Future Generation Computer Systems 114 (2021) 368–382

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

# AFCL: An Abstract Function Choreography Language for serverless workflow specification

Sasko Ristov *, Stefan Pedratscher, Thomas Fahringer

*University of Innsbruck, Technikerstraße 21a, A-6020 Innsbruck, Austria*

ARTICLE INFO

*Article history:*
Received 29 January 2020
Received in revised form 29 June 2020
Accepted 10 August 2020
Available online 18 August 2020

*Keywords:*
AWS Step Functions
Cost
FaaS
IBM Composer
Performance

ABSTRACT

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 iterations 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 behavior 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.



## 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 [1]. 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 [2]. Our initial study revealed three important observations that motivated this paper. 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 [3]. Secondly, even if workaround solutions exist, they may degrade performance (e.g. delayed invocation of a function) [4]. Finally, none of the evaluated FaaS systems appears 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

* Corresponding author.
*E-mail address:* sashko@dps.uibk.ac.at (S. Ristov).

https://doi.org/10.1016/j.future.2020.08.012

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`. Dynamic loop iteration count are important for many dynamic FCs for which iteration counts are statically unknown. Parallelism can be expressed in 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 for composing and executing AFCL FCs with a backend for multiple FC systems. The current implementation of our environment supports backends 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 [5] and as IBM action with IBM composer [6]. 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 costs savings by up to 96.75% when executing them on the two target FaaS systems (with the same AWS Lambda and IBM Cloud Functions).

This manuscript is organized in several sections. Section 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 3. We have carefully selected a diverse set of realistic FCs and built them in Section 4 in order to evaluate AFCL (Section 5). Finally, we conclude our work and present the plans for the future work in Section 6.

## 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.

### 2.1. Task-based workflows

Some platforms for creating workflows (e.g. FireWorks [7], Taverna [8], Kepler [9], or Galaxy [10]) 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 [11], Swift [12], Parsl [13]), 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 [14].

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 [15] workflow management system uses AWDL [16] (Abstract Workflow Definition Language) to describe workflows of tasks and run them on grids and clouds. PyCOMPSs [17] 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.

### 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 [18], Samza [19] or Storm [20] are data-flow oriented workflow platforms. Barika et al. [21] introduced an XML-based language for stream workflow application specification. These platforms can suffer by increased economic costs if tasks are running without incoming data or by inefficient scheduling of data streams to resources with high load.

### 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 [22]. A data-flow engine invokes functions whenever data inputs are available [23]. 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 [24] is another FC system that was recently released. It allows the user to create worfklows 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 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 [25] 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.

### 2.4. Commercial FC systems

Although various existing open source FC systems and public cloud providers advanced the support for FCs, Lopez et al. [26] reported various pros and cons of several FC systems. For example, while IBM Composer has a simple FC language based on

**Table 1**
Control- and data-flow support of several FC systems. S: supported, N: not supported, L: limited support, A: AWS Step Functions, I: IBM Composer, M: Microsoft Azure Logic Apps, G: Google Cloud Composer, F: Fission Workflows.

| Feature | $\mathcal{A}$ | $\mathcal{I}$ | $\mathcal{M}$ | $\mathcal{G}$ | $\mathcal{F}$ | AFCL |
|---|---|---|---|---|---|---|
| Function implementation abstraction | N | N | N | N | N | S |
| Portability | N | N | N | N | N | S |
| Parallel section | S | N | S | S | N | S |
| `parallelFor` loop | L | N | L | L | N | S |
| Dynamic loop iteration count | S | S | S | L | N | S |
| Multiple input parameters | N | N | N | N | S | S |
| Multiple output parameters | N | N | N | N | N | S |
| Data subset (element-index) | S | L | S | S | L | S |
| Data distribution | L | N | L | N | N | S |
| DAG-based data-flow | N | L | L | S | S | S |

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 ($\mathcal{A}$), IBM Composer ($\mathcal{I}$), Microsoft Azure Logic Apps ($\mathcal{M}$) [27] and Google Cloud Composer ($\mathcal{G}$) [28], as well as Fission Workflows ($\mathcal{F}$) [29] 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 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) requires 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 ($\mathcal{A}$, $\mathcal{M}$, and $\mathcal{G}$). $\mathcal{M}$ and $\mathcal{A}$ additionally provide specific constructs to express parallelFor-Each loops but lack support for parallelFor loops. All providers except $\mathcal{F}$ 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).

We observe that only $\mathcal{F}$ support multiple input parameters for functions, but none of the FC systems supports 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. $\mathcal{I}$ and $\mathcal{F}$ 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 $\mathcal{A}$ and $\mathcal{M}$ 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. $\mathcal{A}$ specifies that output of each function can be accessed from a global FC object without support for DAG-based data-flow, while $\mathcal{I}$ and $\mathcal{M}$ 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 paper introduces the AFCL, which supports all evaluated constructs, thereby dominates all evaluated FC systems.

## 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 [30] that can be used to validate FC programs whether they are compatible with AFCL.

### 3.1. AFCL portable approach with a high-level of abstraction

The left part of Fig. 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.

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). 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.

### 3.2. General overview of AFCL

AFCL is based on YAML [31] which is a human-readable data serialization language. YAML is also the base language for several other platforms [29,32] 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

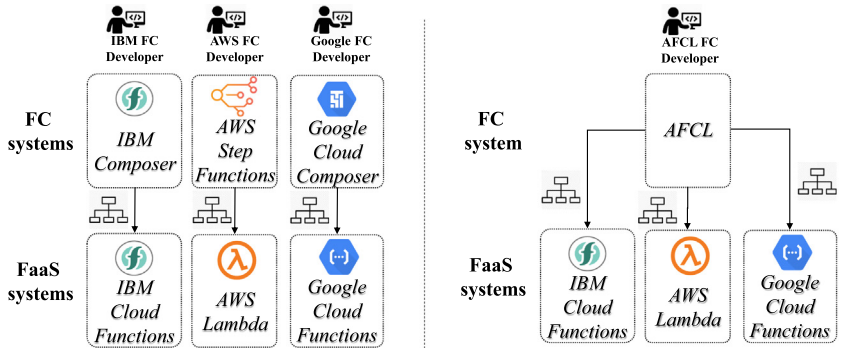



**Fig. 1.** AFCL high-level of abstraction and portable approach.

```
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" }+]?
     }+
   ]?
}
```

**Fig. 2.** Definition of a base function in AFCL.

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*).

### 3.3. Base functions

A base function represents a computational or data processing task. Fig. 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). An 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.

#### 3.3.1. 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 [33], as well as two additional types, *file* and *collection*. The data type *collection* will be explained later in Section 3.6.2.

#### 3.3.2. 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. 2. `Properties` can be used to describe hints about the behavior 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 3.5), built-in constraint (`distribution`) to specify how data is gathered from or distributed among multiple functions (Section 3.6.3), and built-in constraint `element-index` (Section 3.6.2) to specify a subset of a data collection.

In the remainder of this paper 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`).

### 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 on our web site [30].

#### 3.4.1. For

The `for` compound function (Fig. 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 3.6.4).

#### 3.4.2. Parallel

The `parallel` compound function (Fig. 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 3.6.2.

#### 3.4.3. Parallelfor

The `parallelFor` compound function (Fig. 3c) expresses the simultaneous execution of all loop iterations. It is assumed that there are no data dependencies across loop iterations. All other elements of the constructs behave the same as in the `for` compound function described in Section 3.4.1.

### 3.5. Invocation type of a function

The `invoke-type` is a built-in `Property` defined in AFCL. As shown in Fig. 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.

The build-in function `asyncHandler` is used to handle `ASYNC` invoked functions (Fig. 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 coma 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.

### 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.

#### 3.6.1. 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 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, 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.

#### 3.6.2. 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

```
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: [{}+]?
}
```

(a) `for`

```
parallel: {
1: name: "name",
2: dataIns: [{}+]?,
3: parallelBody: [
    {
     section:
      [{function: {}}+]
    }+
   ],
4: dataOuts: [{}+]?
}
```

(b) `parallel`

```
parallelFor: {
1: name: "name",
2: dataIns: [{}+]?,
3: loopCounter: {
     name: "name", type: "type",
     from: "from", to: "to",
     step: "step"?,
   },
4: loopBody:
     [{function: {}}+],
5: dataOuts: [{}+]?
}
```

(c) `parallelFor`

**Fig. 3.** YAML representation of `for`, and `parallel` and `parallelFor` constructs.

```
function: {
1: name: "name", type: "type",
2: dataIns: [{}+]?,
3: properties: [
     {
       name: "invoke-type",
       value: "ASYNC | SYNC"
     }+
   ]?,
4: dataOuts: [{}+ ]?
}
```

(a) `invokeType` property within a function.

```
function: {
1: name: "name", type: "build-in:asyncHandler",
2: dataIns: [
     { name: "name", type: "collection",
       value: "Name1,Name2,...,NameN" }
   ],
3: properties: [
     {
       name: "invoke-type",
       value: "ASYNC | SYNC"
     }
   ]?,
4: dataOuts: [
     { name: "name", type: "boolean" }
   ]
}
```

(b) `asyncHandler` build-in function.

**Fig. 4.** Invocation type of a function.

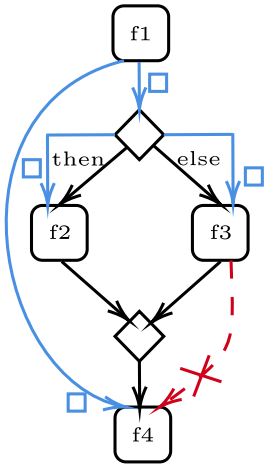
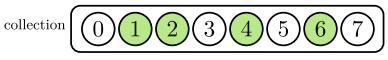



**Fig. 5.** Advanced data-flow support in AFCL: DAG-based Data-flow $f_1 -> f_4$; Illegal data-flow from inner functions of a compound function $f_3 -> f_4$.

```
constraints: [
  {
    name: "element-index",
    value: "<index>"
  }+
]
```

(a) `Index-based`




(b) `element-index(1,2:6:2)`

**Fig. 6.** YAML representation of `Index-based` with corresponding example.

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 3.3.2) `element-index`. With `index` (Fig. 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, $s1$ the start index, $s2$ the end index, and $s3$ a stride.

$$\begin{aligned} e &::= c[, c]* \\ c &::= s1[: s2[: s3]] \end{aligned} \tag{1}$$

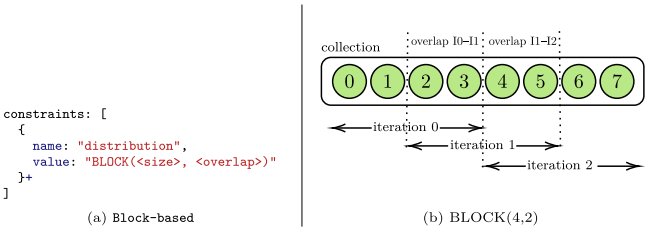
Such an expression can refer to either a specific index or a range of indexes with an optional stride. As an example consider Fig. 6b which selects the elements at positions 1, 2, 4, and 6 of a collection.

### 3.6.3. 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 5.3.

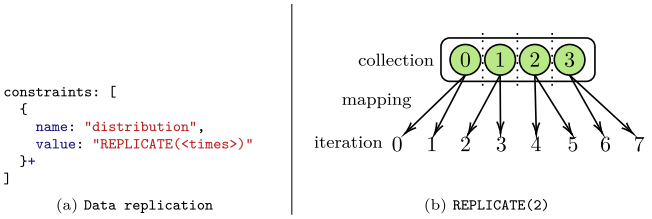
Unlike related work, AFCL offers an additional, build-in, `distribution` *constraint* (explained in Section 3.3.2), which allows the developer to specify how data elements of a collection will be distributed across successor functions. AFCL introduces `block`-based data distribution and data replication.

*Block-based data distribution.* Using the built-in constraint `distribution` and specifying the `block`-based data distribution




(a) Block-based (b) BLOCK(4,2)

**Fig. 7.** YAML representation of `block-based` data distribution.




(a) Data replication (b) REPLICATE(2)

**Fig. 8.** YAML representation of `data replication`.

(Fig. 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. 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).

*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. 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. 8b).

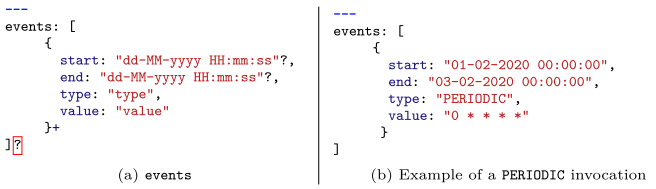
The `dataIn` and `dataOut` ports of each construct presented of this paper can be of type collection in order to collect the data produced by a loop iteration, a parallel section or any other function. After all loop iterations finished, all `dataOuts` are written into the `dataOuts` of the `parallelFor`, `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.

### 3.6.4. Dataloop ports

Every `for` and `while` loop can optionally use `DataLoop` ports (used in Fig. 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.

## 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. 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. 9b. Starting from the 1st of February 2020, the FC is invoked every hour until the 3rd of February, 2020.




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

**Fig. 9.** YAML representation of `events` and example of `PERIODIC` invocation.

## 4. Composing FCs with different FC systems

This section describes two realistic FCs which we composed with AFCL (Section 4.1), AWS Step Functions, and IBM Composer (Section 4.2), as well as their complexity (Section 4.3. 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 2. These compositions will be used for our experimental evaluation in Section 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 providers and can be assumed as base platforms for research on serverless [34] and both FC systems are two mostly referenced serverless technologies in academic publishing [35]. Additionally, Lopez et al. [26] 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 [26,36]. This implementation with a timer operator to sleep reduces the noise in function duration [37,38], 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 [30].

### 4.1. FC applications composed with AFCL

The graphical representation of the corresponding AFCL workflows are shown in Fig. 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.

#### 4.1.1. 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. 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 `calculateTimeTogate` to `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 `selectPassenger` to the specific functions `informPassenger` and `calculateTimeToGate` within the `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. `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 `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 [39]). 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.

#### 4.1.2. 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. 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.

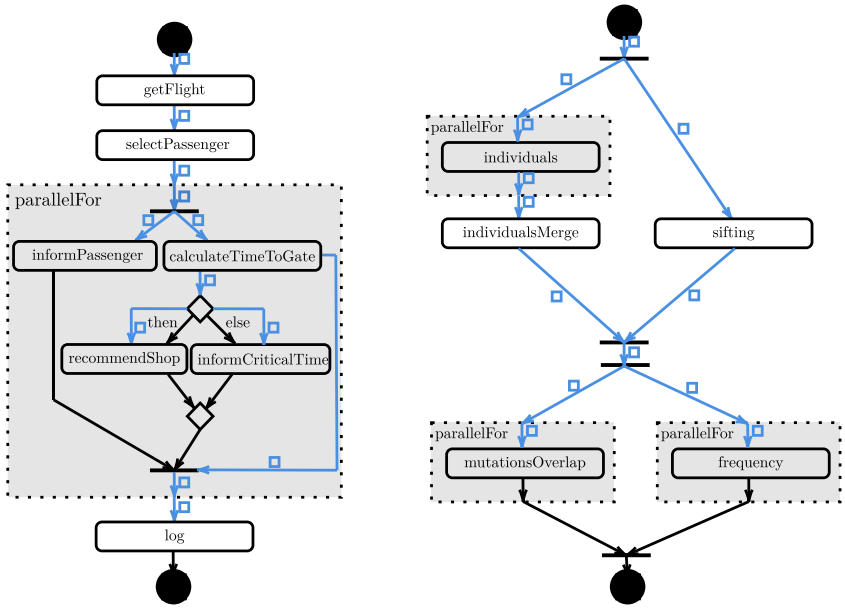
### 4.2. Composing FCs with AWS_Step and IBM_Comp

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_Comp derived the FC files in JSON format. Since IBM_Comp 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_Comp does not support parallelism, we did not composed the GEN FC with IBM_Comp.

All these FC implementations with AWS_Step and IBM_Comp are available on the AFCL web site [30].

[1] https://github.com/pegasus-isi/1000genome-workflow.




(a) GCA. (b) GEN.

**Fig. 10.** Graphical representation of the FCs (a) GCA and (b) GEN. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

**Table 2**
Complexity of all FC implementations in terms of number of iterations and state transitions (for AWS_Step). The complexity of the GCA FC considers the execution of the "then" branch, which is used in the evaluation in Section 5.

| FC | Implementation | Complexity | State transitions |
|---|---|---|---|
| GCA | AFCL | $\mathcal{O}(n)$; $3 \cdot n + 3$ | – |
| GCA | IBM_Comp | $\mathcal{O}(n)$; $3 \cdot n + 3$ | – |
| GCA | AWS_Step | $\mathcal{O}(n)$; $3 \cdot n + 3$ | $\mathcal{O}(n)$; $6 \cdot n + 5$ |
| GEN | AFCL | $O(k)$; $k + 16$ | – |
| GEN | AWS_Step_STAT | $\mathcal{O}(k)$; $k + 16$ | $\mathcal{O}(k)$; $k + 19$ |
| GEN | AWS_Step_DYN | $\mathcal{O}(k)$; $k + 16$ | $\mathcal{O}(k)$; $k + 21$ |

### 4.3. FC complexity analysis

Since both FCs comprise parallel loops, Table 2 presents their complexity in terms of number of iterations ($n$ for the number of passengers for the GCA FC and $k$ for the number of individuals for the GEN FC). Additionally, it shows the state transitions within the state machine for the implementations in AWS_Step because the user is charged for each state transition.

The GCA FC runs the three functions `getFlight`, `selectPassenger`, and `log` only once and $n$ times `calculateTimeToGate`, `recommendShop`, and `informPassenger`, leading to a total of $3 \cdot n + 3$ invocations. The AWS_Step GCA's state machine consists of five states outside the parallel loop (before and after each function) and 6 states within the loop (after each iteration of `MAP`, `Parallel`, `if`, and the three functions).

The GEN FC, on the other side, invokes `individuals` $k$ times, `mutationsOverlap` and `frequency` 7 times and the others only once, which creates a total of $k+12$ invocations. AWS_Step_STAT generates 3 states due to `parallel` construct, or in total $k + 19$ state transitions, while AWS_Step_DYN generates a total of $k+21$ because of the additional *MAP* constructs.

The JavaScript file (IBM_Comp) for the GCA FC comprises only 20 lines of code (LOC), but the generated JSON file has 84 LOC. The corresponding AWS_Step representation has 96 LOC in JSON. The application developer required 125 LOC to compose the same GCA FC with AFCL based on YAML and only 50 LOC using the AFCL JAVA API [14].

## 5. Experimental evaluation of AFCL

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.

### 5.1. Testing methodlogy

#### 5.1.1. 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 (`SeqenceMaxActions`),[2] we were able to run up to $n = 15$ loop iterations. AWS Lambda allows a much higher limit of up to 1000 concurrent function invocations[3] 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 $k = 200$.

[2] https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-limits.
[3] https://docs.aws.amazon.com/lambda/latest/dg/limits.html.

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 $k$.

#### 5.1.2. 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 [37,40]. 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 100 ms.

### 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.

#### 5.2.1. Development effort for the GCA FC

The left column of Table 3 shows the development effort for each used construct of the GCA FC. It took only approximately 85 min to compose the GCA FC with AFCL, which is 12.37% better than 97 min with AWS_Step, and even 23.42% better than 111 min with IBM_Comp.

AFCL required more time (3 min per function) to define the seven FC functions (due to its ability to support multiple input and output data ports), followed by 2 min per function for AWS_Step and one minute for IBM_Comp (less configurable parameters). The developer needs more time (2 min 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 min. AWS_Step supports parallel loops using the `MAP` construct and AFCL provides the `ParallelFor` construct, which took 5 min for each, 5 more minutes each for nesting the `parallel` construct inside the parallel loop, and yet additional 5 min 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 min to specify each multiple input parameters including the inputs to the parallel loop and the nested compound functions (total five). The same time of $5 \cdot 1$ min 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 min 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 min with IBM_Comp, 20 min with AFCL (simpler with data collection and DAG-based data-flow) and 15 min 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.

#### 5.2.2. Development effort for the GEN FC

The right column of Table 3 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 min 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

**Table 3**
Development effort for both FCs with various FC systems. AF: AFCL, A: AWS Step Functions, I: IBM Composer, AD$k$: AWS_Step_DYN with $k$ individuals, AS$k$: 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.

| FC construct | GCA | | | GEN | | |
|---|---|---|---|---|---|---|
| | AF | A | I | AF | AD$k$ | AS$k$ |
| Function definitions | $7 \cdot 3$ | $7 \cdot 2$ | $7 \cdot 1$ | $5 \cdot 3$ | $5 \cdot 2$ | $5 \cdot 2$ |
| Sequence | $4 \cdot 1$ | $4 \cdot 2$ | $4 \cdot 1$ | $4 \cdot 1$ | $4 \cdot 2$ | $4 \cdot 2$ |
| If-then-else | $1 \cdot 10$ | $1 \cdot 10$ | $1 \cdot 10$ | — | — | — |
| Parallel section | $1 \cdot 5$ | $1 \cdot 5$ | X | $2 \cdot 5$ | $2 \cdot 5$ | $2 \cdot 5 + k \cdot 2$ |
| ParallelFor loop | $1 \cdot 5$ | $1 \cdot 5$ | X | $3 \cdot 5$ | $3 \cdot 5$ | — |
| Dynamic loop iterations | $1 \cdot 5$ | $1 \cdot 5$ | $1 \cdot 30$ | $1 \cdot 5$ | $1 \cdot 5$ | X |
| Multiple output param. | $5 \cdot 1$ | $5 \cdot 3$ | $5 \cdot 2$ | — | — | — |
| Multiple input param. | $5 \cdot 1$ | $5 \cdot 3$ | $5 \cdot 2$ | $10 \cdot 1$ | $10 \cdot 2$ | $10 \cdot 2$ |
| Data distribution | $1 \cdot 5$ | $1 \cdot 5$ | $1 \cdot 10$ | $1 \cdot 5$ | $1 \cdot 10$ | $1 \cdot 10$ |
| DAG-based data-flow | $1 \cdot 20$ | $1 \cdot 15$ | $1 \cdot 30$ | — | — | — |
| Input preparation | — | — | — | 5 | $5 + k \cdot 0.5$ | $5 + k \cdot 0.5$ |
| **Total** | **85** | **97** | **111** | **69** | $83 + 0.5 \cdot k$ | $63 + 2.5 \cdot k$ |

required 10 min to develop the data distribution. Secondly, a fixed time of 5 min was needed to create references for the input files and upload them to S3. Thirdly, in the state machine, an additional 0.5 min was spent per function to prepare the input files for each function `individuals`. In AFCL, the input preparation is fixed to 5 min, 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. 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 development 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 parallel 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, respectively. 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.

### 5.3. Makespan evaluation

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

#### 5.3.1. Makespan for the GCA FC

The full potential of AFCL can be observed for the makespan of GCA FC, which is presented in Fig. 12. AFCL_IBM reduces the makespan compared to IBM_Comp starting from 7.34% for the smallest problem size (number of passengers) of $n = 1$ (GCA1 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_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_AWS runs the entire GCA FC from 21.48% ($n = 1$) up to impressive 84.2% ($n = 15$) faster than AWS_Step. Despite the parallelism support through the MAP construct of AWS_Step, AFCL_AWS achieved a higher speedup than AWS_Step compared to AFCL_IBM vs IBM_Comp. This is caused because of the state machine approach of AWS_Step whereas IBM_Comp invokes additional functions for data transfers between functions of an FC. Even more, AFCL_IBM achieved lower makespan than AWS_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_AWS substantially reduced the makespan by approximately 98% compared to AWS_Step, which was mainly caused by the delayed function invocation in the `MAP` construct, while the bias in function duration was negligible.

AFCL_AWS outperforms AWS_Step in terms of efficiency, as well. While the efficiency of AFCL_AWS is retained to 90% for $n = 15$ and downgraded to 29.9% for $n = 200$, AWS_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_AWS achieved lower makespan for $n = 200$ than AWS_Step for $n = 10$. Similarly, IBM_Comp achieved efficiency of only 13.07% for problem size $n = 15$, while AFCL_IBM three times higher, or 39.27%.

#### 5.3.2. Makespan for GEN FC

Fig. 13 shows the makespan results for GEN FC created based on results for AWS_Step_DYN, AWS_Step_STAT and AFCL_AWS. We observe that AFCL_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.

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 synchronization function `IndividualsMerge`. We observe that all three implementations achieved maximal speedup of 1.5 up to $k = 20$, but only

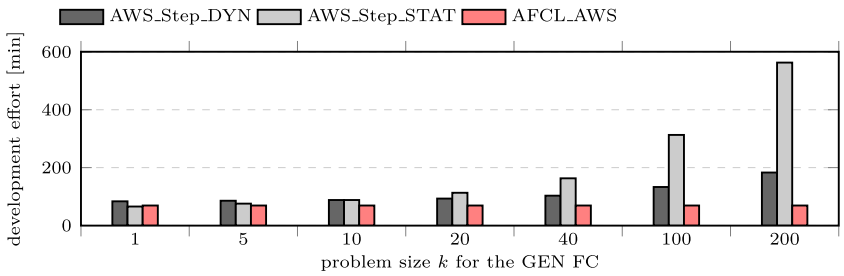



**Fig. 11.** Evaluated development effort to create GEN FC with AFCL and with *AWS_Step* (dynamic sequential and static parallel) for various problem sizes $k$.

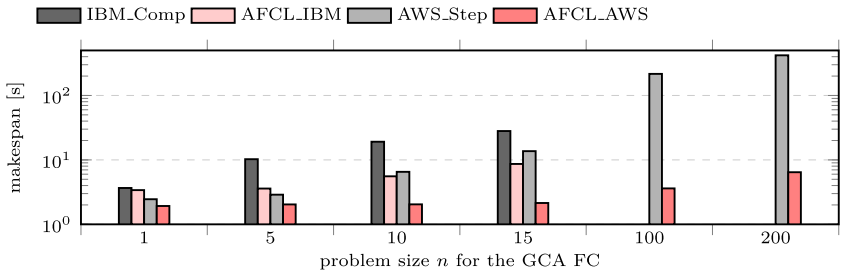



**Fig. 12.** Average makespan (logarithmic) of the GCA FC with different problem sizes $n$ (number of passengers) ranging from 1 to 200, implemented with various FC systems. Two bars for $n = 100$ and $n = 200$ are missing due to the limitation of the IBM Composer to allow only a maximum of 50 function invocations.

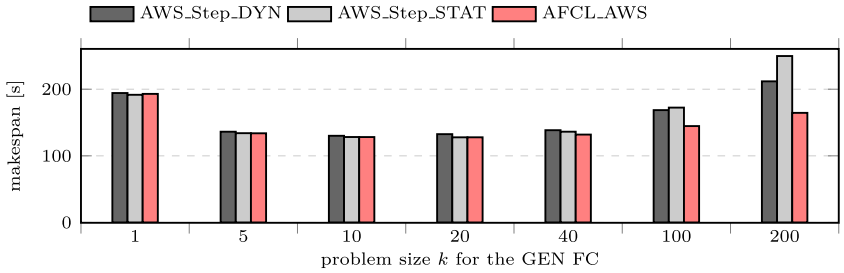



**Fig. 13.** Average makespan of the GEN FC implemented with the dynamic (AWS_Step_DYN) and static compositions (AWS_Step_STAT) on AWS Step Functions, as well as with the AFCL environment (AFCL_AWS) for different $k$ (number of instances of function `individuals`).

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`).

### 5.4. Economic cost evaluation

Cost and performance are often conflicting criteria for workflows in cloud systems [41]. However, our evaluation showed that these two criteria are not necessarily conflicting for all evaluated FC systems.

#### 5.4.1. Costs for GCA FC

Fig. 14 also displays the estimated costs for the GCA FC for the same experiments of Fig. 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.

#### 5.4.2. Costs for GEN FC

Fig. 15 displays the results for the estimated economic costs of the experiments presented in Fig. 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 (164.5 s) for the GEN FC compared to 6.44 s for the GCA FC, thereby the costs for INVOKER duration are not negligible compared to costs for state transitions in AWS_Step.

### 5.5. Discussion

The main advantage of AFCL over AWS_Step are its constructs for allowing multiple input and output data of each function and specify data-flow between functions directly, rather than

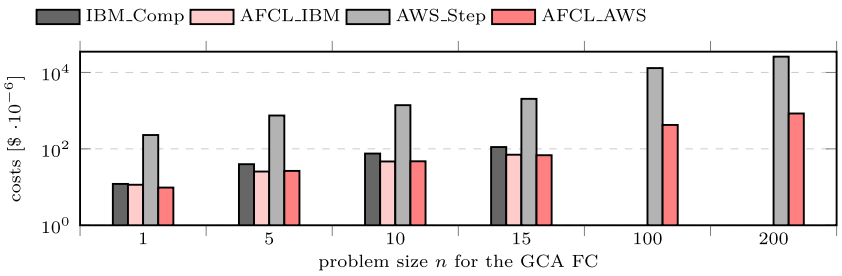



**Fig. 14.** Economic costs (logarithmic scale) for the corresponding experiments presented in Fig. 12.

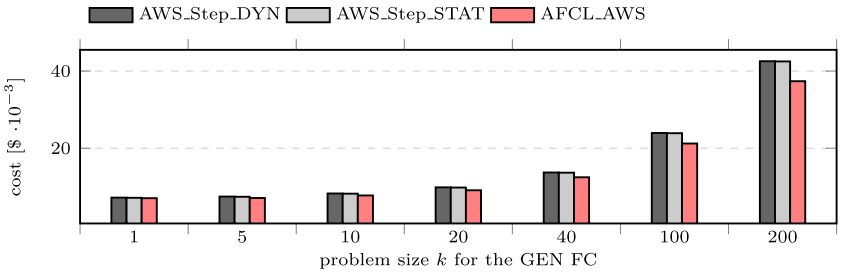



**Fig. 15.** Economic costs for the corresponding experiments presented in Fig. 13.

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 1000 concurrent functions.

### 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 3. 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 s of AFCL INVOKER function execution which uses 128 MB memory. This means that FCs with a makespan shorter than the product $12 \cdot m$, 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.

## 6. Conclusion and future work

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.

Our future work focuses on extending AFCL with other compound functions (e.g. $N - out - of - M$ or discriminators). We are currently developing a web application that will consider all AFCL constraints and help the business community to build their applications as FCs.

Economic costs do not follow the makespan, especially in AWS Step Functions. High degree of parallelism may increase the costs, but does not necessarily reduce the makespan, as reported for the GEN FC. We will work on a novel multi-objective scheduling algorithm that will adapt the AFCL representation of an FC to the optimal degree of parallelism and data-distribution accordingly in order to minimize both objectives, i.e., the economic costs and the makespan. For example, instead of running three functions that run 30 m each, we will run a single function that will get a data collection as an input and run within the same time unit of 100 ms, thereby reducing the economic costs.

## CRediT authorship contribution statement

**Sasko Ristov:** Conceptualization, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Writing - original draft. **Stefan Pedratscher:** Resources, Software, Visualization. **Thomas Fahringer:** Supervision, Validation, Writing - review & editing.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgment

This work is supported by the Austrian Research Promotion Agency (FFG) as part of the INPACT project under contract no 868018.

## References

[1] D. Jackson, G. Clynch, An investigation of the impact of language runtime on the performance and cost of serverless functions, in: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), 2018, pp. 154–160, http://dx.doi.org/10.1109/UCC-Companion.2018.00050.

[2] J.M. Hellerstein, J. Faleiro, J.E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, C. Wu, Serverless computing: One step forward, two steps back, in: CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, 2019.

[3] K. Kritikos, P. Skrzypek, A review of serverless frameworks, in: IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), 2018.

[4] R. Pellegrini, I. Ivkic, M. Tauber, Towards a security-aware benchmarking framework for function-as-a-service, in: International Conference on Cloud Computing and Services Science, 2018.

[5] Amazon step functions, 2019, URL https://aws.amazon.com/step-functions/ (Accessed: 26-06-2019).

[6] IBM Composer, 2019, URL https://github.com/ibm-functions/composer (Accessed: 26-06-2019).

[7] A. Jain, S.P. Ong, W. Chen, B. Medasani, X. Qu, M. Kocher, M. Brafman, G. Petretto, G.-M. Rignanese, G. Hautier, D. Gunter, K. Persson, Fireworks: a dynamic workflow system designed for high-throughput applications, Concurr. Comput.: Pract. Exper. 27 (17) (2015) 5037–5059.

[8] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A.N. de la Hidalga, M.P.B. Vargas, S. Sufi, C. Goble1, The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud, Nucleic Acids Res. 41 (W1) (2013) W557–W561.

[9] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Lee, J. Tao, Y. Zhao, Scientific workflow management and the kepler system: Research articles, Concurr. Comput.: Pract. Exper. 18 (10) (2006) 1039–1065.

[10] E. Afgan, D. Baker, M. Van den Beek, D. Blankenberg, D. Bouvier, M. Čech, J. Chilton, D. Clements, N. Coraor, C. Eberhard, C. Eberhard, B. Grüning, A. Guerler, J. Hillman-Jackson, G. Von Kuster, E. Rasche, N. Soranzo, N. Turaga, J. Taylor, A. Nekrutenko, J. Goecks, The galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2016 update, Nucleic Acids Res. 44 (W1) (2016) W3–W10.

[11] P. Di Tommaso, M. Chatzou, E.W. Floden, P.P. Barja, E. Palumbo, C. Notredame, Nextflow enables reproducible computational workflows, Nature Biotechnol. 35 (4) (2017) 316.

[12] M. Wilde, M. Hategan, J.M. Wozniak, B. Clifford, D.S. Katz, I. Foster, Swift: A language for distributed parallel scripting, Parallel Comput. 37 (9) (2011) 633–652, http://dx.doi.org/10.1016/j.parco.2011.05.005.

[13] Y. Babuji, A. Woodard, Z. Li, D.S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J.M. Wozniak, I. Foster, et al., Parsl: Pervasive parallel programming in python, in: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, in: HPDC '19, Association for Computing Machinery, Phoenix, AZ, USA, 2019, pp. 25–36, http://dx.doi.org/10.1145/3307681.3325400.

[14] AFCLCore: Java API to create and manage AFCL Function Choreographies, 2020, URL https://github.com/stefanpedratscher/AFCLCore (Accessed: 26-06-2020).

[15] S. Ostermann, R. Prodan, T. Fahringer, Extending grids with cloud resource management for scientific computing, in: 2009 10th IEEE/ACM International Conference on Grid Computing, 2009, pp. 42–49, http://dx.doi.org/10.1109/GRID.2009.5353075.

[16] J. Qin, T. Fahringer, Scientific Workflows, Programming, Optimization, and Synthesis with ASKALON and AWDL, Springer, 2012, ISBN: 978-3-642-30714-0. (Print) 978-3-642-30715-7.

[17] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R.M. Badia, J. Torres, T. Cortes, J. Labarta, Pycompss: Parallel computational workflows in python, Int. J. High Perform. Comput. Appl. 31 (1) (2017) 66–82, http://dx.doi.org/10.1177/1094342015594678, arXiv:https://doi.org/10.1177/1094342015594678.

[18] Apache Flink Contributors, Apache flink, 2020, URL https://flink.apache.org/ (Accessed: 13-01-2020).

[19] Apache Samza Contributors, Apache samza, 2020, URL https://samza.apache.org/ (Accessed: 13-01-2020).

[20] Apache Storm Contributors, Apache storm, 2020, URL https://storm.apache.org/ (Accessed: 13-01-2020).

[21] M. Barika, S. Garg, A. Chan, R.N. Calheiros, R. Ranjan, Iotsim-stream: Modelling stream graph application in cloud simulation, Future Gener. Comput. Syst. 99 (2019) 86–105, http://dx.doi.org/10.1016/j.future.2019.04.004.

[22] M. Malawski, A. Gajek, A. Zima, B. Balis, K. Figiela, Serverless execution of scientific workflows: Experiments with hyperflow, AWS lambda and google cloud functions, Future Gener. Comput. Syst. (2017) http://dx.doi.org/10.1016/j.future.2017.10.029.

[23] B. Balis, Hyperflow: A model of computation, programming approach and enactment engine for complex distributed workflows, Future Gener. Comput. Syst. 55 (2016) 147–162, http://dx.doi.org/10.1016/j.future.2015.08.015.

[24] A. John, K. Ausmees, K. Muenzen, C. Kuhn, A. Tan, SWEEP: Accelerating scientific research through scalable serverless workflows, in: IEEE/ACM International Conference on Utility and Cloud Computing Companion, in: UCC '19 Companion, Association for Computing Machinery, Auckland, New Zealand, 2019, pp. 43–50, http://dx.doi.org/10.1145/3368235.3368839.

[25] G. Zheng, Y. Peng, Globalflow: A cross-region orchestration service for serverless computing services, in: IEEE International Conference on Cloud Computing (CLOUD), 2019, pp. 508–510.

[26] P.G. López, M. Sánchez-Artigas, G. París, D.B. Pons, Á.R. Ollobarren, D.A. Pinto, Comparison of faas orchestration systems, in: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), IEEE, 2018, pp. 148–153.

[27] Microsoft azure logic apps, 2019, URL https://azure.microsoft.com/en-us/services/logic-apps/ (Accessed: 26-06-2019).

[28] Google cloud composer, 2019, URL https://cloud.google.com/composer/ (Accessed: 26-06-2019).

[29] Fission workflows, 2019, URL https://github.com/fission/fission-workflows (Accessed: 26-06-2019).
[30] Abstract function choreography language, 2019, URL http://dps.uibk.ac.at/projects/afcl/ (Accessed: 20-12-2019).
[31] YAML ain't markup language, 2019, URL https://yaml.org/ (Accessed: 26-06-2019).
[32] Apache airflow, 2019, URL https://airflow.apache.org/ (Accessed: 26-06-2019).
[33] Json schema: A media type for describing JSON documents, 2019, URL json-schema.org/latest/json-schema-core.html (Accessed: 15-04-2019).
[34] T. Lynn, P. Rosati, A. Lejeune, V. Emeakaroha, A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms, in: 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2017, pp. 162–169, http://dx.doi.org/10.1109/CloudCom.2017.15.
[35] M. Al-Ameen, J. Spillner, Systematic and open exploration of faas and serverless computing research, in: Proceedings of the European Symposium on Serverless Computing and Applications, ESSCA@UCC 2018, Zurich, Switzerland, December 21, 2018., 2018, pp. 30–35, URL http://ceur-ws.org/Vol-2330/short2.pdf.
[36] R. Chard, T.J. Skluzacek, Z. Li, Y. Babuji, A. Woodard, B. Blaiszik, S. Tuecke, I. Foster, K. Chard, Serverless supercomputing: High performance function as a service for science, 2019, arXiv preprint arXiv:1908.04907.
[37] R. Mathá, S. Ristov, R. Prodan, Simulation of a workflow execution as a real cloud by adding noise, Simul. Model. Pract. Theory 79 (2017) 37–53.
[38] R. Mathá, S. Ristov, T. Fahringer, R. Prodan, Simplified workflow simulation on clouds based on computation and communication noisiness, IEEE Trans. Parallel Distrib. Syst. 31 (7) (2020) 1559–1574.
[39] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P.J. Maechling, R. Mayani, W. Chen, R.F. Da Silva, M. Livny, et al., Pegasus, a workflow management system for science automation, Future Gener. Comput. Syst. 46 (2015) 17–35.
[40] H.M. Fard, S. Ristov, R. Prodan, Handling the uncertainty in resource performance for executing workflow applications in clouds, in: 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC), 2016, pp. 89–98.
[41] J.J. Durillo, R. Prodan, Multi-objective workflow scheduling in amazon ec2, Cluster Comput. 17 (2) (2014) 169–189.

**Sasko Ristov** received his Ph.D. in computer science in 2012 from University of Ss. Cyril and Methodius, Skopje, North Macedonia (UKIM) and was Assistant Professor at UKIM until 2019. Since 2016 he is a post-doctoral university assistant at University of Innsbruck. Dr. Ristov authored more than 100 research articles in the areas of performance modeling, optimization and parallel and distributed systems. He received the UKIM's Best Scientist award in 2012 and an IEEE conference best paper award in 2013.

**Stefan Pedratscher** received his Bachelor in computer science in 2018 from University of Innsbruck. He is currently working on his master thesis.

**Thomas Fahringer** received his Ph.D. degree from the Vienna University of Technology in 1993. Since 2003, he has been a full professor of computer science at the Institute of Computer Science, University of Innsbruck, Austria. His main research interests include software architectures, programming paradigms, compiler technology, performance analysis, and prediction for parallel and distributed systems. He is a member of the IEEE.