cloud.ibm"
    }
}
```

Listing 4.2: Implementation of the \texttt{Gateway} class to invoke portable functions across multiple regions of various FaaS providers. It uses \texttt{LambdaInvoker} to invoke functions on AWS and \texttt{OpenWhiskInvoker} for IBM or OpenWhisk without rewriting and rebuilding the code.
4 Portable and Scalable Serverless Applications Development

4.4 Evaluation

We conducted an experiment to evaluate the throughput of jFaaS when running a portable serverless application across AWS Lambda and IBM Cloud Functions, simultaneously. For this purpose, we developed a "no-op" function written in Python, which simply returns immediately. The same function was deployed on AWS Frankfurt (AF), and two regions of IBM Cloud Functions, i.e., IBM Frankfurt (IF), and IBM Tokyo (IT). Each "no-op" function was assigned with 128MB memory.

A simple serverless application using the Gateway class was developed to invoke 1,000 concurrent "no-op" functions in each of the three regions simultaneously. After running the experiment, once one of the regions finished all 1,000 functions, we recorded the execution time and measured how many functions finished in all three regions up to that time. Finally, we calculated the achieved throughput as a ratio of the total number of finished functions and the execution time. In order to have statistically significant results, we repeated each execution three times and calculated the average values for execution time and the achieved throughput.

Figure 4.2 presents the results of the evaluation. We observed that AF finished all 1,000 functions within 6.37 seconds, in front of IF with 7.92 and IT with 9.24 seconds. However, up to the threshold of 6.37 seconds, IF finished in average 807 functions, while IT only 692. Overall, until the threshold of 6.37 seconds, our portable application finished 2,499 functions, which created a throughput of 0.39 "no-op" functions per second. This throughput is 2.5 times higher than the throughput achieved by using a single provider AF, as supported by the state-of-the-art portable tools. The efficiency of scaling resources is 83.3%. The achieved throughput could be increased even more if more regions are used.

Figure 4.2: Execution time of invoked functions in AWS Frankfurt (AF), IBM Frankfurt (IF) and IBM Tokyo (IT) regions. Within the threshold time of 6.37 seconds when AF finished 1,000 functions, IF finished 807, while IT 692 functions.
4.5 Summary

*jFaaS* is a big step forward in building portable and scalable serverless applications. It also offers the runtime system to select which function to run across the multiple FaaS providers in order to improve the overall performance or reduce costs.
5 Portable Serverless Workflow Applications at a High-level of Abstraction

Serverless workflow applications or function choreographies (FCs), which connect serverless functions by data- and control-flow, have gained considerable momentum recently to create more sophisticated applications as part of Function-as-a-Service (FaaS) platforms. Initial experimental analysis of the current support for FCs uncovered important weaknesses, including provider lock-in, and limited support for important data-flow and control-flow constructs. To overcome some of these weaknesses, we introduce the Abstract Function Choreography Language (AFCL) for describing FCs at a high-level of abstraction, which abstracts the function implementations from the developer. AFCL is a YAML-based language that supports a rich set of constructs to express advanced control-flow (e.g. parallelFor loops, parallel sections, dynamic loop iteration counts) and data-flow (e.g. multiple input and output parameters of functions, DAG-based data-flow). We introduce data collections which can be distributed to loop iterations and parallel sections that may substantially reduce the delays for function invocations due to reduced data transfers between functions. We also support asynchronous functions to avoid delays due to blocking functions. AFCL supports properties (e.g. expected size of function input data) and constraints (e.g. minimize execution time) for the user to optionally provide hints about the behaviour of functions and FCs and to control the optimization by the underlying execution environment. We implemented a prototype AFCL Environment that supports AFCL as input language with multiple backends (AWS Lambda and IBM Cloud Functions) thus avoiding provider lock-in which is a common problem in serverless computing. We created two realistic FCs from two different domains and encoded them with AWS Step Functions, IBM Composer and AFCL. Experimental results demonstrate that our current implementation of the AFCL Environment substantially outperforms AWS Step Functions and IBM Composer in terms of development effort, economic costs, and makespan.

This chapter is organized into several sections. Section 5.1 introduces the motivation for our AFCL. Section 5.2 compares with the related work and presents the deficiencies of widely used FC systems, which was the starting point to develop AFCL in order to overcome these deficiencies. The specification of AFCL for FCs at a high-level of abstraction, including function descriptions and data- and control-flow, is elaborated in detail in Section 5.3. We have carefully selected a diverse set of realistic FCs and built them in Section 5.4 in order to evaluate AFCL (Section 5.5). Finally, we conclude our work and present the plans for the future work in Section 5.6.
5.1 Introduction

Cloud computing is moving towards an adaptive environment that tries to provide elastic and scalable services in real time. Function-as-a-Service (FaaS) is a recent programming paradigm that supports serverless computing, simplifies code deployment, eliminates the need for manual resource provisioning, provides high elasticity with low latency which can provision services within a few milliseconds [34]. The most basic scenario is to invoke functions based on events. In order to build more complex applications, functions can be connected by data- and control-flow to form so called function choreographies (FCs) or workflows of functions. Most public cloud providers and open source projects for serverless computing have introduced platforms to support FCs.

Although FaaS platforms continuously advance the support for FCs, existing FC languages and runtime systems are still in their infancy with numerous drawbacks. Current commercial and open source offerings of FC systems are dominated by a few large-scale providers who offer their own platforms, resulting in mostly non-portable FaaS solutions and provider lock-in [35]. Our initial study revealed three important observations that motivated this chapter. Firstly, several important features of FCs (e.g., a function with multiple outputs, parallelFor loops, and efficient data-flow support among multiple functions) are only partially supported. These limitations can cause considerable coding efforts to develop workaround solutions [36]. Secondly, even if workaround solutions exist, they may degrade performance (e.g., delayed invocation of a function) [37]. Finally, none of the evaluated FaaS systems appear to dominate all others in terms of language features to describe complex FCs.

To overcome these weaknesses, we introduce a new Abstract Function Choreography Language (AFCL), which is a novel approach to specify FCs at a high-level of abstraction. AFCL has many numerous advantages compared to the existing FC support provided by FaaS systems, such as more comprehensive control-flow constructs, data collections for efficient data distributions, and dynamic parameters. AFCL supports event-based, synchronous, and asynchronous invocation of base and compound functions. Base functions refer to computational or data processing tasks, while compound functions facilitate nesting of functions which includes if-then-else, switch, sequence, for, while, parallel, and parallelFor. The dynamic loop iteration count is important for many dynamic FCs for which iteration counts are statically unknown. Parallelism can be expressed in the form of parallel and parallelFor compound functions to unleash the elasticity of serverless functions. Data collections and their distribution to multiple successor functions are introduced to enable more fine-grain data-flow as part of an FC with good potential to reduce data transfer, thereby overcoming some of the limitations of several existing FC systems.

We implemented a prototype AFCL Environment (elaborated in detail in Chapter 2) for composing and executing AFCL FCs with a back-end for multiple FC systems. The current implementation of our environment supports back-ends for AWS Lambda and IBM Cloud Functions. We evaluated this implementation by comparing our representation for two realistic workflows from science and the public sector (an airport) that
target the IoT-Edge-Cloud continuum. We also developed FCs as a state machine with AWS Step Functions \[38\] and as IBM action with IBM composer \[39\]. Our experiments indicated substantially lower development effort by up to 62.3\% to create these FCs with AFCL, reduced makespan by up to 84.2\% and cost savings of up to 96.75\% when executing them on the two target FaaS systems (with the same AWS Lambda and IBM Cloud Functions).

### 5.2 Related work

There is a long history in the area of workflow development for industrial, scientific and business workflows. Workflow applications and their supporting platforms can be classified in two groups: task-based and dataflow-based workflows. Tasks of task-based workflows process their input data and then finish usually by producing output data. Tasks in dataflow-based workflows are persistent and maintain a state by continuously waiting for input data, process this data and then generate output data and then wait again for input data without terminating.

#### 5.2.1 Task-based workflows

Some platforms for creating workflows (e.g. FireWorks \[40\], Taverna \[41\], Kepler \[42\], or Galaxy \[43\]) require developers to explicitly compose the application as a workflow with static control- and data-flow. Although all of these frameworks support a recipe file or a GUI for composing workflows, they are closely related to the underlying workflow management systems and require developers to create workflows at a lower level of abstraction than with AFCL and lack support for serverless computing.

Several platforms (e.g. Nextflow \[44\], Swift \[45\], Parsl \[46\]), similar to AFCL, facilitate the creation of workflows (e.g. encoded in YAML or JSON) implicitly from a high-level user code (e.g. JavaScript or Python). AFCL allows the developer to create workflows either with YAML or with a Java-based AFCL API \[47\].

Other platforms facilitate programming models to build dynamic workflows at a high-level of abstraction and whose task-dependency graph is determined during runtime. For example, the Askalon \[48\] workflow management system uses AWDL \[49\] (Abstract Workflow Definition Language) to describe workflows of tasks and run them on grids and clouds. PyCOMPSs \[50\] offers Java or Python constructs to develop a task-based dynamic workflow. These methods run tasks in a VM or a container without support for event-based execution. Both AWDL and COMPS are not designed for FaaS and lack support for serverless, event-based, and asynchronous functions which are important features of FaaS.

#### 5.2.2 Dataflow-based workflows

Data-flow workflows are becoming increasingly popular for data stream processing where tasks continuously process input data and generate output data without terminating, thereby offering fine grain parallelism. Apache Flink \[51\], Samza \[52\] or Storm \[53\] are
data-flow oriented workflow platforms. Barika et al. [54] introduced an XML-based language for stream workflow application specification. These platforms can suffer from increased economic costs if tasks are running without incoming data or by inefficient scheduling of data streams to resources with high load.

5.2.3 Scientific FC systems

Some workflow management systems and their languages are adapted to compose and run FCs. Recently, Hyperflow was extended to support serverless for AWS Lambda and Google Cloud Functions [55]. A data-flow engine invokes functions whenever data inputs are available [56]. With this system, a developer could neither build AD nor GCA FC due to a lack of support to wait for both control and data flow as required by the $\log$ function in AD and GCA FCs.

SWEEP [57] is another FC system that was recently released. It allows the user to create workflows based on Python. AWS Lambda and Fargate containers are used as back-end runtime systems. Based on a DAG representation, a user can create a workflow with sequence and parallel loop (with a dynamic degree of parallelism) constructs. However, SWEEP supports DAG-based data-flow without branches, and functions are limited to a single output. Input data of functions can only be received from immediate predecessors.

GlobalFlow [58] is an FC orchestration service that coordinates multiple serverless services (sub-FCs) as an FC across multiple AWS regions. It overcomes the limit of AWS Step Functions to run an FC in a single AWS region only. Still, GlobalFlow is an FC system that can use AWS only, thereby limiting the portability, scalability and development effort. Moreover, to coordinate sub-FCs in each AWS region, GlobalFlow introduces additional functions, which increases both the makespan and economic costs.

5.2.4 Commercial FC systems

Although various existing open source FC systems and public cloud providers advanced the support for FCs, Lopez et al. [59] reported various pros and cons of several FC systems. For example, while IBM Composer has a simple FC language based on JavaScript, it is designed for short-running FCs only with limited library. AWS Step Functions offers support for parallelism, but its programmability is very limited as the FC developer has to manually code the state machine in JSON.

We evaluated important control- and data-flow constructs of FC systems as supported by well-known public providers, including AWS Step Functions ($A$), IBM Composer ($I$), Microsoft Azure Logic Apps ($M$) [60] and Google Cloud Composer ($G$) [61], as well as Fission Workflows ($F$) [62] which is an open source FC system. For each control- and data-flow construct, we evaluate these FC systems as supported (S), limited support (L), or no support (N). For control-flow support, we selected parallelism and dynamicity (control flow parameters whose values are defined at runtime, e.g. loop iteration counts). Next, we evaluated the support for data-flow comprising data distribution (scatter and
gather data among functions), multiple data inputs and outputs for functions, and DAG-based data-flow (data transfer between any pair of functions, not only between two consecutive functions). We have also evaluated for each supported feature which software skills are required. For this purpose we examined the FC systems by analyzing the provided documentation and by developing several FCs with these systems.

Table 5.1 presents the results of our study regarding control- and data-flow support. All FC systems enforce developers to hardcode the specific function implementation within the FC, which prevents the runtime system to select a specific function implementation for optimal execution (e.g., select a Python implementation of the same algorithm instead of Java or a closer AWS region Frankfurt instead of Tokyo). Moreover, FC systems lock their users and do not support FCs composed for another FC system, which limits the portability. Even worse, some FC systems (e.g. AWS Step Functions) require FCs in JSON format, while others (e.g., IBM) force the users to compose FCs in JavaScript. Most FC systems support parallelism in the form of parallel sections (A, M, and G). M and A additionally provide specific constructs to express parallelFor-Each loops but lack support for parallelFor loops. All providers except F facilitate dynamic loop iteration counts which can be used to determine the size of the loop at runtime (e.g. as an output of some predecessor function).

<table>
<thead>
<tr>
<th>Feature</th>
<th>A</th>
<th>I</th>
<th>M</th>
<th>G</th>
<th>F</th>
<th>AFCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function implementation abstraction</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>S</td>
</tr>
<tr>
<td>Portability</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>S</td>
</tr>
<tr>
<td>Parallel section</td>
<td>S</td>
<td>N</td>
<td>S</td>
<td>N</td>
<td>N</td>
<td>S</td>
</tr>
<tr>
<td>parallelFor loop</td>
<td>L</td>
<td>N</td>
<td>L</td>
<td>N</td>
<td>N</td>
<td>S</td>
</tr>
<tr>
<td>Dynamic loop iteration count</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>L</td>
<td>N</td>
<td>S</td>
</tr>
<tr>
<td>Multiple input parameters</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>Multiple output parameters</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>S</td>
</tr>
<tr>
<td>Data subset (element-index)</td>
<td>S</td>
<td>L</td>
<td>S</td>
<td>S</td>
<td>L</td>
<td>S</td>
</tr>
<tr>
<td>Data distribution</td>
<td>L</td>
<td>N</td>
<td>L</td>
<td>N</td>
<td>N</td>
<td>S</td>
</tr>
<tr>
<td>DAG-based data-flow</td>
<td>N</td>
<td>L</td>
<td>L</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
</tbody>
</table>


We observe that only F support multiple input parameters for functions, but none of the FC systems support multiple output parameters. This lack of multiple input and output parameters can result in function invocation delays and transfer of redundant data as data inputs may contain more data than needed. For instance, let us assume a function $f$ produces two different output data $d_1$ and $d_2$ where the first is needed by successor function $f_1$ and the later is needed by successor function $f_2$. If functions are limited to a single output, then both outputs have to be merged to a single output and both successor functions must receive both outputs even though only a subset is needed.
\( I \) and \( F \) are limited in extracting subsets of a given collection of data. To the best of our knowledge, these systems allow only the extraction of single data elements of a collection. Only \( A \) and \( M \) have limited support for data distribution with a workaround solution to distribute data across iterations of parallel loops. Experiences with this workaround solution show an increased programming effort and lower performance compared to AFCL, which we examine in Section 5.5. \( A \) specifies that the output of each function can be accessed from a global FC object without support for DAG-based data-flow, while \( I \) and \( M \) facilitate only direct data-flow between two consecutive functions.

Our initial analysis of control- and data-flow constructs of numerous widely known FC systems demonstrates limited portability and limited support for several constructs and none of the evaluated FC systems dominates all others in terms of language features. This chapter introduces the AFCL, which supports all evaluated constructs, thereby dominates all evaluated FC systems.

### 5.3 AFCL Specification

In this section we describe the AFCL portable approach and its constructs to compose and describe an FC at a high-level of abstraction. We developed the AFCL schema that can be used to validate FC programs, whether they are compatible with AFCL.

#### 5.3.1 AFCL portable approach with a high-level of abstraction

The left part of Fig. 5.1 shows the current state-of-the-art of all well-known FC systems, which require a separate implementation for an FC for each FC system in order to be able to run it on multiple FC systems.

Figure 5.1: AFCL high-level of abstraction and portable approach.

The main advantage of AFCL is the possibility to compose portable FCs at a high-level of abstraction, which offers two main benefits. While each well-known FC system
uses its corresponding FaaS (e.g. AWS Step Functions uses AWS Lambda to run FC functions), AFCL follows the approach to develop the FC once and then reuse the code to run on another FC system (e.g. AWS Lambda in North Virginia instead of IBM Cloud Functions in London). Moreover, the high-level of abstraction allows the runtime system to select the specific implementation of FC functions (e.g. run a function that implements the blocking matrix-matrix multiplication in Python, instead of Java implementation of Naive matrix-matrix multiplication algorithm).

Another main advantage of AFCL over the other well-known FC systems is its reach set of language constructs, which may reduce the development effort, economic costs and makespan (See Section 5.5). Among others, AFCL supports functions with multiple input and output parameters, properties, and constraints, then parallelFor loops with dynamic iteration counts, DAG-based data-flow, data collections, and data distribution constructs.

### 5.3.2 General overview of AFCL

AFCL is based on YAML [63] which is a human-readable data serialization language. YAML is also the base language for several other platforms [62, 64] that support workflow applications. An AFCL FC consists of functions, which can be either base functions or compound functions. The former refers to a single computational task without further splitting it into smaller tasks, while the latter encloses some base functions or even nested compound functions. All base and compound functions can be connected by different control- and data-flow constructs. An FC is also a compound function. In order to create an FC, all its functions (base and compound) as well as control- and data-flow connections among them, must be specified. In order to facilitate optimized execution of FCs, a user optionally can specify properties and constraints for functions and data-flow connections. In order to simplify the reading of AFCL specifications, we use meta-syntax which extends YAML, such that YAML elements can be contained in `{ }` and appended with wildcards “?” (0 or 1), “*” (0 or more), “+” (1 or more), and “|” (logical or).

### 5.3.3 Base functions

A base function represents a computational or data processing task. Fig. 5.2 presents the definition of a base function in AFCL. The name of a function serves as a unique identifier. Functions are described by type - Function Types (FTs) which are abstract descriptions of their corresponding function implementations (FIs). A FI represents an actual implementation of an FT. The FIs of the same function type are semantically equivalent, but may expose different performance or cost behavior or may be implemented with different programming languages, algorithms, etc. FTs shield implementation details from the FC developer. The selection of a specific FI for an FT is done by an underlying runtime system.
5.3 AFCL Specification

function: {
1: name: "name",
2: type: "type",
3: dataIns: [
    {
        name: "name", type: "type",
        source: "source"?, value: "value"?,
        properties: [{ name: "name", value: "value" }]+?,
        constraints: [{ name: "name", value: "value" }+]?
    }+,
4: properties: [{ name: "name", value: "value" }+]?,
5: constraints: [{ name: "name", value: "value" }+]?,
6: dataOuts: [
    {
        name: "name", type: "type", saveto: "saveto"?,
        properties: [{ name: "name", value: "value" }+]?,
        constraints: [{ name: "name", value: "value" }+]?
    }+,
]?
}?

Figure 5.2: Definition of a base function in AFCL.

Data inputs and outputs

The input and output data of a function are specified through dataIns and dataOuts ports of the function, respectively. The number and type attributes of the dataIns/Outs ports are uniquely determined by the chosen FT. A dataIns port can be specified by (i) setting its source attributed to the name of a data port of another function within the same FC (dataIns from a parent compound function or dataOuts from a predecessor function) or the name of a dataIns port of the FC; (ii) setting its source attribute to a specific URL referring to a file, or an ordered list of URLs referring to an ordered list of files; or (iii) specifying a constant or an ordered list of constants. The data associated with the dataOuts port can be stored in the location specified through its saveto attribute. Linking the data ports of different functions through the source attributes defines the data-flow of AFCL FCs. The value of a dataIns/Outs port can be used to define a constant. Every dataIns/Outs port is associated with a data type. The data types supported for AFCL are JSON datatypes [65], as well as two additional types, file and collection. The data type collection will be explained later in Section 5.3.6.

Properties and Constraints

Properties and constraints are optional attributes, which provide additional information about dataIn ports, dataOut ports, and base and compound functions, as illustrated in Fig. 5.2. Properties can be used to describe hints about the behaviour of functions, e.g. expected size of input data or memory required for execution. Constraints (e.g. finish execution time within a time limit, data distributions, fault tolerance settings) should be fulfilled by the runtime system on a best-effort basis. AFCL introduces built-in property invoke-type to specify whether a function should be invoked synchronously or asynchronously (Section 5.3.5), built-in constraint (distribution) to specify how data is gathered from or distributed among multiple functions (Section 5.3.6), and built-in constraint element-index (Section 5.3.6) to specify a subset of a data collection.
In the remainder of this chapter we omit name, type, source, value, and saveto for simplicity. Additionally, we will use the abbreviation function[{}]+ to specify a list of base or compound functions (omitting name, type, dataIns, and dataOuts).

5.3.4 Compound functions

AFCL introduces a rich set of control-flow constructs (compound functions) to simplify the specification of realistic FCs that are difficult to be composed with any current FC system without support by a skilled software developer. Compound functions contain inner functions, which can be base or compound and they are executed in the order defined by the compound function. The inner functions are called children functions of the compound function. The compound function is called the parent function of the inner functions. An inner function of a compound function $f_i$ can be another compound or base function $f_j$. The term child function of $f_i$ refers to the entire compound function $f_j$. AFCL introduces the following compound functions: sequence, if-then-else, switch, for, while, parallel, and parallelFor. The specifications for the name attribute, dataIns and dataOuts ports, along with the corresponding source and saveto attributes are similar as for a base function.

In the remainder of this text, we will not separately explain the attributes of a compound function and when we use the term function, it can refer to either base function or compound function. We present only the compound functions for, parallel, and parallelFor, while the definition of the complete set of constructs is publicly available [66].

For

The for compound function (Fig. 5.3a) executes its loopBody multiple times based on the specified loopCounter. The value of the loopCounter is initially set to the value specified by the attribute from and is then increased by the value of step until it reaches the value of to or larger. The attributes from, to, and step can be specified with a constant value or with data ports of other functions. To express dependencies across loop iterations the dataLoop ports are used (Section 5.3.6).

Parallel

The parallel compound function (Fig. 5.3b) expresses the parallel execution of a set of sections. Each section within the parallelBody represents a list of base or a compound functions, which can run in parallel with other sections. The parallel compound function can have arbitrary many data input ports, whose associated data can be distributed among inner functions, which is described in Section 5.3.6.

ParallelFor

The parallelFor compound function (Fig. 5.3c) expresses the simultaneous execution of all loop iterations. It is assumed that there are no data dependencies across loop
for: {
  1: name: "name",
  2: dataIns: [{}+],
  3: dataLoops: [
    name: "name", type: "type",
    initSource: "source",
    loopSource: "source",
    value: "constant",
  ],
  4: loopCounter: {
    name: "name", type: "type",
    from: "from",
    to: "to",
    step: "step",
  },
  5: loopBody: [{function: {}}+],
  6: dataOuts: [{}+]
}

for: {
  1: name: "name",
  2: dataIns: [{}+],
  3: parallelLoops: [
    name: "name", type: "type",
    initSource: "source",
    loopSource: "source",
    value: "constant",
  ],
  4: loopCounter: {
    name: "name", type: "type",
    from: "from",
    to: "to",
    step: "step",
  },
  5: loopBody: [{function: {}}+],
  6: dataOuts: [{}+]
}

parallelFor: {
  1: name: "name",
  2: dataIns: [{}+],
  3: parallelCounter: {
    name: "name", type: "type",
    from: "from",
    to: "to",
    step: "step",
  },
  4: parallelBody: [{function: {}}+],
  5: dataOuts: [{}+]
}

Figure 5.3: YAML representation of for, and parallel and parallelFor constructs.

5.3.5 Invocation type of a function

The invoke-type is a built-in Property defined in AFCL. As shown in Fig. 5.4a, this Property can be used to specify whether a base function should run asynchronously (ASYNC) or synchronously (SYNC). By default, if the invoke-type property is defined within a compound function, all nested base functions within that compound function will inherit this property and are invoked with the specified invoke-type. Otherwise, the base function will be invoked as specified invoke-type property. If ASYNC is specified, the FC designer must guarantee that the FC still operates correctly. Without specifying any invoke-type in any of the parent compound functions, the base functions are executed synchronously in AFCL.

(a) invoke-type property within a function.  (b) asyncHandler build-in function.

(a) invokeType property within a function.  (b) asyncHandler build-in function.

Figure 5.4: Invocation type of a function.

The build-in function asyncHandler is used to handle ASYNC invoked functions (Fig 5.4b).
This function can be used the same way as a base function is used, while the type field specifies that it is a build-in function. The build-in function has one input parameter, representing a comma separated list of names of ASYNC invoked functions (e.g. FunctionName1) and one boolean output parameter which represents whether all of these invoked functions finished. asyncHandler can be invoked with invoke-type i) ASYNC, meaning that asyncHandler immediately returns with the output parameter set to true if all functions (specified in the input parameter) finished otherwise it is set to false, or ii) SYNC, meaning that asyncHandler waits for all functions (specified in the input parameter) to finish before it returns.

5.3.6 Data-flow in AFCL

Most FC systems offer basic support to express data-flow within an FC, primarily by storing outputs of functions to a variable which can be used as input to other FC functions. AFCL supports various constructs to express more complex data-flow scenarios, which can also improve the performance of the resulting FC. The data-flow in AFCL is expressed by connecting source data ports to sink data ports of functions. A source data port can be an input data port for the entire FC, for a compound function, or for an output data port of a base function. A sink data port can be an output data port of the entire FC, an output data port of a compound function, or an input data port of a base function. When a source data port is connected to a sink data port data-flow, the data produced at the source data port will be available at the sink data port at runtime when the data is to be consumed. One source data port may have multiple sink data ports, in which case each sink data port will receive a copy of the data produced at the source data port.

AFCL allows the application developer to describe how data flows from dataOut ports of one or multiple functions into a single dataIn port of subsequent functions. Since AFCL supports nesting of compound functions, we also support data-flow from dataIn ports of a parent compound function to the dataIn ports of inner functions. For every function in AFCL, it must be guaranteed that whenever the control-flow reaches the function, all the dataIn ports of the function have been assigned well defined values. When the control-flow leaves a function that is invoked synchronously, all its dataOut ports must be well-defined, as well. Otherwise, for a function that is invoked asynchronously, the developer is responsible to synchronize data. For functions with basic control-flow, this is straightforward. In the following sections, we describe more complex data-flow scenarios.

DAG-based data-flow of conditional compound functions

AFCL allows a developer to set the source of dataIns port of a function to the dataOut of a data port of any other function. Fig. 5.5 shows such data-flow from function $f_1$ to function $f_4$. However, data-flow is challenging for conditional compound functions if-then-else and switch for which not all control-flow branches are executed at runtime. Therefore, as shown in Fig. 5.5 we have to prevent that a successor function
f4, outside of a conditional compound function, reads data from dataOuts of any inner function f2 or f3 as one of these functions may never execute. In order to prevent this illegal case, any outside successor function can only read data from dataOuts of the whole conditional compound, which will always be defined. A dataOuts port of a conditional compound has a source value with a comma separated list of dataOut ports of other functions. This entry must contain one element for each possible branch within the compound construct. In addition, if no else branch, or no default case is defined, the list must also contain a NULL element, which indicates that no data is available in that case.

Figure 5.5: Advanced data-flow support in AFCL: DAG-based Data-flow $f_1 \rightarrow f_4$; Illegal data-flow from inner functions of a compound function $f_3 \rightarrow f_4$.

Data collections

Many real world FCs may operate on datasets instead of on single data elements. Furthermore, there are numerous cases where it makes sense to collect data elements from the output of a set of functions and include them in a collection (e.g. a consumer of message queues or stream-processing tools) for further processing by subsequent functions. Collections are also well suited to exploit data parallelism by distributing collection data elements to loop iterations or parallel sections. Collections may contain a static or dynamic number (unknown at the time when an FC is composed but not yet executed) of data elements. In order to support this feature as part of AFCL, we introduce the concept of a data collection. The elements of a data collection are of JSON datatypes and they can be distributed onto base and compound functions. The data port with type collection represents a list of data elements provided by the user as the initial input of an FC or produced by FC functions as an intermediate result.

Subsets of data collections can be specified by using the build-in constraint (explained in Section 5.3.3) element-index. With index (Fig. 5.6a), the developer can specify certain positions of the data collection. The value of element-index is a list of comma separated expressions. Note that in the absence of the type element-index, the entire data collection is specified.
The following grammar specifies the syntax of the construct `element-index`, where \( e \) denotes the element index, \( c \) a colon expression, \( s_1 \) the start index, \( s_2 \) the end index, and \( s_3 \) a stride.

\[
e ::= c[ ], c[ ]^* \\
c ::= s_1[ : s_2[ : s_3] ]
\] (5.1)

Such an expression can refer to either a specific index or a range of indexes with an optional stride. As an example consider Fig. 5.6b which selects the elements at positions 1, 2, 4, and 6 of a collection.

**Data distribution**

Most existing FC systems mainly replicate collected data to multiple functions or loop iterations without supporting the concept of data distributions. This inefficient data transfer between functions may initiate a considerable delay in function invocation time, as shown in Section 5.5.3.

Unlike related work, AFCL offers an additional, build-in, `distribution constraint` (explained in Section 5.3.3), which allows the developer to specify how data elements of a collection will be distributed across successor functions. AFCL introduces `block-based` data distribution and data replication.

**Block-based data distribution** Using the built-in constraint `distribution` and specifying the `block-based` data distribution (Fig. 5.7a), optionally in combination with `size` and `overlap`, a data collection is partitioned in contiguous blocks of a specific length and distributed to the different successor functions or different loop iterations (as shown in Fig. 5.7b). As a comparison, some existing FC systems support distribution of a single element of a collection to a loop iteration (e.g. AWS Step Functions with the MAP construct).
5.3 AFCL Specification

Data replication Another option is to replicate a certain dataOuts port and then distribute to the different successor functions. AFCL allows data replication by specifying the constraint distribution in combination with REPLICATE, as well as the number of replications (times), as shown in Fig. 5.8a. The elements of a data collection will be replicated a specific number of times to the successor functions or to different loop iterations (as shown in Fig. 5.8b).

The dataIn and dataOut ports of each construct presented in this chapter can be of type collection in order to collect the data produced by a loop iteration, a parallel section or any other function. After all loop iterations finished, all dataOuts are written into the dataOuts of the parallelFor, parallel, while and for construct, which is of type collection and can be accessed by subsequent functions. If both, element-index and distribution are specified within the same data port, element-index has higher precedence.

DataLoop ports

Every for and while loop can optionally use DataLoop ports (used in Fig. 5.3a) to represent inputs to functions specified in a loop body. These ports get their initial value from the optional initSource field or a constant value from the value field. A loopSource field specifies a data-flow from the output of a function of the loop body.
which can be used as input to functions executed in the next loop iteration. **name** is an unique identifier of a DataLoop port and **type** specifies the data type of the value.

### 5.3.7 Event-based invocation

Events in AFCL are specified in a separate YAML file. By keeping events in a separate file, a user can execute the same FC based on different events or multiple FCs based on the same event. Fig. [5.9a](#) shows the definition of events in AFCL. The **start** and **end** fields represent the period of time when the event is active, which means that the FC will be invoked only if an active event happens. If **start** date is not specified, the event is active immediately, while the event is active until it is removed if the **end** is not specified. The **type** of an event could be **ONCE**, **PERIODIC** (run an FC periodically after a specific period of time), or external events, such as **STREAM_DATA** (run the FC for every data item coming out of a data stream), **NEW_FILE** (run the FC whenever a new file is added to a storage e.g. S3 bucket) or **NEW_DATABASE_ENTRY** (run the FC whenever there is a new entry in a specified database). The **value** is represented in each case as a string, which expresses e.g. a **cron** for the **PERIODIC** invocation. An example of such **PERIODIC** invocation is shown in Fig. [5.9b](#). Starting from the 1st of February 2020, the FC is invoked every hour until the 3rd of February, 2020.

```yaml
---
events: [  
{
  start: "01-02-2020 00:00:00",
  end: "03-02-2020 00:00:00",
  type: "PERIODIC",
  value: "0 * * * *"
}
---
```

(a) **events**

```
---
events: [  
{
  start: "01-02-2020 00:00:00",
  end: "03-02-2020 00:00:00",
  type: "PERIODIC",
  value: "0 * * * *"
}
]
```

(b) Example of a **PERIODIC** invocation

Figure 5.9: YAML representation of events and example of **PERIODIC** invocation.

### 5.4 Composing FCs with different FC systems

This section describes two realistic FCs which we composed with AFCL (Section 5.4.1), AWS Step Functions, and IBM Composer (Section 5.4.2), as well as their complexity (Section 5.2). AWS Step Functions (AWS_Step) and IBM Composer (IBM_Comp) FC systems are most advanced among most common FC systems according to our evaluation in Section 5.2. These compositions will be used for our experimental evaluation in Section 5.5. The main reason why we decided to evaluate AFCL with AWS_Step and IBM_Comp is because AWS_Step has advanced mostly from public public providers and can be assumed as base platforms for research on serverless [67] and both FC systems are two mostly referenced serverless technologies in academic publishing [68]. Additionally, Lopez et al. [59] reported that IBM_Comp results in smaller overhead (the overall makespan minus the execution time of the FC) for sequential loops, while AWS_Step generates lower overhead when invoking concurrent functions.
We carefully selected two realistic FCs with different characteristics, including degree of parallelism (number of functions that can be executed in parallel), complex and dynamic control-flow, and complex data-flow as described in the following sections. The Gate Change Alert FC (GCA) is realistic FCs as used in industry. Since duration of its functions is within hundreds of milliseconds, which is within the range of performance instability, all functions are emulated and have been programmed with a timer operator to sleep for a pre-defined duration, similar as performed by several researchers [59, 30]. This implementation with a timer operator to sleep reduces the noise in function duration [69, 70], which helps for better evaluation because the differences in the overall makespan will be mainly due to the composition constructs (data- and control-flow,) rather than the deviance in the duration of the same function. The second FC – the Genome1000 FC (GEN) – is an open-source scientific application. To generalize our findings, unlike for the GCA FC, we used the original implementation of the GEN FC since the duration of its functions is within seconds. In the following sections we describe both FCs and how they have been composed with AFCL. All original FC representations can be found at the AFCL web page [66].

5.4.1 FC applications composed with AFCL

The graphical representation of the corresponding AFCL workflows are shown in Fig. 5.10. Black arrows are control-flow edges and blue arrows with small squares are data-flow edges. Control-flow and data-flow edges may overlap. For simplicity reasons, multiple data-flow edges between any pair of functions are represented with a single data-flow edge.

GCA: Gate Change Alert FC

The GCA FC is a public space management application that performs a series of actions after a gate of a specific flight has changed at an airport. Fig. 5.10a shows the graphical representation of the GCA FC composed in AFCL. After the GCA FC is invoked, it reads the information about the flight and the new gate (function getFlight) and then loads all available passenger data of that flight with the function selectPassenger. Thereafter, for every passenger from that flight that is already at the airport, a set of additional functions are invoked. Function informPassenger informs the passenger about the new gate, while calculateTimeToGate is a complex function that determines the location of a passenger and estimates the time needed to the new gate. If this waiting time is below a threshold, then the GCA FC will execute the recommendShop function. Otherwise, the passenger is notified through informCriticalTime to proceed to the new gate. Once all passengers are informed, function log logs the status for all passengers for further data analysis.

The GCA FC has a complex data-flow, for which there is limited support by the related work. There is a DAG-based data transfer from each instance of calculateTimeToGate to log thus the latter function can log the time to reach a certain gate for every passenger for further data analysis. Existing approaches (e.g. IBM Comp) would
move data from \texttt{calculateTimeToGate} to \texttt{log} through all functions in between, which may potentially raise security and performance issues, or require additional development effort. The most complex data-flow is the distribution of the passenger data from the function \texttt{selectPassenger} to the specific functions \texttt{informPassenger} and \texttt{calculateTimeToGate} within the \texttt{parallelFor} compound function. Existing approaches would replicate data to each section (e.g. the parallel construct of AWS Step) or distribute a single element to each function (e.g. \texttt{MAP} construct of AWS Step), whereas our approach can limit distribution of data to the minimum data required for every loop iteration or group multiple data to a single iteration. We thus can reduce data transfer (avoid data replication) and invocation time (send smaller amount of data). Furthermore, function \texttt{selectPassenger} has two outputs, the number of passengers and the ID of each passenger. In contrast to AFCL, existing approaches merge all outputs into a
single output which can lead to redundant data transfer if only part of the output data is needed. The GCA FC has two potentials of parallelism. Firstly, informPassenger, calculateTimeToGate and recommendShop can be executed simultaneously for different passengers. Additionally, informPassenger and calculateTimeToGate can be invoked simultaneously for the same passenger. Another important characteristic of GCA is the dynamic number of passengers \( n \) (output of the function selectPassenger) that determines the number of iterations for the parallelFor construct and which is only known once GCA runs. This dynamic behavior of the GCA FC makes the composition impossible in any DAG-oriented workflow language (e.g. Pegasus DAX - Directed Acyclic graph in XML [71]). AFCL supports dynamic parameters which are defined at runtime, whereas, to the best of our knowledge, dynamic parallelFor loops are not supported by related work.

The Genome1000 FC

The Genome FC (GEN) is a scientific FC, which identifies mutational overlaps using data from the 1000 genomes project\(^1\) in order to provide a null distribution for rigorous statistical evaluation of potential disease-related mutations. Among others, the GEN FC cross-matches which person has which mutations and determines the mutation’s sift score.

Fig. 5.10b shows the graphical representation of the GEN FC composed in AFCL. Each instance of the function Individual fetches and parses single nucleotide polymorphism (SNPs) variants in a chromosome and determines which individuals contain these chromosome variants. The function Individuals_merge merges all outputs of Individual, while Sifting computes the mutation for all SNP variants (the SIFT scores). Next, Mutation_overlap measures SNP variants (the overlap in mutations), while Frequency measures the frequency of mutational overlaps. For every of the six super populations (African, Mixed American, East Asian, European, Great Britain and South Asian) as well as for all populations, a separate instance of the last two functions is invoked.

The GEN FC has a high potential of parallelism in three sections without conditional branches. One section (individuals) also uses a dynamic parallel loop iteration count. Every function of the other two sections (Mutation_overlap and Frequency) has two separate data inputs (outputs from Individuals_merge and Sifting). Related work offers limited support to express the GEN FC. A workaround solution is to combine both outputs in a single input, which will increase the invocation time of functions and could reach the FaaS provider limit for the size of the input faster. The data transfer is realized through file (accessed by reference) transfers between functions.

The GEN FC has dynamic degree of parallelism \( k \) (specified while composing the FC) based on the number of individuals that are sent to each function Individual, and two static parallelFor loops with 7 iterations, one for every super population and one for all populations.

\(^1\)https://github.com/pegasus-isi/1000genome-workflow
5.4.2 Composing FCs with AWS_Step and IBM_Comp

AWS_Step offers a JSON representation of the FC. We developed one state machine for the GCA FC, while for the GEN FC we were able to create two state machines, a dynamic one (AWS_Step_DYN) using MAP and a static one (AWS_Step_STAT) using parallel to parallelize multiple instances of individuals, mutationOverlap, and frequency functions. AWS_Step_STAT can achieve better performance by simultaneously running the functions of the type parallel without loop, but it also requires considerable development effort to scale the composition of the GEN FC because each function and the state within parallel must be manually created. AWS_Step_DYN requires less effort by the developer in order to change the degree of parallelism k.

We built the GCA FC with Javascript and IBM_Composer derived the FC files in JSON format. Since IBM_Composer does not support a parallel loop, we developed the loop for each passenger as a serial while-loop whose body is a sequence of two functions (InformPassenger and CalculateTimeToGate), as well as an if-condition. Additional development effort was needed to create a workaround solution, a new component "type": "let" after function SelectPassenger in order to transfer data for all passengers to the last function log. As IBM_Composer does not support parallelism, we did not composed the GEN FC with IBM_Composer.

5.5 Evaluation

This section evaluates AFCL and its environment and compares them with two state-of-the-art FC systems including AWS North Virginia and IBM London. We composed the two FCs (GCA and GEN) with AFCL and then run them with IBM Cloud Functions and AWS Lambda. We then composed the FCs and run them with IBM Composer (IBM_Comp) and AWS Step Functions (AWS_Step) in the same regions. We evaluated the development effort, makespan and economic costs of the AFCL Environment with various problem sizes of GCA and GEN FCs.

5.5.1 Testing methodology

Experiments

Since the GCA FC has a dynamic parameter n that determines the parallel loop iteration count, we created several instances of the GCA FC by varying the number of loop iterations (output of the function selectPassenger). Based on this experimental setup we examined the scaling behavior of the AFCL Environment and compared it against the current implementation for AWS_Step and IBM_Comp by using their native FC API. Due to a limitation of IBM Cloud Functions to execute a maximum of 50 function invocations (SequenceMaxActions), we were able to run up to n = 15 loop iterations. AWS Lambda allows a much higher limit of up to 1,000 concurrent function invocations.


https://docs.aws.amazon.com/lambda/latest/dg/limits.html
and we varied up to the problem size of 200. Every function of the GCA FC is invoked with a minimum possible memory of 128 MB. Similarly, we defined several experiments for the GEN FC by varying the number of individuals \( k \) (scaling compute resources), both for static and dynamic compositions, up to \( k = 200 \).

We also examined the development effort by measuring the time in terms of minutes to develop each FC for every different FC system. For the static composition of the GEN FC (AWS\_Step\_STAT), a developer must create a separate composition for each problem size \( k \) (number of individuals) by manually specifying the functions within a parallel section. For this reason, we also evaluated the development effort to create AWS\_Step\_STAT of the GEN FC with AWS\_Step for various problem sizes \( k \).

**Test plan for a fair evaluation**

We created a detailed test plan for a fair comparison that mitigates the impact of the cloud performance variability when running complex workflow applications [69, 72]. We fixed the control-flow path of the GCA FC (the "then" branch of if-then-else) to run the sub-FC comprising all functions except informTimeCritical.

We repeated the execution of each experiment 6 times and considered the average value for the overall makespan by omitting the first execution (cold start). For evaluating economic costs, we neglected the monthly free tiers of AWS Lambda and IBM Cloud Functions, in order to study scenarios when costs are charged by these systems. The economic costs for IBM\_Comp are derived based on the runtime and the amount of memory used during function execution. AWS\_Step charges three types of costs: number of invoked functions, their compute resources used at runtime for each invocation, and the number of state transitions. We omitted costs for the additional services (e.g. S3 storage) for the GEN FC as they are identical for all FC systems (e.g. AWS\_Step and AFCL\_AWS use the same files in S3).

In order to provide a fair comparison (different execution time or failure rate), we composed all FC implementations to use the same function implementations, although we could not use the full benefit of AFCL. For the GCA FC, this required that each atomic function could not have a collection as an input, which means that we used only BLOCK(1) for the GCA FC and counter of the parallel loop for the GEN FC.

Although we used the same function implementations for the corresponding FC implementations, still we used the execution time of each lambda function of AFCL experiments also for AWS\_Step and IBM\_Comp to calculate the economic costs. This simplification can be justified also with the pricing schemes of the providers for the duration of used resources, i.e., usage time is rounded to the closest 100ms.

**5.5.2 Development effort evaluation**

We examined the application development effort for AFCL, AWS\_Step and IBM\_Comp. One computer science master student learned the workflow language of every FC system and then composed each FC with every FC system.
Development effort for the GCA FC

The left column of Table 5.2 shows the development effort for each used construct of the GCA FC. It took only approximately 85 minutes to compose the GCA FC with AFCL, which is 12.37\% better than 97 minutes with AWS Step, and even 23.42\% better than 111 minutes with IBM Com.

<table>
<thead>
<tr>
<th>FC construct</th>
<th>GCA</th>
<th>GEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function definitions</td>
<td>7·3</td>
<td>5·3</td>
</tr>
<tr>
<td>Sequence</td>
<td>4·1</td>
<td>4·1</td>
</tr>
<tr>
<td>If-then-else</td>
<td>1·10</td>
<td>1·10</td>
</tr>
<tr>
<td>Parallel section</td>
<td>1·5</td>
<td>2·5</td>
</tr>
<tr>
<td>ParallelFor loop</td>
<td>1·5</td>
<td>3·5</td>
</tr>
<tr>
<td>Dynamic loop iterations</td>
<td>1·5</td>
<td>5·5</td>
</tr>
<tr>
<td>Multiple output param.</td>
<td>5·1</td>
<td>—</td>
</tr>
<tr>
<td>Multiple input param.</td>
<td>5·1</td>
<td>10·1</td>
</tr>
<tr>
<td>Data distribution</td>
<td>1·5</td>
<td>1·5</td>
</tr>
<tr>
<td>DAG-based data-flow</td>
<td>1·20</td>
<td>5·5</td>
</tr>
<tr>
<td>Input preparation</td>
<td>—</td>
<td>5</td>
</tr>
<tr>
<td>Total</td>
<td>85</td>
<td>69</td>
</tr>
</tbody>
</table>

Table 5.2: Development effort for both FCs with various FC systems. AF: AFCL, A: AWS Step Functions, I: IBM Composer, ADk: AWS Step_DYN with k individuals, ASk: AWS Step_STAT with k individuals, X: not supported, —: not used in the workflow. All measurements are specified in minutes as a product of how many times the developer needed to develop the FC construct and the time to develop that FC construct.

AFCL required more time (3 minutes per function) to define the seven FC functions (due to its ability to support multiple input and output data ports), followed by 2 minutes per function for AWS Step and one minute for IBM Comp (less configurable parameters). The developer needs more time (2 minutes per sequence) in AWS Step due to the states compared to other FC languages.

IBM Comp does not support any parallelism, therefore a sequential while loop has been used with a dynamic iteration count expressed with the LET construct. This implementation took approximately 30 minutes. AWS Step supports parallel loops using the MAP construct and AFCL provides the ParallelFor construct, which took 5 minutes for each, 5 more minutes each for nesting the parallel construct inside the parallel loop, and yet additional 5 minutes for setting the dynamic loop iteration.

AFCL offers a very simple way to specify the multiple inputs and outputs for each function, including both atomic and compound functions, especially for those that are nested in the parallel loop. The developer needed 1 minute to specify each multiple input parameters including the inputs to the parallel loop and the nested compound functions (total five). The same time of 5·1 minute was needed for specifying the multiple outputs of the selectPassenger, as well as the outputs of the nested functions inside the parallel loop. Since neither of AWS Step and IBM Comp supports multiple inputs and outputs,
the developer had to use a workaround solution with the states for the AWS_Step and combine multiple outputs into a single output object and then parse it for functions that use a part of it as an input.

In terms of data distribution, AFCL allows the developer to use constructs like `BLOCK` to distribute data between different loop iterations. Since `AWS_Step` allows the distribution of only one element to each loop iteration, the same development effort is needed as for AFCL for the GCA FC only. For IBM_Comp it took approximately 10 minutes to adapt the functions and make the collection accessible to the successive iterations.

The `LET` statement of IBM_Comp has been used to provide the data for the `log` function (DAG-based data-flow), which took approximately 30 minutes with IBM_Comp, 20 minutes with AFCL (simpler with data collection and DAG-based data-flow) and 15 minutes with `AWS_Step` (specify data items from the state). The developer does not need to prepare inputs for the GCA FC as the all data inputs are values in the required JSON format.

We observe that a developer needs more effort to compose (i) FCs that has many functions with AFCL, (ii) FCs comprise a long sequence and many input and output parameters with `AWS_Step`, and (iii) FCs that consist of dynamic loop iterations and complex data-flow (data distribution and DAG-based data-flow) with IBM_Comp.

### Development effort for the GEN FC

The right column of Table 5.2 presents details for development effort to compose the GEN FC with AFCL, `AWS_Step_STAT`, and `AWS_Step_DYN` for problem size $k$ in general because it is known at the composition time of the GEN FC. The development time for functions, sequences, the two parallel sections, the two parallel loops, and the dynamic loop iterations followed the same distribution as for the GCA FC. However, `AWS_Step_STAT` needs additional 2 minutes per function to manually develop the $k$ functions in the parallel section for `individuals` compared to AFCL and `AWS_Step_DYN`, which present parallelization of `individuals`, `mutationsOverlap`, and `frequency` with three parallel loops. The GEN FC with AFCL and `AWS_Step_DYN` required a fixed development effort (two parallel loops, i.e., FCs are built once and invoked with different input data (problem size $k$).

While all data inputs for the GCA FC were an output of another function, they were already in the required JSON format, thereby no additional preparation was needed. However, the input data for the GEN FC was a file with 10,000 inputs, which needs to be prepared, i.e., to be split to multiple chunks, upload them to S3 and assign a reference to each chunk. Due to the limited data distribution of `MAP` in `AWS_Step`, which can distribute only one element per a loop iteration, additional development effort is needed to distribute data among `individuals` from the states. Firstly, both implementations of the GEN FC with `AWS_Step` required 10 minutes to develop the data distribution. Secondly, a fixed time of 5 minutes was needed to create references for the input files and upload them to S3. Thirdly, in the state machine, an additional 0.5 minute was spent per function to prepare the input files for each function `individuals`. In AFCL, the input preparation is fixed to 5 minutes, since only one input parameter is changed.
to run a different number of individuals. Using the counter of the parallel loop for data
distribution, no additional development effort is needed to prepare the input for the FC
in AFCL. Nevertheless, the functions of the GEN FC are not implemented to work with
collections of files to be consistent with the AWS_Step implementations.

We observe that the development effort for composing the GEN FC with AFCL is
affected by the total number of function definitions, while the implementations with
AWS_Step are additionally affected by the problem size $k$.

Fig. 5.11 shows the development effort to compose the GEN FC with AFCL, AWS_Step
STAT, and AWS_Step_DYN for various problem size $k$. We observe that the develop-
ment effort for the static model created with AWS_Step depends highly on the problem
size. For small problem size ($k \leq 5$), the developer required the smallest amount of time
due to a simpler state machine (lower development time for input preparation and par-
allel sections). However, the development effort worsens significantly for higher values
of $k \geq 10$, where significantly more time is needed to manually create parallel section
with size $k$ and prepare the input data to distribute to each instance of individuals.
More precisely, AFCL requires less development effort for the problem size $k = 200$ for
up to 62.3% and 87.74% compared to AWS_Step_DYN and AWS_Step_STAT, respec-
tively. The negligible trade-off is the region of lower problem size, where AFCL requires
6.15% more development effort than AWS_Step_STAT, while still keeping the advantage
of 17.37% compared to AWS_Step_DYN.

![Figure 5.11: Evaluated development effort to create GEN FC with AFCL and with
AWS_Step (dynamic sequential and static parallel) for various problem sizes $k$.]

### 5.5.3 Makespan evaluation

In this section we present the results of the evaluation for the makespan for both FCs.

**Makespan for the GCA FC**

The full potential of AFCL can be observed for the makespan of GCA FC, which is
presented in Fig. 5.12. AFCL IBM reduces the makespan compared to IBM Comp
starting from 7.34% for the smallest problem size (number of passengers) of \( n = 1 \) (GCAI without parallel loop) up to 69.17% makespan reduction for problem size \( n = 15 \). This significant improvement is a result of the potential of AFCL\textsubscript{IBM} to invoke multiple instances of the `parallelFor` and multiple functions in the `parallel` compound function, supported by our data distribution among the inner functions. AFCL\textsubscript{AWS} runs the entire GCA FC from 21.48% \(( n = 1 )\) up to impressive 84.2\% \(( n = 15 )\) faster than AWS\textsubscript{Step}. Despite the parallelism support through the MAP construct of AWS\textsubscript{Step}, AFCL\textsubscript{AWS} achieved a higher speedup than AWS\textsubscript{Step} compared to AFCL\textsubscript{IBM} vs IBM\textsubscript{Comp}. This is caused because of the state machine approach of AWS\textsubscript{Step} whereas IBM\textsubscript{Comp} invokes additional functions for data transfers between functions of an FC.

Even more, AFCL\textsubscript{IBM} achieved lower makespan than AWS\textsubscript{Step} for \( n = 10 \) and \( n = 15 \).

Also for higher problem sizes \( n = 100 \) and \( n = 200 \) which is only supported by AWS Lambda, AFCL\textsubscript{AWS} substantially reduced the makespan by approximately 98\% compared to AWS\textsubscript{Step}, which was mainly caused by the delayed function invocation in the MAP construct, while the bias in function duration was negligible.

AFCL\textsubscript{AWS} outperforms AWS\textsubscript{Step} in terms of efficiency, as well. While the efficiency of AFCL\textsubscript{AWS} is retained to 90\% for \( n = 15 \) and downgraded to 29.9\% for \( n = 200 \), AWS\textsubscript{Step}'s efficiency is only 18.07\% for \( n = 15 \) and only 0.58\% for problem size \( n = 200 \). Another interesting observation is that AFCL\textsubscript{AWS} achieved lower makespan for \( n = 200 \) than AWS\textsubscript{Step} for \( n = 10 \). Similarly, IBM\textsubscript{Comp} achieved efficiency of only 13.07\% for problem size \( n = 15 \), while AFCL\textsubscript{IBM} three times higher, or 39.27\%.

**Makespan for GEN FC**

Fig. 5.13 shows the makespan results for GEN FC created based on results for AWS\textsubscript{Step}_DYN, AWS\textsubscript{Step}_STAT and AFCL\textsubscript{AWS}. We observe that AFCL\textsubscript{AWS} outperforms
AWS·Step·DYN with respect to makespan for every problem size $k$, starting from 0.66\% for $k = 1$ up to 22.34\% for $k = 200$. AFCL·AWS is also faster than AWS·Step·STAT for every problem size $k > 1$, since the parallel section of AWS·Step·STAT does not generate overhead compared to a loop with a size $k = 1$ for AFCL·AWS.

![Figure 5.13: Average makespan of the GEN FC implemented with the dynamic (AWS·Step·DYN) and static compositions (AWS·Step·STAT) on AWS Step Functions, as well as with the AFCL Environment (AFCL·AWS) for different $k$ (number of instances of function individuals).](image)

The advantage of efficient data distribution for a parallel loop can be observed for large problem sizes ($k \geq 100$) where AFCL·AWS achieved up to 34.15\% lower makespan than AWS·Step·STAT. For these high values of $k$, the inefficient replication of data to the inner functions of a parallel section and collecting their outputs under AWS·Step·STAT resulted in a higher makespan than for AWS·Step·DYN, which does not replicate, but distributes data in the same way as AFCL·AWS.

Although all three implementations can exploit parallelism, the speedup is limited due to the synchronisation function IndividualsMerge. We observe that all three implementations achieved maximal speedup of 1.5 up to $k = 20$, but only AFCL·AWS retained the speedup of 1.17 for problem size $k = 200$, while AWS·Step·DYN and AWS·Step·STAT achieved slowdown (speedup of 0.92 and even 0.77, respectively). This shows the benefit of data distribution for higher problem size. While AFCL·AWS distributes the data set among the inner functions of the parallelFor by sending only the necessary data (BLOCK), AWS·Step·DYN distributes only element by element (BLOCK(1)), while AWS·Step·STAT does not distribute, but replicates the complete data set to each individuals (sends all data inputs to every function individuals).

### 5.5.4 Economic cost evaluation

Cost and performance are often conflicting criteria for workflows in cloud systems. However, our evaluation showed that these two criteria are not necessarily conflicting for all evaluated FC systems.
5.5 Evaluation

Costs for GCA FC

Fig. 5.14 also displays the estimated costs for the GCA FC for the same experiments of Fig. 5.12. Since AFCL IBM outperformed IBM Comp in terms of makespan, it generates lower costs, as well, starting from 5.26% (for \( n = 1 \)) up to 36.95% (for \( n = 15 \)). AFCL AWS could reduce the costs for GCA FC by remarkable 96 − 97% compared to AWS Step. Even more, AFCL AWS runs GCA200 cheaper (almost twice) than AWS Step runs GCA10.

![Figure 5.14: Economic costs (logarithmic scale) for the corresponding experiments presented in Fig. 5.12.](image)

Costs for GEN FC

Fig. 5.15 displays the results for the estimated economic costs of the experiments presented in Fig. 5.13 including GEN FC implemented with AWS Step DYN, AWS Step STAT, and AFCL AWS. Since AWS Step STAT performs two state transitions less than AWS Step DYN, it generates slightly lower costs than AWS Step DYN. On the other side, running the GEN FC in AFCL AWS environment generated the lowest costs. More precisely, AFCL AWS is cheaper than AWS Step DYN by 1.40% for \( k = 1 \) up to by 12.07% for \( k = 200 \).

AFCL AWS runs the GEN FC cheaper than both compositions in AWS Step because the cost for the duration of the INVOKER function is lower than the costs for state transitions that both compositions AWS Step perform. However, although AFCL AWS is the cheapest FC system for the GEN FC, still the costs savings of AFCL AWS are much lower than for the GCA FC. The reason is because the duration of the INVOKER function (AFCL AWS) is much longer (164.5s) for the GEN FC compared to 6.44s for the GCA FC, thereby the costs for INVOKER duration are not negligible compared to costs for state transitions in AWS Step.

5.5.5 Discussion

The main advantage of AFCL over AWS Step are its constructs for allowing multiple
5 Portable Serverless Workflow Applications at a High-level of Abstraction

Figure 5.15: Economic costs for the corresponding experiments presented in Fig. 5.13.

input and output data of each function and specify data-flow between functions directly, rather than through the states. Specifying data-flow between specific functions or distributing among functions of parallel loops or sections reduces the makespan of AFCL FCs compared to the AWS Step FCs. AWS Step is more expensive than AFCL due to the charges for state transitions, compared to the overhead of AFCL - the INVOKER function that runs during the FC execution. This is more emphasized in the GCA FC, where the parallel loop nests multiple functions, each of which creates a new state transition in each loop iteration.

IBM Comp generates lower LoC compared to AFCL, which makes it a preferable system for simple FCs. However, developers are challenged to use their software skills and effort to manage FCs with complex data-flow (DAG-based data-flow, dynamic loop iterations, and multiple input and output parameters). The economic costs of AFCL and IBM Comp are similar as IBM Comp charges the user for the used resources by the FC functions only. Although IBM Comp invokes a function to transfer data between each two consecutive functions, still the overhead of such ephemeral functions is negligible because they run in a very short time using small amount of resources. The main disadvantage of IBM Comp is that it can run FCs sequentially only. Even if IBM Comp runs FCs using parallel loops or sections, still, it limits the users to 50 concurrent functions only. On the other side, AFCL utilizes IBM Cloud Functions, whose limit is 1,000 concurrent functions.

5.5.6 Threats to validity

The rich set of AFCL constructs with many features reduced the development effort for both complex FCs evaluated in Table 5.2. However, for simple FCs, whose control-flow mainly follows data-flow, or FCs with many functions, application developers need more time to define each function and specify the DAG-based data-flow in AFCL, compared to AWS Step and IBM Comp. The former’s approach with ”states” simply requires to store to and read data from states, while the latter relies on the underlying runtime system for this. Nevertheless, the AFCL development effort overhead will be negligible.
for simple FCs.

AFCL achieved significantly lower cost, makespan and development effort than with AWS Step and IBM Comp for both evaluated FCs. AFCL compared to AWS Step achieved up to 97% cost saving for larger problem sizes. The considerable cost savings for the GCA FC is achieved due to short makespan and a high degree of parallelism, which reduces the costs for the INVOKER function to a negligible value compared to the costs for state transitions in AWS Step. However, for longer running FCs, such as the GEN FC, the benefit of AFCL is only 12%. According to the current cost model of AWS Step Functions, costs of every state transition is equal to the costs for 12 seconds of AFCL INVOKER function execution which uses 128MB memory. This means that FCs with a makespan shorter than the product 12 · m, where m is the number of state transitions, will benefit by using AFCL. On the other side, FCs with low degree of parallelism and long running functions will not benefit in terms of cost when they are executed with the AFCL Environment, but only in terms of development effort and makespan. In general, long-running FCs are not suitable for serverless in terms of costs. They are better suited for VMs (e.g. EC2) or containers (e.g. ECS). On the other side, event-based short-running FCs, which are very well suited for serverless computing, should be composed with AFCL to benefit by reduced costs, makespan and development effort.

5.6 Summary

AFCL addresses the need for FC systems to support programming at a high-level of abstraction following the “develop the FC once and reuse it later” approach, rather than to develop a separate implementation for each FC system. Moreover, AFCL offers more sophisticated constructs for expressing control- and data-flow which are not sufficiently supported by existing FC systems.

Our prototype AFCL Environment overcomes portability limitations and vendor lock-in by supporting multiple backends (currently AWS Lambda and IBM Cloud Functions). Experiments with two realistic FCs demonstrated that with AFCL, a developer can compose FCs with up to 62.3% less development effort than with AWS Step Functions and IBM Composer. This reduction in development effort is mainly due to AFCL’s constructs for sophisticated data distribution, to reuse the inputs and outputs as variables for other constructs, and specify data-flow between functions directly, rather than through the states in AWS Step Functions or using workaround solutions with LET in IBM Composer. Furthermore, our performance studies highlighted that all evaluated FC implementations can be executed significantly cheaper by up to 96.75% and substantially faster up to 84.2% than with the corresponding FC systems (AWS Step Functions and IBM Composer). AFCL runs any FC cheaper than IBM Composer and executes all FCs with complex control-flow and many short running functions cheaper than AWS Step Functions. FCs developed with AFCL run always faster than the equivalent implementations in AWS Step Functions and IBM Composer.
6 Conclusion and Future work

This master thesis introduces the AFCL Environment, a platform for development, scalable and portable execution of function choreographies across all well-known FaaS systems. It comprises several modules which facilitate development of functions and offers development of portable FCs at a high-level of abstraction.

The Dependency Aware FaaSifier advances state-of-the-art by automatically converting and deploying an existing monolithic application to well-known public cloud providers. Current tools have limited support to deploy existing monolithic applications with code and package dependencies to serverless functions. DAF simplifies the development and within the evaluation we managed to fasten an existing application by a factor of 3.64.

jFaaS introduces a way to easily build portable and scalable serverless applications. The tool achieved a throughput of 0.39 ”no-op” functions per second on two state-of-the-art providers, which is 2.5 times better than the throughput on a single provider.

Additionally, the thesis introduced the Abstract Function Choreography Language, a high-level programming support to construct abstract function choreographies. Experiments with two realistic function choreographies showed that the development effort is up to 62.3% less than with current function choreography systems. The evaluated choreographies are executed up to 96.75% cheaper and up to 84.2% faster.

There are several possible extensions and future work is already planned. One possible extension is the Dependency Aware FaaSifier in order to support other programming languages. Additionally, a way to consider provider specific constraints like writing to a file system would be beneficial for the tool. JFaaS could be extended to support additional FaaS providers and different resource types. Finally, AFCL could be extended with other compounds like n-out-of-m or discriminators.
Bibliography


Bibliography


