FaaScinating Resilience for Serverless Function Choreographies in Federated Clouds

Sasko Ristov, Dragi Kimovski, and Thomas Fahringer, *Member, IEEE*

***Abstract*—Cloud applications often benefit from deployment on serverless technology Function-as-a-Service (FaaS), which may instantly spawn numerous functions and charges users for the period when serverless functions are running. Maximum benefit is achieved when functions are orchestrated in a workflow or *function choreographies* (*FCs*). However, many provider limitations specific for FaaS, such as maximum concurrency or duration often increase the failure rate, which can severely hamper the execution of entire FCs. Current support for resilience is often limited to function *retries* or *try-catch*, which are applicable within the same cloud region only. To overcome these limitations, we introduce *rAFCL*, a middleware platform that maintains reliability of complex FCs in federated clouds. In order to support resilient FC execution under *rAFCL*, our model creates an alternative strategy for each function based on the required availability specified by the user. Alternative strategies are not restricted to the same cloud region, but may contain alternative functions across five providers, invoked concurrently in a single alternative plan or executed subsequently in multiple alternative plans. With this approach, *rAFCL* offers flexibility in terms of cost-performance trade-off. We evaluated *rAFCL* by running three real-life applications across three cloud providers. Experimental results demonstrated that *rAFCL* outperforms the resilience of AWS Step Functions, increasing the success rate of entire FC by 53.45%, while invoking only 3.94% more functions with zero wasted function invocations. *rAFCL* significantly improves availability of entire FCs to almost 1 and survives even after massive failures of alternative functions.**

***Index Terms*—Availability, disasters, failures, function-as-a-service, HPC, reliability, workflow applications.**

I. Introduction

Cloud data centers usually comprise several independent availability zones to increase their reliability. Consequently, cloud providers often claim availability of "four nines" for their services. Moreover, serverless computing, particularly Function-as-a-Service (FaaS), gains more traction among developers because it supports a higher level of automation, simplifies code deployment, eliminates the need for manual resource provisioning, and provides increased elasticity with low communication latency. The FaaS model enables service provisioning within a few milliseconds [1] and is used for data analytics primarily in the cloud and at the edge [2].

To fully benefit from FaaS, developers usually build more complex serverless applications by connecting functions with data- and control-flow in batch-oriented *function choreographies* (FCs) [3] or serverless workflows. For this purpose, all dominating cloud providers offer *FC systems*. For example, AWS offers Step Functions to build FCs but can run functions only on its *corresponding FaaS system* AWS Lambda. For non-critical FCs, users usually rely on providers' resilient techniques. Unfortunately, failures are inevitable and may occur at any time, which burdens the execution of critical FCs because of four deficiencies.

Firstly, although serverless functions are usually lightweight short-running, still, they could fail, block, or not even start due to various reasons, such as software, network, or hardware failures. Functions failures can cause adverse consequences unless they are not anticipated and handled appropriately [4]. The entire FC may fail even by *sporadic failures*, that is, if even a single function fails and no appropriate resilient measure is taken. While FCs offer a novel programming model in serverless computing, which provides high degree of parallelism and controlled flow of interconnected functions within an FC, still, a failure of an individual function usually leads to a failure to meet the deadline constraint or often a failure of the entire FC [5]. Therefore, specific resilient techniques are required for various kinds of FCs [6].

Secondly, FC systems of dominating cloud providers are still in their infancy with numerous drawbacks, individual limitations, and constraints [7], which may lead to failures. As a consequence, functions may terminate with errors due to maximum duration time of a function, input and output data size, size of the code, and limited allocated memory. Even worse, most of these limitations cannot be detected statically and are visible at runtime only. Furthermore, FCs that need to be scaled are also affected by the FaaS systems limitations, such as the declared limit of 1,000 concurrent functions. Running a higher number of concurrent functions will either cause a failure, or users will experience a delay in their makespan [8]. Such scenarios often lead to *massive failures*. Moreover, many providers offer authentication based on timely limited session token, which may cause massive failures until the token is renewed.

Manuscript received 23 November 2021; revised 17 March 2022; accepted 20 March 2022. Date of publication 24 March 2022; date of current version 12 October 2022. This research is supported from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 951745, in particular, the FF4EUROHPC CALL-2, for the project entitled "CardioHPC Improving DL-based Arrhythmia Classification Algorithm and Simulation of Real-Time Heart Monitoring of Thousands of Patients." The associate editor coordinating the review of this article and approving it for publication was M. Tornatore. *(Corresponding author: Sasko Ristov.)*

Sasko Ristov and Thomas Fahringer are with the Distributed and Parallel Systems Group, University of Innsbruck, 6020 Innsbruck, Austria (e-mail: sashko@dps.uibk.ac.at; tf@dps.uibk.ac.at).

Dragi Kimovski is with the Institute of Information Technology, Klagenfurt University, 9020 Klagenfurt, Austria (e-mail: dragi.kimovski@aau.at).

Digital Object Identifier 10.1109/TNSM.2022.3162036

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

Thirdly, although cloud providers offer support for resilient techniques, the application domain is limited within the same region of that provider without support for *federated clouds*. For instance, AWS Step Functions in Frankfurt may run serverless functions in its corresponding FaaS system AWS Lambda in Frankfurt only, thereby the resilient techniques are available for AWS Step Functions (e.g., *alternate resource*) and AWS Lambda (e.g., *function retry*) in Frankfurt only. Moreover, these techniques might be unable to handle various natural-based [9], weather-based [10], technology-based disasters [11], or disasters deliberately caused by human attacks [12].

Finally, a common weakness of cloud providers is the specified level of availability in their service level agreements (SLA). For instance, AWS guarantees 99.95% availability of their regions. However, the way how the availability is calculated is inconsistent because, for instance, AWS calculates availability per customer each five minutes. If in this time period, a customer does not run any function, availability is considered as 100%.[1] This approach results in a much higher availability than the actually achieved one.

To bridge the gap of the above-mentioned shortcomings in FC resilience, this manuscript introduces a novel *resilient rAFCL middleware platform* that may run an *alternative strategy* comprising multiple alternative functions of each primary function of an FC across the globe. *rAFCL*'s resilient techniques are applicable on top of our recent *xAFCL* FC management system [13], which runs FCs in federated clouds. In this paper, we extend *xAFCL* to monitor FC functions and run alternatives across the *top five* clouds AWS, Azure, Google, Alibaba, and IBM. Unlike existing FC systems which are limited to a single cloud region, *rAFCL* may run and apply resilient techniques on the same region, other region of the same cloud provider, or even on any region of other cloud providers without provider lock-in. Moreover, to mitigate the impact of the primary functions failure, the decision how many alternative functions to be invoked in other cloud providers is taken by *rAFCL* at runtime for maximum flexibility. *rAFCL* offers numerous advantages compared to the resilient techniques of individual cloud providers. Firstly, using proactive and reactive resilient techniques, *rAFCL* reaches the required availability of each function and thereby of the entire FC. Secondly, *rAFCL* uses greedy scheduling algorithm which minimizes the cost of alternative strategies. Finally, *rAFCL* can automatically distribute alternative functions of an FC to multiple regions across several FaaS systems, thus increasing scalability and improving performance. *rAFCL* increases the success rate of the entire FC by 53.45% with average increase of the average number of invocation by only 3.94% with zero wasted function invocations for functions with availability of at least 60%. With the flexible approach of the alternative strategies to run sequentially or parallel, *rAFCL* may speed up execution of a function by 52.8% with a trade-off of 2.5× higher cost, even for functions with success rate of 25%.

In summary, the contributions of this work are:

- Publicly available *rAFCL* platform for building, scheduling, and monitoring FCs across five FaaS systems;
- Unlike existing approaches, which impose the restriction for resilient techniques within the same region only, *rAFCL* offers resilient techniques in federated clouds;
- The *rAFCL* resilience model considers latest start and finish time, maximum function duration, and number of retries for every function of an FC;
- The *rAFCL* scheduler selects user-provided alternative functions to reach the required availability and minimize the cost of each primary function of the FC;
- *rAFCL* survives even in case of massive failures.

This paper is organized in seven sections. Section II presents thorough and critical overview of the related work. Section III provides a summary of the existing resilient techniques supported by the established cloud providers and the *rAFCL* support for resilient FCs in federated clouds. Section IV describes the overall architecture of *rAFCL* and its current implementation. The *rAFCL* resilience model and scheduler are elaborated in Section V. We evaluate resilience of *rAFCL* and discuss its limitations in Section VI. Finally, we conclude our work and present future work in Section VII.

II. Related Work

This section compares resilient techniques of *rAFCL* with cloud providers and other FC systems. It also presents how *rAFCL* advances beyond the state-of-the-art.

A. Resilient Techniques by Cloud Providers

Many cloud providers offer FC systems to orchestrate functions in the form of an FC. Moreover, their FC systems provide support for resilience, mainly as a reactive measure in case some of its functions fail. In general, cloud providers offer retry and try-catch, and some offer specific resilient techniques such as function rollback or try-catch-finally.

AWS Step Functions provide both reactive and proactive resilient techniques. Users can specify the maximum number of retries for every FC function. Additionally, AWS Step Functions supports try-catch blocks during the FC development. Moreover, AWS Step Functions introduced a state machine pattern that retains the state of a function by using reactive measures to invoke another function in case of a failure state [14]. However, these techniques are limited within the same AWS region. Similarly, IBM Composer supports try-catch-finally and function retry. Google Composer provides retrying a function and raises various kinds of exceptions without handlers in case a function fails. Alibaba Serverless Workflow offers try-catch, function retry, and rollback.

B. FC Systems for Federated Clouds

Several FC systems were introduced recently that can run FCs across FaaS systems. Malawski *et al.* [15] extended Hyperflow [16] to support FCs and run them on the specified regions of AWS or Google. MPSC [17] went a step further as it can run FCs across backend FaaS systems AWS Lambda and IBM Cloud Functions. However, both HyperFlow and MPSC may run an FC on a single FaaS system region at a time, thereby are restricted to the limitations of that FaaS system.

[1] https://aws.amazon.com/lambda/sla/

Risco *et al.* [18] introduced OSCAR, a programming language agnostic serverless framework, which offloads functions to a single cloud region when the edge resources are overloaded. However, OSCAR neither considers the concurrency limitations of the FaaS providers nor can it schedule complex FCs with data and control flow dependencies. Baarzi *et al.* [19] present the concept of virtual serverless providers, which federates multiple cloud providers through a third party entity responsible for enabling interoperability between the functions and preventing vendor lock-in. However, the presented concept does not consider the resilience of FCs, which can lead to a higher number of Service Level Objective (SLO) violations due to execution failures.

Other FC systems use Python to orchestrate FCs. One such FC system is PyWren [20], in which a single serverless function (the executor) fetches Python code and the input data from the cloud storage and runs that code. With this approach, it is possible to overcome some cloud provider limitations (input data size or function code size). However, there are still several weaknesses. Firstly, it generates additional runtime overhead to fetch the code and data input. Secondly, the function developer must rewrite all functions written in a programming language different from Python. Thirdly, the duration of the FC is limited to the maximum duration of the FaaS system. Finally, if the executor function fails, no resilient technique is provided and the entire FC fails. Another FC system is SWEEP [21], which supports AWS Lambda and AWS Fargate containers as backend FaaS systems. However, SWEEP does not support FCs with branches and is limited to a single region of AWS. Finally, CRUCIAL [22] offers a novel programming model to orchestrate functions in Java by introducing the concept of cloud thread, which corresponds to a single cloud function. CRUCIAL relies on AWS Lambda and uses its retry techniques for fault tolerance. However, CRUCIAL supports a single cloud region and provider and the retry resilient technique only.

Another approach for orchestrating serverless functions is to faasify a monolith and use it to orchestrate functions in an FC. Node2FaaS [23] supports porting of monoliths as FCs by offloading their methods as functions. Although Node2FaaS ports functions on multiple FaaS systems, the generated FC can run on one region of a single FaaS system at a time, without the option for a multi-FaaS scenario. Moreover, resilient techniques have to be implemented in the monolith with considerable development effort. Node2FaaS restricts scalability because the monolithic architectural style does not utilize the serverless architecture. It also restricts resilience in a single region because the generated hybrid FC (monolith + serverless) runs functions on one region only of a single cloud provider. Moreover, this approach can be applied to simple functions without code and package dependencies [24].

GlobalFlow [25] is an FC system that can distribute an FC written in AWS Step Functions across multiple regions of AWS. This approach may partially overcome the concurrency limit of a single region and massive failures. Still, the FC is locked in AWS, thereby limiting portability. Moreover, similar to PyWren, GlobalFlow runs additional functions, increasing overall makespan and economic costs.

C. FC Schedulers

Several works describing FCs schedulers exist. The introduced FC scheduling approaches are usually written in high level languages, such as Python. To begin with, Pheromone [26] uses data-centric scheduling for FCs and triggers successor functions once data is available. Pheromone allows developers to explicitly specify shuffling data-flow between FC functions. Sequoya [27] is another FC system that acts as a proxy and offers several scheduling policies to developers, based on various quality of service (QoS) parameters. After a function finishes, it runs successor functions of the FC on OpenWhisk or AWS. Wukong [28] is an FC system that uses a decentralized scheduler, which splits the directed acyclic graph (DAG) of the FC into multiple static schedules. The schedules may contain multiple copies of tasks, however the executors collaborate with each other to run each task only once. The three FC systems, Pheromone, Sequoya, and Wukong, are suitable for scheduling the FCs whose functions work over the same data in a single storage. However, none of them is applicable if FC functions are distributed in federated clouds across the globe. Moreover, they are limited to a single programming language and cloud region, while *rAFCL* is programming language and FaaS provider agnostic.

Other approaches optimize the makespan with budget constraints. Kijak *et al.* [29] introduced a scheduler that selects functions assigned with various memory size and optimizes the cost-performance ratio. Skedulix [30], on the other side, offloads functions from OpenFaaS to AWS Lambda to minimize the costs by given deadline constraint. However, both approaches are restricted to small-scale FCs due to concurrency limitations of a single cloud region. In contrast, our *rAFCL* scheduler is among the first that considers resilience of FC functions in federated clouds.

III. Resilience in Serverless

In general, resilient techniques can be classified as reactive or proactive [31]. Proactive resilient techniques tend to minimize the number of failures that occur. Their goal is to anticipate failures and proactively minimize their effects, thereby increase the system reliability. However, even with proactive measures, failures are inevitable in computer systems and networks. As a consequence, reactive resilient measures are needed to recover the system and minimize failures' impact. The goal of reactive resilient measures is to continue to serve client requests even in the presence of failures and to recover quickly within an acceptable period of time.

A. Reactive Resilient Techniques

Function retry is the simplest reactive measure. This means restarting the same function again after a failure occurs. Often, failures caused by network issues, power outages, hardware failures, throttling or API invoked errors are handled by retrying the functions. Short-running serverless functions will benefit to be retried in case of a failure. *Alternate function* is another reactive technique that may alleviate the impact of disruptions, especially in case of a permanent failure of the primary function. The failed function is replaced with the

same function code but deployed with more memory. Because serverless functions are in general stateless, they do not keep the state of executed code in case of a failure. This causes that the above-mentioned techniques are not applicable if functions increase or decrease value of some data item stored in a database [22]. Therefore, the presented reactive measures are valid for idempotent functions, i.e., functions whose result is not affected if they are executed multiple times.

While reactive measures may reduce the impact of failures, still, in general, computer systems need mechanisms to minimize the failure rate. For example, reactive measures may be not applicable in real-time systems where data will be lost if a function, which handles data stream, fails.

B. Proactive Resilient Techniques

Checkpointing is a proactive resilient technique that saves the state of the running application to a persistent storage. In case of a failure, the system may be restored to the state of the latest checkpoint [32]. However, serverless functions are stateless which makes checkpointing not applicable. *Redundancy* is another proactive technique that performs replication of the idempotent functions. This technique improves the system reliability, however it induces higher cost as trade-off. In terms of execution, the successor functions of the FC can start immediately after one replica has finished execution [33] or after all replicas finish execution [34].

However, all instances of a single function may fail if, for example, some data input requires more memory than the function is assigned. In such case, none of the previously mentioned techniques will succeed. One possible example is a recursive implementation that may fail due to reaching the memory limit faster, but an iterative implementation may work for a specific input. Such cases need another proactive technique, called *design diversity*, where different versions of a function are invoked concurrently [35]. This technique increases the development effort for creating other implementations; however it improves further resilience of FCs. Implementations may be developed in different programming languages or with improved algorithms and such diverse functions be orchestrated in an FC.

C. Function Failures Affect the Entire FC

Functions failures, caused by various reasons, affect the entire FC [36]. Some functions may not even start if the input JSON file exceeds the size limit or there is an authentication error. Functions may stop their execution before completion, most commonly due to an invalid input. Other types of abnormal execution is if a function execution blocks or runs longer than its maximum duration, or the output of a function exceeds the limit. In all cases, the output JSON object is either not completely generated or contains `error` key instead of the correct ones, which may cause all successor functions of the FC to fail or to produce invalid results. Any of the above-mentioned failures affect the execution of FCs and may lead to an error state, such as wrong results produced by the entire FC, or correct result produced after the deadline constraint.

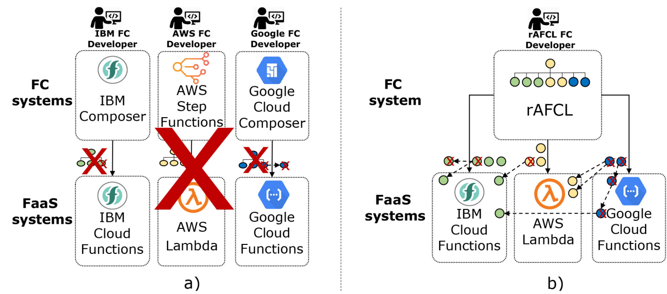


Fig. 1. Existing FC systems offer resilient techniques within the same region (a), while *rAFCL* in federated clouds (b).

D. Resilient Techniques in Existing FC Systems

Fig. 1 a) presents three scenarios for which the execution of the entire FC may fail using the existing resilient techniques of FC systems. Firstly, as the example for IBM shows, due to the provider limitations (e.g., input size limitation), the entire FC cannot complete even with sporadic failures. The reason is that when a function fails the specific number of retries and the try-catch block also fails for the same reason, there is no support for diversity on another region of the same or another provider that supports more relaxed limitation. Secondly, even if the cloud provider offers alternate resource, still, that resource may also fail, leading again to a failure of the entire FC. Thirdly, the network communication or the entire data center may be unreachable, which locks the user in the hands of the cloud provider's region. Similarly, the expired session token or any other failure in authentication may lead to massive failures.

E. Towards Resilient Techniques in Federated Clouds

The lack of advanced support for resilience of FCs motivated us to introduce *rAFCL*, which offers various types of resilient techniques, including alternate functions, function retry, diversity, and redundancy in federated clouds. As shown in Fig. 1 b), *rAFCL* allows the FC developer to specify multiple alternative functions, some of which on another region of the same cloud provider or even on another cloud provider. Moreover, *rAFCL* may run multiple replicas concurrently or sequentially across various regions and providers. Finally, *rAFCL* supports diversity, or may run various alternative functions from the same semantics as the primary function, if the latter fails. All these resilient techniques are applicable within a single language Abstract Function Choreography Language (AFCL) [3], instead of developing multiple versions of the FC in specific language of each provider. This means that *rAFCL* supports resilience for federated clouds.

IV. *rAFCL* Implementation

This section describes the *rAFCL* approach and the system architecture of the current *rAFCL* implementation. The implemented proactive and reactive resilient techniques are also explained. Finally, the extensions to AFCL that were needed to support the new resilient techniques are also detailed.

A. *rAFCL* Approach

The general *rAFCL* approach is to support all resilient techniques described in Section III and to provide resilience in federated clouds. Firstly, *rAFCL* offers FC developers to specify the number of retries for each primary function. Secondly, a backup alternative strategy with multiple subsequent alternative plans is introduced for each primary function of the FC. Each alternative plan comprises the number of concurrent invocations of replicas, alternative functions, and diversity from the primary function.

Each primary function may have multiple alternative functions, which may be a part of an alternative plan. These alternative functions can be implemented in any programming language, deployed with arbitrary amount of memory in any region of all top five FaaS systems. Having multiple alternative functions that are hosted in different regions of many providers, may significantly increase reliability of functions and the entire FC. The only requirement for each alternative function is to have the same semantics (functionality), data inputs and data outputs. This is required to retain the consistency in the FC execution. Unlike the FC systems of all well-known cloud providers, which support resilient techniques for FCs that are limited to a single region, to the best of our knowledge, *rAFCL* is the first platform that supports resilience in federated clouds. The main innovation is the possibility to invoke alternative functions that can be deployed in federated clouds, rather than in the same region as the primary function only.

In order to allow a unified resilience in federated clouds, *rAFCL* disables the standard resilient settings on every individual cloud provider. This way *rAFCL* creates a homogeneous starting point and re-configures resilience from scratch to guarantee uniform behaviour across all cloud providers and their regions. For instance, *rAFCL* automatically deactivates function retries on the cloud provider side to make sure all FaaS systems are behaving the same under the control of the FC developer. The FC developer can then specify a maximum number of retries for each function in AFCL, which is applicable regardless on which FaaS system the primary function runs. *rAFCL* then retries the primary function as many times as specified, before starting the alternative strategy.

If the primary function fails during execution, or does not complete within the maximum duration timeout even after the specified number of retries, then the first alternative plan of the alternative strategy for that primary function is executed. An alternative plan is a collection of one or more alternative functions of the primary function. When any of these alternative functions returns (finishes with success), *rAFCL* cancels the other invoked alternative functions of the same alternative plan. Subsequently, it ignores the other successive alternative plans from the alternative strategy and the rest of the FC continues to run with the other successor primary functions following the control-flow of the FC. In case all alternative functions from the first alternative plan fail, *rAFCL* proceeds with execution of alternative functions of the second alternative plan. This process of sequential invocation of the next alternative plan continues until all alternative plans of the alternative strategy are invoked. If all alternative plans of the alternative strategy fail, *rAFCL* returns an exception. Similar like the top five cloud providers, *rAFCL* supports resilience for idempotent functions, but with *at least once* function execution model in federated clouds.

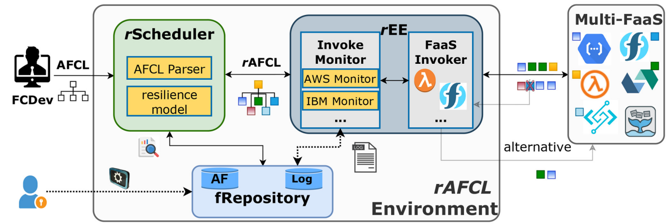


Fig. 2. *rAFCL* system architecture.

Another innovative resilient technique is the specification of latest start time (LST), latest finish time (LFT), and maximum duration for each function. These constraints include the primary function and all alternative functions within the alternative strategy. This feature is very important for the functions that belong to the critical path of the FC because any delay of those functions directly affects the makespan of the entire FC. In case these constraints are not met, then *rAFCL* cancels all started functions and returns an exception to save further costs.

B. *rAFCL* System Architecture

Fig. 2 depicts the system architecture of *rAFCL*. It consists of *rAFCL* scheduler (*r*Scheduler), function repository (*fRepository*) and resilient enactment engine (*rEE*), which comprises *Invoke Monitor* and *FaaS Invoker*. *rAFCL* may run FCs across multi-FaaS federated cloud.

The input to *rAFCL* is the specified FC and the required availability for each function. For example, FC developers can choose how often the individual primary function should be retried and the required availability for each alternative plan of the alternative strategy.

*f*Repository stores two kind of data for each function. Firstly, the *rAFCL* administrator configures *f*Repository with all cloud providers, their regions, and all deployed functions. Secondly, *rAFCL* administrator stores resource link for all primary functions and their alternative functions. Finally, for each function invocation, *f*Repository stores execution logs (invocation time, duration, return time, and status message) and availability during those executions.

*r*Scheduler uses availability data for alternative functions to create an alternative strategy for each primary function during runtime. It then generates alternative plans that reach the required availability by including multiple alternative functions across multiple regions and cloud providers concurrently. This greatly reduces the probability that all alternative functions fail. To reduce costs, *r*Scheduler generates alternative plans using a greedy algorithm that minimizes the number of alternative functions needed to reach the required availability within the alternative plans. Finally, *r*Scheduler creates an optimized AFCL file (*r*AFCL) for the FC, which includes alternative strategy for every primary function.

Afterwards, *r*Scheduler passes the optimized *r*AFCL file to *r*EE, which parses it and creates an executable workflow based on its control- and data-flow. *r*EE then runs the executable workflow by passing all primary functions to the Invoke Monitor module. This module submits the ready primary functions to the FaaS Invoker module, which invokes them on the specified FaaS system. Invoke Monitor comprises several monitors for each FaaS system, which monitors the execution of all primary functions that are specified with resilience. The invocation process is monitored and the information is saved in the *f*Repository. If a primary function fails, then Invoke Monitor retries the execution based on the specified number of retries. If all retries fail, then Invoke Monitor proceeds with the alternative strategy which contains alternative plans calculated by *r*Scheduler earlier. Moreover, Invoke Monitor monitors if the specified time constraints (e.g., LST) are met and raises exceptions in case they are violated. By saving data about all previous invocations in *f*Repository, *rAFCL* updates the availability of each function. Instead of supporting resilient techniques for each FC system individually, FaaS Invoker interacts with FaaS systems directly to support resilient techniques in federated clouds.

The overall *rAFCL* is publicly available on Github.[2] This repository contains the parser for AFCL to extract all primary functions, and follows the control- and data-flow between functions. Both modules of *r*EE are implemented in separate Java projects. The `InvokeMonitor` interface is a part of the `FTjFaaS`[3] project and monitors the execution of a function on the specified provider. The `FaaSInvoker` interface is a part of the `jFaaS` project,[4] which is a portable interface that invokes functions on various cloud providers and is a part of our existing *xAFCL* FC management system.

C. Resilience Specification in FCs

1) Function Description in AFCL: AFCL is a YAML-based language and allows FC developers to orchestrate functions by connecting them with different control- and data-flow constructs. AFCL offers a reach set of control-flow constructs including `parallelFor`, `parallel`, `switch`, or `if` and data-flow constructs including data distribution with indexing or blocking. We refer the reader to our previous paper [3] for details about how an FC can be developed using AFCL.

AFCL offers FC developers to specify `properties` of primary functions (e.g., the location of the function). The location of each primary function is specified in the value of the built-in key `resource` in the properties, as it is presented in Fig. 3. The same field is used regardless where the function is deployed, which allows to run and monitor FCs in federated clouds. AFCL also offers FC developers to setup `constraints` (e.g., number of retries), which are used for specifying various settings that are important during runtime.

2) Extensions in AFCL for Resilience: Each function of an FC may be configured with a `constraints` field, which is

[2]https://github.com/sashkoristov/enactmentengine/

[3]https://github.com/sashkoristov/FTjFaaS

[4]https://github.com/sashkoristov/jFaaS/

```
properties:
  - name: "resource"
    value: "java:arn:aws:lambda:eu-west-1..."  #Run on AWS
constraints:
  - name: "FT-Retries" #Retry 2 times this function.
    value: "2"
  - name: "C-latestStartingTime"
    value: "2020-03-03 18:48:05.123"
  - name: "C-latestFinishingTime"
    value: "2020-03-03 18:49:05.123"
  - name: "C-maxRunningTime"
    value: "5000"
  - name: "FT-AltStrat-requiredAvailability"
    value: "0.9" # min. avail. for each alternative plan
```

Fig. 3. Support for resilience in *rAFCL*.

parsed by *r*EE and processed accordingly. We introduced several new built-in settings for resilience in *rAFCL*. To begin with, the constraint `FT-Retries` (see Fig. 3) is a mandatory field which contains an integer to represent the number of times the primary function is retried before *rAFCL* proceeds with the alternative strategy. Further on, due to the performance variability of cloud resources and FCs [37], we introduced three novel time constraints as optional fields. These novel parameters are important for the overall execution of the FC, which are also shown in Fig. 3. They define which parameters should be monitored during execution, i.e., LST, LFT, and maximum function duration.

The `C-latestStartingTime` constraint is a timestamp that specifies LST at which a primary function has to be invoked within the FC. For example, if a function belongs to the critical path, delaying its start will cause the delay of the entire FC. In many cases, the results of the FC may be irrelevant after the deadline constraint passes. Before invoking a function, *rAFCL* checks if this constraint has been set and whether the current time is later than LST. If so, neither the primary function, nor the alternative strategy will be invoked. Instead, an exception will be thrown to save further execution costs. Similarly, the `C-latestFinishingTime` constraint may be optionally used to set LFT by which the primary function, or any of the alternative functions must return with a success. Otherwise, *rAFCL* throws an exception to save further execution costs. Finally, `C-maxRunningTime` is used to specify the maximum round trip time (5.000 ms) for which the primary function should finish, after which, *rAFCL* throws an exception.

The last constraint (`FT-AltStrat-requiredAvailability`) lets the FC developer to specify the required availability for each alternative plan. If this optional constraint is specified, *r*Scheduler will automatically generate multiple alternative plans for that primary function during runtime. In the resulting optimized *r*AFCL file, the `FT-AltStrat-requiredAvailability` constraint is removed from the YAML file and is replaced with the multiple alternative plans of the alternative strategy for that primary function. The *r*Scheduler adds one constraint with a prefix `FT-AltPlan-` and the number of each alternative plan. Fig. 4 presents an example of an alternative strategy with two alternative plans `FT-Alt-Plan-0` and

```
constraints:
  - name: "FT-AltPlan-0" #If AWS fails, retry two f on IBM
    value: "0.9879;https://jp-tok...;https://eu-gb...;"
  - name: "FT-AltPlan-1" #If both tasks (jp and gb) fail,
  # run alternatives on IBM Frankfurt and Dallas
    value: "0.9808;https://eu-de...;https://us-south...;"
```

Fig. 4. A possible output of *r*Scheduler. *r*Scheduler replaces the constraint `FT-AltStrat-requiredAvailability` from Fig. 3 with calculated alternative plans.

`FT-Alt-Plan-1`. The `value` attribute of each alternative plan is filled with the total calculated availability and the resource links for all alternative functions that should be invoked in that alternative plan. For instance, the calculated availability for the alternative plan `FT-Alt-Plan-0` is 0.9879. If the primary function that runs on AWS fails, then both alternative functions (in IBM Tokyo and London) will be invoked concurrently as a part of the alternative plan `FT-Alt-Plan-0`. If they both fail, *rAFCL* proceeds with the alternative plan `FT-Alt-Plan-1`.

V. *rAFCL* Resilience Model and Scheduler

*r*Scheduler follows two design principles to satisfy required availability for each function. Firstly, *rAFCL* selects the alternative functions with the highest availability and minimum number of alternative functions within the first alternative plan, with a main goal to minimize the cost. Further on, each follow up alternative plan has at least the same number of alternative functions. The second principle is that one alternative function may be used in maximum one alternative plan of the alternative strategy. The reason is that if an alternative function does not finish after the specified number of retries, it is highly probable that it fails again in another alternative plan.

A. *rAFCL Resilience Model*

Let $\mathbb{F} = \bigcup_{j=1}^{j=N} \{f_j\}$ denotes the set of all primary functions f_j of an FC with specified resilience constraints. The other functions of the FC without specification for resilience, as well as the control- and data-flow between functions are beyond the scope of this manuscript. Each function f_j may be specified with minimum required availability $a_j \in A, \forall j = 1, 2, \ldots, N$ and $\mathbb{AF}_j$ is the set of all alternative functions for each function $f_j \in F$ that can be selected in the alternative strategy. Furthermore, we denote the set of all sets AF_j for each function $\mathbb{AF} = \bigcup_{j=1}^{j=N} \{\mathbb{AF}_j\}$. Besides, we represent the method that returns the availability of the alternative function af_i as $A(af_i)$. Availability of each function depends on several factors. First, the network links between *rAFCL* and the cloud region must be up and running so that the request for the function execution reaches the cloud region and the response of the function returns to *rAFCL*. Second, the cloud platform should be up and running and provide required resources to run the function. For example, different regions may assign less powerful CPUs, which may breach the maximum function duration limitation. Finally, the code of the function should run properly and comply with the cloud infrastructure and limitations. For example, AWS Lambda offers write access to the `/tmp` folder and trying to write on another location would cause a failure. Therefore, we model availability $A(af_i)$ of an alternative function af_i in Equation (1) as a product of three availability functions (i) availability of networking $NA(af_i)$ between *rAFCL* and the cloud region where the function runs, (ii) availability of the region $RA(af_i)$ where the function runs, and (iii) availability of the code of the function $CA(af_i)$.

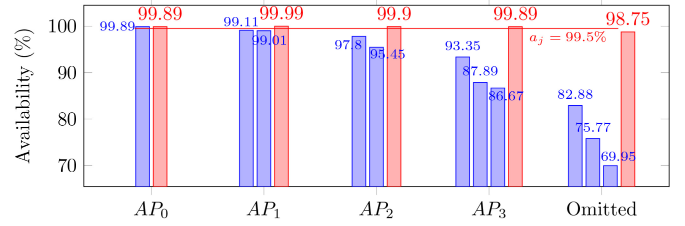


Fig. 5. The output four alternative plans (AP_0 to AP_3) and their availability (numbers in red color) from *r*Scheduler for a given single function f_j with required availability $a_j = 99.5\%$ and given 11 alternative functions af_i with their individual availability $A(af_i)$ (numbers in blue color).

$$A(af_i) = NA(af_i) \cdot RA(af_i) \cdot CA(af_i) \tag{1}$$

The current implementation of *rAFCL* measures $A(af_i)$ for each function without splitting the availability of network, region, and code.

Finally, $\mathbb{AS} = \bigcup_{j=1}^{N} \{\mathbb{AS}_j\}$ denotes the set of alternative strategies $\mathbb{AS}_j$ for each $f_j \in F$. Each alternative strategy contains at least one alternative plan AP_p, such that each alternative plan contains a subset of alternative functions of the primary function f_j. The overall availability of each alternative plan AP_p is denoted with a_p. The total availability when the first $1 \le k \le m$ alternative functions of the array $af_i, i = 1, 2, \ldots, m$ are executed concurrently is calculated based on Equation (2).

$$A(k) = 1 - \prod_{i=1}^{i=k} [1 - A(af_i)]. \tag{2}$$

If the first k alternative functions are selected for an alternative plan AP_p, then $a_p = A(k)$.

B. *rScheduler Implementation*

We implemented a greedy *r*Scheduler which determines multitude of alternative plans for each primary function specified with required availability. If the FC developer has specified the "`FT-AltStrat-requiredAvailability`" constraint for a primary function, then *r*Scheduler will proactively generate an alternative strategy for this primary function at runtime.

We illustrate the *r*Scheduler algorithm with an example. Let the input to *r*Scheduler is a single primary function f_j with required availability $a_j = 99.5\%$ for which it needs to create an alternative strategy. For instance, a function that adds a timestamp to an image. Let *r*Scheduler receives 11 alternative functions af_i as potential alternative functions of function f_j, each with individual availability $A(af_i)$, as depicted in Fig 5. For instance, the alternative functions may be deployed in *six* regions of AWS, *two* regions of IBM and *three* regions of Google, all in Europe to minimize latency.

*r*Scheduler firstly orders the alternative functions descending based on their availability, which is already done in Fig. 5. Based on the required availability $a_j = 99.5\%$, *r*Scheduler selects the first function af_1 alone within the first alternative plan AP_1 because it has a higher availability ($A(af_1) = a_1 = 99.89\%$) than the required one ($a_j = 99.5\%$). Then, *r*Scheduler selects the following two alternative functions af_2 and af_3 in the alternative plan AP_2 to achieve total availability of $a_2 = 99.99\%$ based on (2), which is higher than the required $a_j = 99.5\%$. We observe that this availability is even higher than a_1, but at the same time more expensive since AP_1 runs one function only. Similarly, *r*Scheduler places af_4 and af_5 into AP_3. Afterwards, *r*Scheduler selects the next three functions to reach the required availability a_j because the total availability of af_6 and af_7 is only $99.19\% < 99.5\%$. Finally, *r*Scheduler does not place any of af_9, af_{10}, and af_{11} in a new alternative plan because their total availability $98.75 < 99.5 = a_j$.

Note that *r*Scheduler considers the memory configuration of a function implicitly. Usually, function's availability is higher when it is assigned with more memory. Therefore, *r*Scheduler selects the function's code with more memory earlier than the same code with less memory.

C. rScheduler Formal Algorithm

Algorithm 1 presents the formal greedy *r*Scheduler algorithm. The *r*Scheduler algorithm requires the following inputs: the set of all primary function *F* of an FC that are specified with resilience constraints, the set $\mathbb{A}$ of required availability a_j for each primary function f_j, and the sets of all alternative functions $\mathbb{AF}_j$, including their availability $A(af_i)$, for each primary function. After the *r*Scheduler algorithm finishes, it provides as output a set of alternative strategies, one for each primary function f_j. Because the algorithm is greedy it tries to find the alternative plans with the lowest number of alternative functions first. Therefore, the algorithm minimizes the number of needed invocations to reach a required availability.

Algorithm 1 iterates over all primary functions (lines 2-17). For each primary function, it ranks all alternative functions in descending order based on their availability (Line 3) and initializes the starting parameters for each iteration (Lines 4-7). Afterwards, it iterates over the unassigned alternative functions (Lines 8-17) to select the first *k* alternative functions for which the total availability a_p is greater than the required availability a_j. If this is the case, the algorithm creates a new alternative plan, stores the alternative functions in it and removes them from the list (lines 11-14). If the total availability a_p of the selected *k* alternative functions is less than the required a_j, the algorithm increases *k* by one, that is, checks the total availability including the following alternative function from the ranked list (Line 16). At the end of each iteration for the primary functions, the algorithm ignores the last alternative functions whose total availability is smaller then the required one. If the algorithm finds at least one alternative plan within the alternative strategy, it includes the alternative strategy in the set of all alternative strategies (Line 17).

Algorithm 1: *r*Scheduler Greedy Algorithm

Input : $\mathbb{F} = \bigcup_{j=1}^{j=N} \{f_j\}$; // A set of functions with specified resiliency constraints
Input : $\mathbb{A} = \bigcup_{j=1}^{j=N} \{a_j\}$; // Min. availability for each f_j
Input : $\mathbb{AF} = \bigcup_{j=1}^{j=N} \{\mathbb{AF}_j\}$; // A set of $\mathbb{AF}_j$ (alternative functions) for each function $f_j \in F$
Output: $\mathbb{AS} = \bigcup_{j=1}^{N} \{\mathbb{AS}_j\}$; // a set of alt. strategies $\forall f_j \in F$

1 **Function** rAFCL($\mathbb{F}$, $\mathbb{A}$, $\mathbb{AF}$):
2 　**for** $j \leftarrow 1$ *to* N **do** // Iterate over all functions
3 　　$RANK \leftarrow Desc(\mathbb{AF}_j)$; // order all alternative functions of f_j descending based on availability
4 　　$\mathbb{AS}_j \leftarrow \emptyset$; // create an empty strategy for f_j
5 　　$m \leftarrow |\mathbb{AF}_j|$; // the number of alternative functions for f_j
6 　　$k = 1$; // initialize the number of replicas in the alt. plan
7 　　$p = 0$; // initialize the counter for f_j's alternative plans
8 　　**while** $k < m$ **do**
9 　　　$a_p = Avail(RANK)$; // Calculate joint availability of the first k functions based on (2)
10 　　　**if** $a_p \geq a_j$ **then**
11 　　　　$AP_p \leftarrow \bigcup_{r=1}^{k} \{rank_r\}$; // add the k functions in the alternative plan AP_p
12 　　　　$RANK \leftarrow \bigcup_{r=1}^{m} \{rank_r\} \setminus \bigcup_{r=1}^{k} \{rank_r\}$; // remove the first k functions from the rank
13 　　　　$\mathbb{AS}_j \leftarrow \mathbb{AS}_j \bigcup AP_p$; // store the alt. plan to the strategy
14 　　　　$p{+}{+}$; // increase the alternative plans counter
15 　　　**else**
16 　　　　$k{+}{+}$; // try with one more alternative function to reach the required availability
17 　　$\mathbb{AS} \leftarrow \mathbb{AS} \bigcup \mathbb{AS}_j$
18 　**return** *AS*;

VI. *rAFCL* EVALUATION

This section evaluates *rAFCL* and its *r*Scheduler with three complementary use cases. Firstly, we show the availability level that *rAFCL* may achieve theoretically and the cost-performance trade off if it uses various number of alternative functions distributed on 23 regions across the globe, considering all AWS regions (Section VI-A). Further on, using a synthetic version of a real business FC with complex control- and data-flow, we evaluate how sporadic functions failures affect FCs that run on AWS Step Functions FC system in one region only and FCs that run across 23 regions of AWS and IBM using our *rAFCL* (Section VI-B). Finally, we evaluate how embarrassingly parallel FCs survive massive functions failures with AWS Step Functions and our *rAFCL* (Section VI-C). Although we used only the AWS's Step Functions FC system in our evaluation, the results will be similar for all other cloud providers because their FC systems offer similar resilient techniques within a single cloud region only, as presented in Section II-A. Finally, this section also discusses *rAFCL*'s limitations.

A. Theoretical Analysis of rAFCL Resilience for a Single Function Deployed Across Multiple Regions

Before evaluating resilience of entire FCs, we analyze how different success rates (availability) of alternative functions

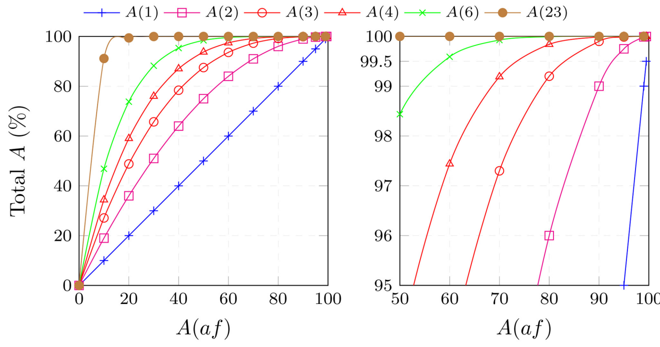


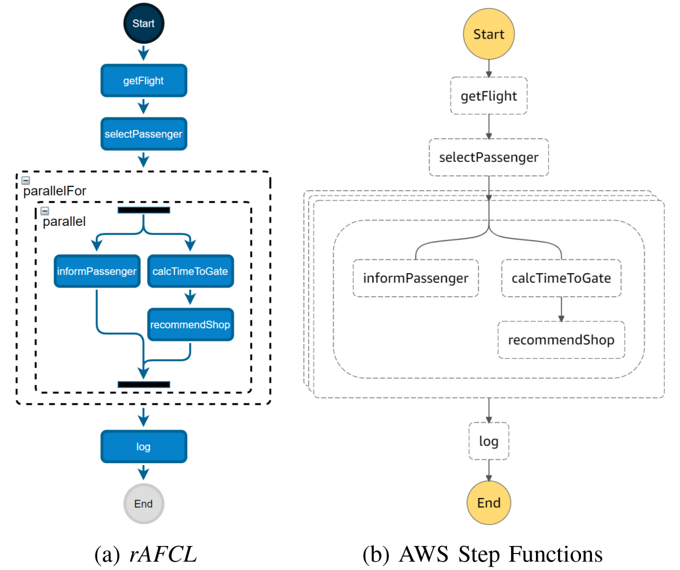
Fig. 6. Total availability $A(k)$ based on (2) for concurrent execution of various number of alternative functions *af* as a function of availability of a single alternative function $A(af)$. The left figure shows the total availability for all possible values of $A(af)$. The right figure is focused on total availability more than 95%.

and their number affect the execution costs and performance. The total theoretical availability of a system that uses multiple alternative functions for a single primary function can be calculated using (2), where availability of each alternative function can be retrieved from its historical executions. Using multiple alternative functions across the globe significantly increases availability of the entire FC, which can be observed in Fig. 6. $A(1)$ represents the availability of a single alternative function, which is the case with all cloud providers. This locks the user with the given availability of the selected provider. *rAFCL*, on the other side, allows FC developers to specify multiple alternative functions across the globe if the primary function fails. For instance, by using all six AWS regions in Europe, which have similar network latency to University of Innsbruck, Austria of up to 20 ms,[5] *rAFCL* can achieve $A(6) = 99.5\%$ availability even if availability of a single alternative function is only $A(af) = 0.6$. More realistic use case is if the alternative function has the lowest reimbursable availability of $A(af) = 95\%$ at AWS, for which *rAFCL* can achieve total availability of $A(2) = 99.75\%$ using two regions only. Moreover, $A(23)$ shows an example of availability close to 1 that can be achieved by using all 23 AWS regions across the globe. *rAFCL* may run alternatives across all regions from the other cloud providers (ca. 50).

Further on, we analyze the cost-performance trade-off of *parallel* execution of all alternative functions within a single alternative plan versus *sequential* execution of multiple alternative plans, each with a single alternative function. Namely, since the failures of functions are independent events, availability of the alternative functions is the same, regardless if they are executed as a sequence, one after the other, or concurrently. The concurrent invocation reduces the overall execution time but may increase the invocation cost compared to the sequential execution of alternatives. On the other side, the sequential execution of alternative plans minimizes the invocation cost, but may increase overall execution time. Table I shows the results of the cost-performance tradeoff as a function of the success rate used for the primary function using the sequential and parallel execution of alternative functions. We observe that all functions achieved their success rate of minimum $a_p = 99\%$. However, the way how this is achieved differs. We observe that execution time of the parallel alternatives is always better than the sequential alternatives by up to $\delta_{ET} = 52.8\%$ for the lowest function success rate $A(af_i) = 0.25$. However, for functions with high success rate, execution time is higher in parallel alternatives because the FaaS Invoker needs more time to invoke larger number of alternatives compared to sequential alternatives. On the other side, the number of invocations of alternative functions is significantly higher when they are executed concurrently compared to sequential execution. This is also emphasized for the experiment with the lowest success rate $A(af_i) = 0.25$, with increase of the cost by $\delta_c = 2.56\times$.

TABLE I
AVERAGE EXECUTION TIME (*ET*) IN SECONDS, AVERAGE INVOCATIONS OF ALTERNATIVE FUNCTIONS, AND SUCCESS RATE OF 100 EXECUTIONS OF A SINGLE PRIMARY FUNCTION WITH VARIOUS SUCCESS RATE, BOTH FOR SEQUENTIAL AND PARALLEL EXECUTION OF ALTERNATIVE FUNCTIONS. δ_{ET} AND δ_c REPRESENT THE REDUCTION OF ET AND INCREASE OF THE INVOCATION COST IN %

	Sequential			Parallel			Par. vs. seq	
$A(af_i)$	ET(s)	Inv.	$a_p(\%)$	ET(s)	Inv.	$a_p(\%)$	$\delta_{ET}(\%)$	$\delta_c(\%)$
0.25	7.75	3.74	99	3.66	13.32	100	-52.8	256.2
0.5	3.89	1.92	100	3.01	3.67	99	-22.6	91.2
0.75	2.84	1.35	99	2.65	1.75	99	-6.7	29.63
0.95	2.07	1.03	99	2.19	1.04	100	5.8	0.97



(a) *rAFCL* (b) AWS Step Functions

Fig. 7. The GCA FC.

B. How Do Sporadic Function Failures Affect FCs?

This set of experiments evaluates how FCs behave in case of sporadic failures of functions. We used the Gate Change Alert (GCA) FC [3], a public section FC to evaluate how is the FC execution affected by functions failures. We implemented the GCA FC in *rAFCL* and AWS Step Functions FC systems. Fig. 7 presents both implementations visualized with the corresponding GUIs (the FCEditor of *rAFCL*[6] and AWS Step Functions' GUI). The GCA FC

[5] https://cloudharmony.com/

[6] https://github.com/sashkoristov/FCEditor

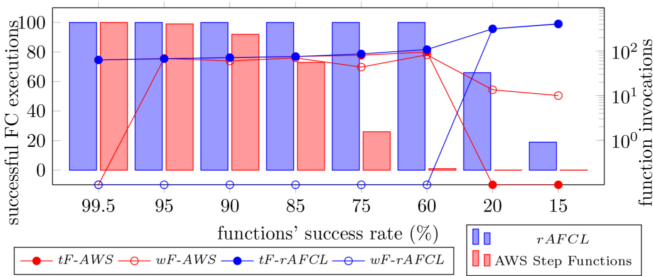


Fig. 8. The results of invoking the GCA FC 100 times with configured resilience with various success rate for each function of the GCA FC using *rAFCL* and AWS Step Functions; columns show the number of successful FC executions; *tF* denotes the average numbers of function invocations per FC (minimum is 63), while *wF* the average number of function invocations that where wasted for failed FCs.

performs several actions at an airport after a gate of a specific flight has changed. After reading the information about the new gate of the flight (`getFlight`), `selectPassenger` loads data of all passengers of that flight that are already at the airport, including their position. Thereafter, a set of additional functions are invoked in parallel for each passenger that `selectPassenger` returns. The passenger is informed about the new gate with `informPassenger`. At the same time, `calculateTimeToGate` estimates the time to gate based on the passenger positions and notifies the passenger. Afterwards, based on the passenger location, `recommendShop` recommends a few nearby shops and their special offers. Finally, after all passengers are informed, `log` is executed to log the status for all passengers.

We have bounded the parallel loop iteration count to 20 passengers at the airport, which resulted in execution of $20 \cdot 3 + 3 = 63$ functions for the overall FC. If any of these primary function does not succeed, the whole FC is considered failed. Moreover, the critical path contains all 43 functions except the 20 instances of `informPassenger`. This means that if any of these 43 functions fails and is retried and its finish time is delayed, then the makespan of the FC increases accordingly.

All functions in the GCA FC were developed to run with various success rate probability, starting from the lowest value of 15% up to 99.5%, as presented in Fig. 8. We used the same success rate probability for all functions of both FCs. We set up both FCs with the following resilient techniques. Since AWS Step Functions can run FCs only within the same region, we configured each primary function to be retried two times (default), after which the entire FC fails. Our *rAFCL*, on the other side, supports resilience techniques in federated clouds. Instead of retries in the same region, we configured 23 alternative plans for the primary function, each of which contains one alternative function in a separate region. This approach is cost-aware, but relaxes the deadline.

The goal of this experiment is to determine success rate of the GCA FC on both platforms. For both platforms, we determined the number of successful executions of the entire FC, the total numbers of function invocations per FC (minimum is 63), and the average number of wasted function invocations for failed FCs.

The results of the experiment are shown in Fig. 8. In contrast to AWS, which achieved 100% success rate for the entire FC, only in the cases when the function success rate is $A = 99.5\%$, our *rAFCL* provided a high success rate of 100% for all experiments whose functions had at least $A = 60\%$ success rate. However, the entire GCA FC finished one time on AWS Step Functions when its functions had a success rate $A = 60\%$. On average, *rAFCL* increases the success rate of entire FC by 53.45% with average increase of the average number of invocation by only 3.94% with zero wasted function invocations. For functions with smaller success rates, AWS Step Funcitons could not finish any FC, while *rAFCL* succeeded even with success rate $A = 15\%$.

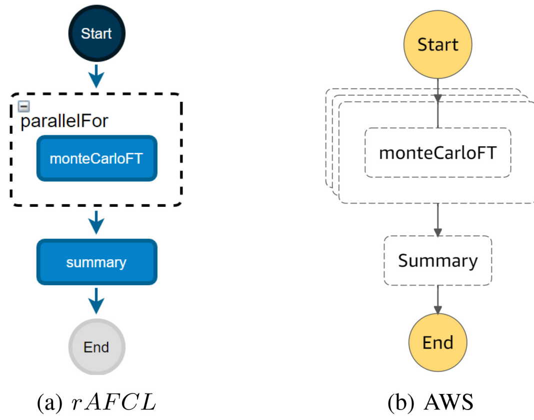
By comparing the number of invoked functions (*tF*), we observe that the additional cost for such high FC success is similar for both platforms. With a very low success rate of individual functions $A \leq 20\%$ the number of function invocation rapidly increases, but still *rAFCL* succeeds 66% of FCs, while all FCs that were executed on AWS failed. Finally, *wF* shows the number of wasted function invocations due to failure of the entire FC. We observe a high number of wasted function invocation per FC for AWS for the realistic cases when $A \geq 60\%$. The low number of wasted functions for $A \leq 20\%$ was observed because all retries of the first or the second function failed and thereby the entire FC failed. On the other hand, *rAFCL* tries 23 alternatives and succeeds, with the trade-off of much higher cost.

C. How Do Massive Functions Failures Affect FCs?

The final set of experiments was conducted to evaluate how FCs behave in case of massive failures when most of the functions, nested in a parallel loop, fail. Such failures may happen if the functions reach the limitations from the provider [7], [38]. For example, a function within a parallel loop is assigned with low amount of memory, or is invoked with a big problem size, or exceeds maximum duration time, or the function deployment is deleted or not reachable. Moreover, due to the concurrency or throughput limitations by cloud providers, the maximum number of concurrent function invocations is also limited. After exceeding the provider concurrency limit, all invocations will be rejected and thereby fail, leading to a massive failure. Finally, massive failures may happen due to failed authentication, such as an expired session or token for that cloud provider. Neither retries nor try-catch resilient techniques are useful within the same region. However, our *rAFCL* may run alternative functions on another provider where the authentication succeeds.

For this purpose, we used the embarrassingly parallel Monte Carlo algorithm, which is intensively used in HPC, cloud, and serverless computing [39], [40], [41], [42]. We implemented it as an FC, as presented in Fig. 9. The FC comprises of a parallel loop that runs multiple instances of a single primary function `monteCarloFT`, which is specified with resilience. Afterwards, the reduction `summary` function collects the data from all instances of the `monteCarloFT` function and returns the final result.

In order to evaluate massive failures in a real life scenario, we used three cloud providers AWS, IBM, and Alibaba and their regions in Frankfurt and Tokyo. We developed



(a) $rAFCL$ (b) AWS

Fig. 9. The Monte Carlo algorithm implemented as an FC in *rAFCL* and AWS Step Functions.

TABLE II
REGION, NETWORK, AND THE TOTAL AVAILABILITY FOR EACH ALTERNATIVE MONTECARLOFT FUNCTION. *r*SCHEDULER OUTPUTS TWO ALTERNATIVE PLANS FOR REQUIRED AVAILABILITY $a_j = 99.5\%$

Region	$RA(\%)$	$NA(\%)$	$A(af_i)(\%)$	AP_p	a_p
IBM Frankfurt	99	99.9	98.9	1	99.7
Alibaba Frankf.	95	99.9	94.91		
IBM Tokyo	99	95	94.05	2	99.94
Alibaba Tokyo	95	95	90.25		
AWS Tokyo	95	95	90.25		

`monteCarloFT` in Python. We used the deployment of AWS Frankfurt as a primary function, while the other five deployments as alternatives. The `summary` function was deployed in AWS Frankfurt, but only `monteCarloFT` functions were monitored due to the massive failures scenario.

Each deployment of `monteCarloFT` was configured with a predefined success rate based on the real values given of the providers' SLAs, as specified in column *RA* of Table II. We used 95% availability for AWS and Alibaba regions and 99% for IBM. Our justification is that IBM is a highly available cloud as it offers reimbursement for the highest ceiling of availability of 99.99%. For networking, we used 99.9% availability for Frankfurt since Frankfurt is closer to Austria, while for remote regions in Tokyo we used lower availability of 95%. Since the code of `MonteCarloFT` is simple, we set the code availability $CA = 100\%$. With these configurations, the alternative function in IBM Frankfurt has the highest availability, while the alternative functions in Alibaba and AWS Tokyo were configured with the lowest availability (90.25%), calculated with Equation (1).

*r*Scheduler developed an alternative strategy with two alternative plans for the primary function in AWS Frankfurt with specified required availability $a_j = 99.5\%$. The first alternative plan included the alternative functions deployed in IBM and Alibaba Frankfurt with calculated availability of $a_p = 99.7\%$. The second alternative plan included the other three functions all deployed in the Tokyo region of all evaluated providers with calculated availability of $a_p = 99.94\%$. With above-described configuration we achieved realistic sporadic failures.

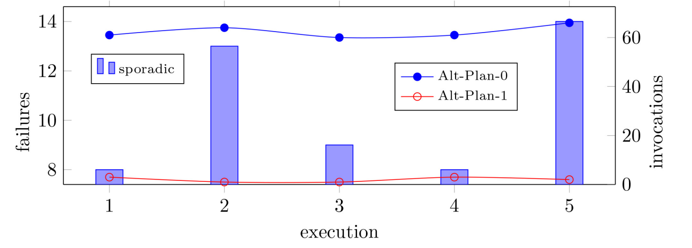


Fig. 10. Failed primary (sporadic) and invoked alternative functions of the Monte Carlo FC running on *rAFCL*.

This experiment tackles the concurrency limit of cloud providers. All top five cloud providers AWS, Google, Microsoft, Alibaba, and IBM report that they limit their FaaS systems to 1000 concurrent functions. However, several researchers reported that the real concurrency limit is in fact smaller because of several reasons. First, FaaS systems have other limitations, such as the total number of running functions within a second, which means that the FaaS system, such as Alibaba, will fail many short-running functions [13]. Second, Sampe *et al.* [8] reported huge delay of 40 s to invoke 1000 functions on IBM, which may breach the deadline constraint of the FC. Therefore, based on the reported value of Wang *et al.* [38] that AWS supports 200 concurrent invocations, we configured the primary `monteCarloFT` function in AWS Frankfurt to fail all instances in the parallel loop once the concurrency limit of 200 is reached. For this purpose, we added an input parameter *funcId* which overwrites the availability from Table II to 0 for all functions that are invoked with *funcId* > 200. This means that two kind of failures may happen. Firstly, from the first 200 functions, some sporadic number of functions fail, based on the specified availability in Table II. Secondly, all functions with *funcId* > 200 fail.

We ran both implementations of the Monte Carlo FC in AWS Frankfurt. For statistically relevant observations, we repeated each execution five times, similar as done by Casanova *et al.* [43]. As expected, all executions on AWS Step Functions failed because all functions with *funcId* $>$ 200 failed after default times of retrying. However, for each failed `monteCarloFT` function, *rAFCL* activates the alternative strategy. Fig. 10 shows the number of failed primary functions and invoked alternative plans for the Monte Carlo FC executed with *rAFCL*. We observe that the average sporadic failures are as expected (ca. 10), and together with massive failures, *rAFCL* invoked 62.4 alternative functions on average. On average, two alternative functions of the first alternative plan also failed, for which the follow up alternative plan was successful. We can conclude that the scheduled plan of *rAFCL* succeeded to achieve the required availability of $a_j = 99.5\%$, even for massive failures.

The evaluated massive failures that are initiated by concurrency limitations of the cloud providers may never happen, if users run FCs with a small problem size. Another option to avoid concurrency limitations is if the FC management system controls the concurrency of the parallel loops. *rAFCL* offers such option with another constraint `concurrency` within a parallel loop, which expects a positive integer number that

specifies the number of active threads such that each thread manages one function invocation. However, although this constraint eliminates the risk of massive failures, still, it affects the overall FC makespan.

D. rAFCL Limitations

To the best of our knowledge, *rAFCL* is the only middleware platform that overcomes resilience limitations of each individual FaaS system by unifying the maximum allowed limitations of each cloud provider, thereby preventing massive failures in federated clouds and increasing availability. The trade-off for high availability of *rAFCL* is potentially increased cost and higher execution time. However, resilient techniques of related work do not recover from a permanent failure of a function, which causes the entire FC to fail.

*r*Scheduler may create alternative plans automatically based on its greedy algorithm. Still, the current implementation of *r*Scheduler provides the alternative plans proactively and does not change them in case of massive failures. Moreover, *rAFCL* requires that all alternative functions are already deployed to know their ARN (Amazon Resource Names) or URL and to be able to run them. Such process of developing and deploying multiple alternative functions across multiple regions and FaaS systems may be a tedious operation. However, several techniques and tools exist to alleviate this process. Developers can deploy the same function in other regions of the same FaaS system by running a simple script written for serverless.com, Terraform, or Cloudify [44]. Moreover, the Serverless Application Analytics Framework (SAAF) [45] allows developers to develop functions in different programming languages only once and then encapsulate it with specific wrappers (handlers) for each FaaS system.

VII. Conclusion and Future Work

We developed *rAFCL*, a FaaScinating resilient platform that supports software developers to build and run function choreographies on multiple FaaS systems with resilience. In contrast to existing approaches, *rAFCL* can exploit the benefits of multiple alternatives of primary functions across all regions of multiple FaaS systems. Based on the specified required availability, the built-in scheduler calculates during runtime alternative functions that will be invoked in case the primary functions of the FC fail.

rAFCL can significantly improve availability of a single function and the entire FC. The *rAFCL* resilience model can successfully finish an entire FC even if the success rate of its functions is only 60%, increasing on average, the success rate of entire FC by 53.45% with average increase of the average number of invocation by only 3.94% with zero wasted function invocations, compared to running FCs on AWS Step functions. Moreover, *rAFCL* can successfully recover from massive failures by invoking many alternative functions to other regions of the same cloud provider and to any other provider. We have experimentally confirmed this with an FC that runs the primary functions on one region (AWS Lambda Frankfurt), while alternative functions on five other regions on three cloud providers (AWS Lambda, IBM Cloud Functions, and Alibaba Function Compute). *rAFCL* survived even when both alternative functions in the latter two cloud regions failed.

We showed experimentally that *rAFCL* achieves availability of almost 1 when running realistic FC across many providers. Potentially, *rAFCL* can run alternative functions of FCs across more than 50 regions, which also improves scalability by overcoming concurrency limitation of individual region. Moreover, the number of concurrent executions of functions, as well as the number of alternative functions can be increased even further by extending the *rAFCL* enactment engine with other public FaaS systems (e.g., CloudFlare) or private resources with open source serverless frameworks.

We plan to extend *r*Scheduler by a multi-objective algorithm that considers makespan and cost as an addition to availability. Moreover, we will develop AI-based monitoring module in *rAFCL*, which will proactively predict massive failures and adapt both primary functions and alternative strategies during runtime accordingly.

References

[1] D. Jackson and G. Clynch, "An investigation of the impact of language runtime on the performance and cost of serverless functions," in *Proc. Int. Conf. Utility Cloud Comput. Companion*, 2018, pp. 154–160.

[2] S. Nastic *et al.*, "A serverless real-time data analytics platform for edge computing," *IEEE Internet Comput.*, vol. 21, no. 4, pp. 64–71, Jul. 2017.

[3] S. Ristov, S. Pedratscher, and T. Fahringer, "AFCL: An abstract function choreography language for serverless workflow specification," *Future Gener. Comput. Syst.*, vol. 114, pp. 368–382, Jan. 2021.

[4] B. E. Helvik, P. Vizarreta, P. E. Heegaard, K. Trivedi, and C. Mas-Machuca, *Modelling of Software Failures*. Cham, Switzerland: Springer Nature Switzerland AG, 2020, pp. 141–172.

[5] W. Li, X. Sun, K. Liao, Y. Xia, F. Chen, and Q. He, "Maximizing reliability of data-intensive workflow systems with active fault tolerance schemes in cloud," in *Proc. Int. Conf. Cloud Comput.*, 2020, pp. 462–469.

[6] T. Guedes, L. A. Jesus, K. A. C. S. Ocaña, L. M. A. Drummond, and D. de Oliveira, "Provenance-based fault tolerance technique recommendation for cloud-based scientific workflows: A practical approach," *Clust. Comput.*, vol. 23, no. 1, pp. 123–148, 2020.

[7] P. G. López, M. Sánchez-Artigas, G. París, D. B. Pons, ÃĄ. R. Ollobarren, and D. A. Pinto, "Comparison of FaaS orchestration systems," in *Proc. Int. Conf. Utility Cloud Comput. Companion*, 2018, pp. 148–153.

[8] J. Sampé, G. Vernik, M. Sánchez-Artigas, and P. García-López, "Serverless data analytics in the IBM cloud," in *Proc. 19th Int. Middlew. Conf. Ind.*, Rennes, France, 2018, pp. 1–8.

[9] T. Gomes *et al.*, "A survey of strategies for communication networks to protect against large-scale natural disasters," in *Proc. 8th Int. Workshop Resilient Netw. Des. Model. (RNDM)*, 2016, pp. 11–22.

[10] M. Tornatore *et al.*, "A survey on network resiliency methodologies against weather-based disruptions," in *Proc. Int. Workshop Resilient Netw. Des. Model. (RNDM)*, 2016, pp. 23–34.

[11] C. M. Machuca *et al.*, "Technology-related disasters: A survey towards disaster-resilient software defined networks," in *Proc. Int. Workshop Resilient Netw. Des. Model.*, 2016, pp. 35–42.

[12] M. Furdek *et al.*, "An overview of security challenges in communication networks," in *Proc. Int. Workshop Resilient Netw. Des. Model.*, 2016, pp. 43–50.

[13] S. Ristov, S. Pedratscher, and T. Fahringer, "XAFCL: Run scalable function choreographies across multiple FaaS systems," *IEEE Trans. Services Comput.*, early access, Nov. 16, 2021, doi: 10.1109/TSC.2021.3128137.

[14] S. Hong, A. Srivastava, W. Shambrook, and T. Dumitras, "Go serverless: Securing cloud via serverless design patterns," in *Proc. Workshop Hot Topics Cloud Comput. (HotCloud)*, Boston, MA, USA, 2018, p. 11. [Online]. Available: https://dl.acm.org/doi/10.5555/3277180.3277191

[15] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows: Experiments with HyperFlow, AWS lambda and Google cloud functions," *Future Gener. Comput. Syst.*, vol. 110, pp. 502–514, Sep. 2020.

[16] B. Balis, “HyperFlow: A model of computation, programming approach and enactment engine for complex distributed workflows,” *Future Gener. Comput. Syst.*, vol. 55, pp. 147–162, Feb. 2016.

[17] A. Aske and X. Zhao, “Supporting multi-provider serverless computing on the edge,” in *Proc. Int. Conf. Parallel Process. Companion*, 2018, pp. 1–6.

[18] S. Risco, G. Moltó, D. M. Naranjo, and I. Blanquer, “Serverless workflows for containerised applications in the cloud continuum,” *J. Grid Comput.*, vol. 19, no. 3, pp. 1–18, 2021.

[19] A. F. Baarzi, G. Kesidis, C. Joe-Wong, and M. Shahrad, “On merits and viability of multi-cloud serverless,” in *Proc. ACM Symp. Cloud Comput.*, New York, NY, USA, 2021, pp. 600–608.

[20] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, “Occupy the cloud: Distributed computing for the 99%,” in *Proc. Symp. Cloud Comput.*, 2017, pp. 445–451.

[21] A. John, K. Ausmees, K. Muenzen, C. Kuhn, and A. Tan, “Sweep: Accelerating scientific research through scalable serverless workflows,” in *Proc. Int. Conf. UCC Companion*, 2019, pp. 43–50.

[22] D. Barcelona-Pons, P. Sutra, M. Sánchez-Artigas, G. París, and P. García-López, “Stateful serverless computing with crucial,” *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 3, p. 39, Mar. 2022.

[23] L. R. D. Carvalho and A. P. F. de Araújo, “Remote procedure call approach using the Node2FaaS framework with terraform for function as a Service,” in *Proc. Int. Conf. Cloud Comput. Serv. Sci.*, 2020, pp. 312–319.

[24] S. Ristov, S. Pedratscher, J. Wallnoefer, and T. Fahringer, “DAF: Dependency-aware FaaSifier for node.js monolithic applications,” *IEEE Softw.*, vol. 38, no. 1, pp. 48–53, Jan./Feb. 2021.

[25] G. Zheng and Y. Peng, “Globalflow: A cross-region orchestration service for serverless computing services,” in *Proc. IEEE CLOUD*, 2019, pp. 508–510.

[26] M. Yu, T. Cao, W. Wang, and R. Chen, “Restructuring serverless computing with data-centric function orchestration,” 2021, *arXiv:2109.13492*.

[27] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, “Sequoia: Enabling Quality-of-Service in serverless computing,” in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 311–327.

[28] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, “Wukong: A scalable and locality-enhanced framework for serverless parallel computing,” in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 1–15.

[29] J. Kijak, P. Martyna, M. Pawlik, B. Balis, and M. Malawski, “Challenges for scheduling scientific workflows on cloud functions,” in *Proc. IEEE 11th Int. Conf. Cloud Comput. (CLOUD)*, 2018, pp. 460–467.

[30] A. Das, A. Leaf, C. A. Varela, and S. Patterson, “Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications,” in *Proc. Int. Conf. Cloud Comput. (CLOUD)*, 2020, pp. 609–618.

[31] M. A. Mukwevho and T. Celik, “Toward a smart cloud: A review of fault-tolerance methods in cloud systems,” *IEEE Trans. Services Comput.*, vol. 14, no. 2, pp. 589–605, Mar./Apr. 2021.

[32] S. Ristov, T. Fahringer, D. Peer, T.-P. Pham, M. Gusev, and C. Mas-Machuca, “Resilient techniques against disruptions of volatile cloud resources,” in *Guide to Disaster-Resilient Communication Networks*. Cham, Switzerland: Springer Nature Switzerland AG, 2020, pp. 379–400.

[33] W. Cirne, F. Brasileiro, D. Paranhos, L. F. W. Góes, and W. Voorsluys, “On the efficacy, efficiency and emergent behavior of task replication in large distributed systems,” *Parallel Comput.*, vol. 33, no. 3, pp. 213–234, Apr. 2007.

[34] A. Benoit, M. Hakem, and Y. Robert, “Fault tolerant scheduling of precedence task graphs on heterogeneous platforms,” in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2008, pp. 1–8.

[35] P. E. Ammann and J. C. Knight, “Data diversity: An approach to software fault tolerance,” *IEEE Trans. Comput.*, vol. 37, no. 4, pp. 418–425, Apr. 1988.

[36] J. Ejarque, M. Bertran, J. Á. Cid-Fuentes, J. Conejero, and R. M. Badia, “Managing failures in task-based parallel workflows in distributed computing environments,” in *Proc. Eur. Conf. Parallel Process.*, 2020, pp. 411–425.

[37] S. Ristov, R. Mathá, and R. Prodan, “Analysing the performance instability correlation with various workflow and cloud parameters,” in *Proc. Euromicro Int. Conf. PDP*, 2017, pp. 446–453.

[38] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, “Peeking behind the curtains of serverless platforms,” in *Proc. USENIX Annu. Techn. Conf.*, Boston, MA, USA, 2018, pp. 133–145.

[39] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, “On the FaaS track: Building stateful distributed applications with serverless architectures,” in *Proc. 20th Int. Middlew. Conf.*, Davis, CA, USA, 2019, pp. 41–54.

[40] S. Eismann, J. Grohmann, E. van Eyk, N. Herbst, and S. Kounev, “Predicting the costs of serverless workflows,” in *Proc. Int. Conf. Perform. Eng.*, 2020, pp. 265–276.

[41] J. Sampe, P. Garcia-Lopez, M. Sanchez-Artigas, G. Vernik, P. Roca-Llaberia, and A. Arjona, “Toward multicloud access transparency in serverless computing,” *IEEE Softw.*, vol. 38, no. 1, pp. 68–74, Jan./Feb. 2021.

[42] D. Barcelona-Pons and P. García-López, “Benchmarking parallelism in FaaS platforms,” *Future Gener. Comput. Syst.*, vol. 124, pp. 268–284, Nov. 2021.

[43] H. Casanova *et al.*, “Developing accurate and scalable simulators of production workflow management systems with Wrench,” *Fut. Gener. Comput. Syst.*, vol. 112, pp. 162–175, Nov. 2020.

[44] L. R. de Carvalho and A. P. F. de Araujo, “Performance comparison of terraform and cloudify as multicloud orchestrators,” in *Proc. Int. Symp. Clust. Cloud Internet Comput. (CCGRID)*, 2020, pp. 380–389.

[45] R. Cordingly *et al.*, “The serverless application analytics framework: Enabling design trade-off evaluation for serverless software,” in *Proc. Int. Workshop Serverless Comput. (WoSC)*, 2020, pp. 67–72.

Sasko Ristov received the Ph.D. degree in computer science from Ss. Cyril and Methodius University, Skopje, North Macedonia, where he was an Assistant Professor from 2013 to 2017. He is a Postdoctoral University Assistant with the University of Innsbruck, Austria. His research interests include performance modeling and optimization of distributed systems, in particular workflow applications and serverless computing.

Dragi Kimovski received the doctoral degree from TU Sofia, Bulgaria, in 2013. He is a Tenure Track Postdoctoral Researcher with the Institute of Information Technology, University of Klagenfurt, Austria. He was an Assistant Professor with University for Information Science and Technology, Ohrid, North Macedonia. His research interests include distributed systems and multiobjective optimization.

Thomas Fahringer (Member, IEEE) received the Ph.D. degree from the Vienna University of Technology in 1993. He has been a Full Professor of Computer Science with the Institute of Computer Science, University of Innsbruck, Austria, since 2003. 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 ACM.