UNIVERSITY OF MISKOLC

FACULTY OF MECHANICAL ENGINEERING AND INFORMATICS

DISSECT-CF-FaaS: A Simulation Environment for Simulating Functions-as-a-Service

PhD dissertation

Author

Dilshad Hassan Sallo

MSc in Advanced Computer Science with a Specialisation in Software Technology

'JÓZSEF HATVANY' DOCTORAL SCHOOL
OF INFORMATION SCIENCE, ENGINEERING AND TECHNOLOGY

HEAD Of DOCTORAL SCHOOL
Prof. Dr. Jenő SZIGETI

ACADEMIC SUPERVISOR
Prof. Dr. Gábor Kecskeméti

Miskolc, 2024

Declaration

The author hereby declares that this thesis has not been submitted, either in the same or in a different form, to this or to any other university for obtaining a PhD degree. The author confirms that the submitted work is his own and the appropriate credit has been given where reference has been addressed to the work of others.

Dilshad H. Sallo

Acknowledgments

First and foremost I want to gratefully and sincerely thank Almighty God for assisting me to complete this project. I owe special thanks to Prof. Dr. Gabor Kecskemeti for his excellent guidance, patience and most importantly, his friendship during doing this project.

I would like to express special thanks to the Kurdistan Regional Government to fully support me during my study. I thank my parents, my friends, and all other family members for encouraging and supporting me throughout this study.

Finally, I dedicate this work to my sons Shad and Shan, wishing them an unclouded childhood and power to make their dreams come true.

Dilshad H. Sallo

Abstract

Serverless computing is a new style of delivering cloud services in an easy-environment that abstracts a user from the burden of managing resources and infrastructure. This computing paradigm is adopted by several commercial providers to offer elasticity to develop applications with a fine-grained cost model. Despite the widespread use of Functions-as-a-Service (FaaS) within the research community, conducting experiments, evaluating the performance and provisioning policies in commercial providers remains a difficult endeavour and costly. Therefore, simulators have been opted as an alternative solution. Unfortunately, there are no established simulation frameworks that can support research focussing on the challenges accompanying serverless computing. The existing serverless simulators focus on specific functionality or aspects, but they could not meet the expectations compared to the services offered by commercial providers.

Therefore, this dissertation focuses on introducing a comprehensive serverless environment to the DISSECT-CF simulator to enable simulating realistic FaaS solutions and evaluating scenarios reliant on the concepts of the serverless paradigm. This environment is capable of generating realistic FaaSs and imitating serverless providers' in terms of cost-model, associating triggers, resource constraints, and customising the configurations of FaaSs. Moreover, it reveals the internal behaviour of provisioned resources that occur during simulation by extracting performance metrics in a parallel manner. The evaluation shows that our environment is able to simulate and evaluate FaaSs scenarios by properly reflects providers' policies and captures the behaviour of FaaS component.

Contents

1 Introduction **13**
- 1.1 Research goals . . . 14
- 1.2 Dissertation guide . . . 14

2 Background and Literature Review **16**
- 2.1 Introduction . . . 16
- 2.2 Background . . . 17
  - 2.2.1 Serverless computing . . . 17
    - 2.2.1.1 Serverless computing terminology . . . 17
    - 2.2.1.2 Serverless computing use cases . . . 18
    - 2.2.1.3 Serverless computing providers . . . 19
    - 2.2.1.4 Open-source serverless computing frameworks . . . 21
    - 2.2.1.5 Serverless and other computing paradigms . . . 22
    - 2.2.1.6 Virtualization, containerization and serverless computing . . . 22
  - 2.2.2 Traces . . . 23
  - 2.2.3 Simulation environments . . . 23
  - 2.2.4 Evaluation methodologies . . . 25
- 2.3 Related works . . . 26
  - 2.3.1 Cloud simulators . . . 26
    - 2.3.1.1 Implementation language . . . 28
    - 2.3.1.2 Simulation type . . . 28
    - 2.3.1.3 Simulation input . . . 28
    - 2.3.1.4 Support extension . . . 28
    - 2.3.1.5 Lack of parallelsation . . . 29

2.3.2 Serverless simulators . . . 30
2.3.3 Discussion and concluding remarks . . . 32
2.3.4 Overview of DISSECT-CF simulator . . . 32
2.4 Summary . . . 34

3 Generating Realistic Serverless Traces 35
3.1 Introduction . . . 35
3.2 Architecture of the trace generator and serverless model . . . 36
3.2.1 Configuration setup layer . . . 37
3.2.2 FaaS creation layer . . . 37
3.2.3 Serverless management layer . . . 38
3.3 Generating realistic traces from Azure Functions dataset . . . 38
3.3.1 Generating invocations, execution times and allocated memory 39
3.3.2 Evaluation of generator approach . . . 42
3.4 Improving our previous generator approach . . . 44
3.4.1 Limitations of previous approach . . . 44
3.4.2 Architecture of enhanced approach . . . 45
3.4.2.1 Improving the percentiles of generated traces . . . 46
3.4.2.2 Scaling workload with real users' behavior . . . 49
3.4.2.3 Converting generated trace to standard formats . . . 50
3.4.3 Evaluation of improved percentiles . . . 51
3.4.4 Evaluation of users' behavior . . . 52
3.4.5 Evaluation of converting approach . . . 55
3.4.5.1 DISSECT-CF . . . 55
3.4.5.2 Other simulators . . . 56
3.4.5.3 SimFaaS . . . 58
3.5 Summary . . . 59

4 An Extension of DISSECT-CF to Simulate Function-as-a-Service 62
4.1 Introduction . . . 62
4.2 Extending our previous model . . . 63
4.3 Proposed architecture . . . 63
4.3.1 Configuration setup layer . . . 64
4.3.2 FaaS Creation layer . . . 65
4.3.3 Serverless management layer . . . 65
4.3.4 Cost modeling and statistics layer . . . 66
4.4 An illustrative walk through of our extensions . . . 67
4.5 Experiments . . . 70

4.5.1 Evaluation cost model . . . 70
4.5.2 Evaluation of provisioned concurrency . . . 71
4.5.3 Evaluation of trigger . . . 72
4.5.4 Evaluation of performance metrics . . . 72
4.6 Summary . . . 74

5 Parallel Event System to Reveal the Internal Behavior of our Serverless Environment 75
5.1 Introduction . . . 75
5.2 PDES issues and challenges . . . 76
5.3 Prominence of recurrent events . . . 77
5.4 The parallelisation of simultaneous events . . . 80
5.5 Evaluation of the parallel event system . . . 82
5.5.1 Validation of the parallel event system . . . 82
5.5.2 Performance of the parallel event system . . . 84
5.6 Evaluation of our serverless environment using parallel event system . 87
5.7 Summary . . . 89

6 Conclusion 91
6.1 Summary . . . 91
6.2 Contributions . . . 92
6.3 Future works . . . 93

7 Author's Publication and Software Availability 94
7.1 Author's publication . . . 94
7.2 Software availability . . . 95

List of Figures

2.1 Architectural view of DISSECT-CF [57] 33

3.1 Architecture of proposed model 37
3.2 Flowchart of generating trace by Generic Trace Producer and other components 41
3.3 Coefficient of determination evaluation for generated execution time (left-figure) and memory utilisation (right-figure) 43
3.4 Dynamic memory allocation of our model 43
3.5 Architecture of generator approach 46
3.6 Flowchart for process of generating trace 48
3.7 Coefficient of determination evaluation for generated percentiles with help of GA, for execution time (left-figure) and memory utilisation (right-figure) 52
3.8 Number of orchestration triggers invoked in one day 54
3.9 Percentage of invocations for timer trigger per user, Left-figure: first minute of the day, Right-figure: Last minute of the day 54
3.10 Performance metrics extracted from simulation, left-figures were generated by simFaaS, right-figures were generated by our model 60

4.1 Architecture of our serverless environment 64
4.2 A demonstration scenario executed on our extended simulation environment 68

5.1 The average number of serverless functions executed simultaneously in Azure Functions provider 78
5.2 Representing multiple events in Table 5.1 occur at a specific time . . 79
5.3 Diagram of Timed class and Parallel class 80

5.4 Boxplot diagrams for JobDispatchingDemo class and our classes . . . 84
5.5 The execution time in seconds of different workload sizes simulated by parallel and sequential versions . . . 86
5.6 Percentage of average life-time, running-time and idle-time of instance for different workloads . . . 89

List of Tables

2.1 Comparison of virtual machines, containers and serverless computing features . . . 24
2.2 Review articles for cloud simulators . . . 27
2.3 Extensions of various aspects that based on cloud simulators . . . 29
2.4 Comparison of the examined serverless simulators with servereless features . . . 31

3.1 Top 10 users by number of tasks submitted to the real provider . . . 53
3.2 Scaling workloads with real users' behaviour . . . 53
3.3 Simulating twenty thousand functions with different formats . . . 56
3.4 Open-source simulators used recently by researchers in their evaluation process . . . 57
3.5 Comparing simulating functions using DISSECT-CF and GridSim in terms of simulated timespan . . . 58

4.1 Simulating 100k invocations using different memory sizes . . . 71
4.2 Simulating 100k invocations using the AWS provider with instances having 128 MB memory . . . 72
4.3 Using timer trigger for five groups of functions with different trigger intervals . . . 73
4.4 Average performance metrics of the instances were extracted while simulating different workload sizes . . . 73

5.1 Three events with different frequencies . . . 79
5.2 The execution time(s) of parallel version using four different sizes of list 85
5.3 The execution time(s) of parallel and sequential versions in four different degrees of parallelism . . . 86

5.4 Extracted performance metrics of different workloads using our serverless environment 88
5.5 The execution time in seconds of our serverless environment using sequential and parallel versions 89

List of abbreviation

BaaS Backend-as-a-Service 17
CSV Comma-Separated Values 55
DES Discrete Event Simulation 25
GPU Graphics Processing Unit 76
GWF Grid Workload Format 55
FaaS Functions-as-a-Service 4
HPC High Performance Computing 82
IaaS Infrastructure-as-a-Service 13
IoT Internet of Things 17
PaaS Platform-as-a-Service 16
PDES Parallel Discrete Event Simulation 29
PM Physical Machine 22
QoS Quality of Service 58
SaaS Software-as-a-Service 16
SWF Standard Workload Format 55
VM Virtual Machine 17

Chapter 1

Introduction

Serverless is a new computing paradigm adopted by several cloud providers. It provides a new style of delivering cloud services by letting a user to mainly concentrate on coding rather than managing backend-infrastructure and operations [70, 96]. In most cases, commercial providers are not always the most favourable choices for researchers to execute and evaluate their desired scenarios, due to having a costly and complex environment (leading to non-reproducible results) [53, 72].

One alternative solution is simulators, which were opted by the research communities to evaluate scenarios in reduced-cost and easy-to-setup-environments [35, 43]. Over the last years, numerous cloud simulators have been built to support the Infrastructure-as-a-Service (IaaS) model [10, 16, 35, 54, 106]. They offer a flexible environment to experiment on various algorithms and scenarios in the field of infrastructure management. To obtain respectable precision, simulators use real traces often collected and offered by commercial providers. These traces represent comprehensive information about executed tasks reflecting users' behaviour within providers.

Despite the widespread use of cloud simulators, they are still mainly focused on supporting more traditional IaaS scenarios, and this reduces their applicability in the serverless and FaaS domains. There are several features essential to support the serverless models that are missing from most IaaS frameworks. For example, the need of simulating and execute multiple events in parallel. Moreover, workload traces typically employed by IaaS simulators are not well adoptable to the new computing model. Thus, they do not represent the new kind of users' behaviour. In addition to that, IaaS simulators are not designed to take responsibility for managing the necessary infrastructure, complex provisioning, and configurations on behalf of a user, which is how he/she deals with serverless systems.

The need for an integrated serverless environment capable of mimicking the behaviour of real providers is essential towards evaluating applications and scenarios reliant on the concepts of this computing paradigm. To fulfil the researchers' purpose, the environment has to support the newly introduced features, computing style, and resource constraints that led to exist of this computing type.

As serverless technology still based on underlying infrastructure that is abstracted from a user, it is beneficial to extend existing IaaS simulators to support serverless functionalities and features. The IaaS simulator called DISSECT-CF [57] is selected to fulfill this purpose as it is extensible, allows sharing low-level computing, supports loading and managing several trace file formats, and its performance is significantly higher than most simulators in the field [71]. All these reasons are leading us to the main aim of the research.

1.1 Research goals

This research aims to develop a comprehensive serverless environment on DISSECT-CF simulator. This new environment is capable of simulating and evaluating serverless applications and scenarios. The research is divided into three goals.

1. The ability to generate realistic serverless workloads close to real users' behaviour. Also supporting scaling such workloads to fit any researchers' desired scenario.

2. The ability to mimic real serverless providers in terms of provisioning resources policy, internal mechanism, estimating costs and provided services.

3. The ability to perform parallel execution for revealing the internal behaviour of large-scale simulation session.

1.2 Dissertation guide

The list below shows the organisation of the chapters, which constitute this dissertation.

Chapter 2 gives an overview of serverless computing and the terminology behind the existence of this computing. Moreover, it analyses the most popular providers that support this computing paradigm. It then explains the role of traces and a simulation environment in the research area. After this, it presents the literature

review that conducted related to our study including limitations and common features that exist in current cloud and serverless simulators. Subsequently, it exposes the concluded remarks that urged us to perform this study. Finally, it provides an overview of DISSECT-CF and the reason behind selecting this robust simulator to be the foundation to achieve our goals.

Chapter 3 introduces our novel approach for generating realistic serverless traces. It commences by proposing the architecture of a model that consists of two layers built on the top of DISSECT-Cf. It then demonstrates the mechanism of producing serverless workloads. Besides, it exposes the enhancement that introduced to enable our approach to improve the quality of traces by using a genetic algorithm. Additionally, it allows scaling of workloads with attention to real users' behaviour. It also supports the reusability of generated traces to be used by simulators that model different computing approaches. Finally, it offers different ways of evaluating our generator approach.

Chapter 4 looks into the layers that constitute the architecture of our introduced serverless environment that supports several serverless providers. Followed by detailed steps to reveal the internal mechanism of this environment. Finally, it deals with the evaluation of the functionalities and services that are offered by this serverless environment.

Chapter 5 has the spotlight on the prominence of recurrent events in DISSECT-CF and demonstrates on proposed solution (parallel event system) to the core of DISSECT-CF. After this, it shows a performance comparison between the sequential and parallel versions. Finally, it deals with the investigation of the parallel version in our serverless environment with a scenario that require high performance.

Chapter 6 provides a summary of the dissertation and highlighted the outcomes. Additionally, it summarises the main contributions of the author towards the scientific community. Finally, it considers possible future work that could be done to further extend this project towards various applications and scenarios that reliant on the concepts of the serverless paradigm.

Chapter 7 provides the list of scientific publications achieved by the author to support the introduced contributions. Moreover, it gives a link to the source code of introduced serverless environment.

Chapter 2

Background and Literature Review

2.1 Introduction

The term cloud computing was first introduced in 2006 to offer elastic and reliable services to the maximum number of users [11]. Cloud computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the data centers that provide those services. The way and type of delivering these services led to constitute several models to describe products, most prominently, IaaS, Software-as-a-Service (SaaS), and Platform-as-a-Service (PaaS) [31].

Research in these fields cannot rely on commercial cloud providers such as AWS and Azure, as their inflexibility to setup experiments with the same behaviour and cost often limit the required levels of reproducibility and scalability. Thus, many IaaS simulators have been developed as alternative solutions to experiment scenarios in a cloud computing environment [16, 59, 72].

Recently, serverless computing was introduced as FaaS, to deliver cloud services in style that abstracts a user from the burden of managing infrastructure. Serverless computing enables software development agility towards supporting various real-world scenarios. The need to introduce serverless computing to the research community in a simulation environment is essential towards evaluating scenarios that reliant on the concepts of the serverless paradigm.

This chapter gives an overview of serverless computing terminology and its popular providers. Section 2.2 gives background information on serverless providers and highlights the role of the simulation environment and traces in the research community that led to exist various simulators in different aspects of computing. Additionally, it presents the IaaS simulators and their shared common features and

limitations. Section 2.3 shows conducted surveys related to the aim of our research study and the existed limitations in current serverless computing simulators. Finally, Section 2.4 summarise the key concepts behind this study.

2.2 Background

2.2.1 Serverless computing

Serverless computing is a step forward to provide a cloud environment. It's style differs from traditional cloud computing in terms of releasing users from the burden of managing underlying infrastructure and operations [70]. Serverless does not mean that there is no server in general, but the operations, scalability and backend-infrastructure that related to server is fully abstracted from users, and managed by cloud providers [48].

Serverless computing is defined as a complete computing model that incorporates two different services, namely, FaaS and Backend-as-a-Service (BaaS). FaaS is concept that focuses on event-driven execution of written functions to accomplish target actions. BaaS is cloud service that manages by provider to backend functions to accomplish a task. This includes any type of service in which server management, configuration, scaling, and billing are abstracted from the end user.

Serverless is gaining momentum over public clouds as a great solution for industry and development. By contrast, it is limited in the research community in terms of offering research insight into infrastructure and concepts behind [13].

Serverless computing has become dominant environment for applications whose nature of workloads depends on lightweight and stateless services [70] such as big data [20], Internet of Things (IoT) [14, 26], scientific computing [34], artificial machine learning model training [22], mathematical Computation [98], and workflow applications [52, 69].

2.2.1.1 Serverless computing terminology

Function trigger: Serverless providers introduce triggers to invoke functions when certain events happen [33]. This gives different methods for defining how a function will be run on the basis of associated trigger. Trigger types can vary to meet a user's scenario demand, such as timers with the predefined schedule or a blob that responds to new or updated blob status.

Instance lifetime: An Instance is a Virtual Machine (VM) which runs our tasks in the cloud. The lifetime of an instance starts by initial state that requires

allocating sufficient resources and loading configurations. Then, the instance will be able to handle incoming requests (running state). Whenever there are no more dispatched requests, the instance will be kept warm (idle state) for short-time for future reuse or terminated [67].

Cold-start and warm-start: Cold-start happens when a task is dispatched and there is no instance available to run this task. Thus, the platform needs to boot a new one from scratch and allocate appropriate resources that meet the task's demand. The time spent on initialising instance to be ready is called cold-start delay. In the opposite scenario, warm-start represents an available instance that is ready to accommodate the dispatched task [105].

Auto-scaling: One of the essential aspects of serverless (FaaS) platforms is auto-scaling instances based on incoming requests [111]. This ensures provisioning of resources rapidly in response to workload, including scaling to zero that allow instances to be run when there is demand. Essentially, the auto-scaling solution behind the provider will make task dispatch more or less likely to be in the cold start state. What makes serverless providers unique is their ability to launch instances quickly and manage them efficiently.

Utilization-based billing: A factor that attracts a user towards serverless platforms is paying only for consumed resources during the actual running time [112] of the function. This pattern is fairly inexpensive compared with renting and paying fixed-term resources in traditional IaaSs.

2.2.1.2 Serverless computing use cases

The advantages of serverless computing, such as scalability and minimized administrative overhead, make it a fitting choice for various use cases. Here are some common use cases.

Event-Driven Applications: one of the most popular use cases is the implementation of event-driven applications that leverage serverless computing architecture to respond to various events in real-time [33, 114]. As the serverless environment is built upon an event-driven design that inherently enables loose coupling, it runs applications in response to functions triggered by events, which facilitates asynchronous processing without requiring continuous server resources, independent scale, deployment flexibility, service abstraction, and isolation. The serverless computing model can respond to event changes or updates in the state, such as HTTP requests, file uploads, messages from a message queue, and database updates.

Multimedia Processing: the serverless architecture is commonly used in image and video processing tasks. By combining serverless functions such as compute and

flow, it is possible to build resilient and scalable serverless video processing systems. These systems not only deliver improved performance and efficiency but also come with reduced costs [29, 117]. Triggering serverless functions immediately upon file uploads to a storage service enables efficient processing for activities such as resizing images or video processing, generating thumbnails, encoding or transcending videos, data extraction, and transformation.

Chatbots: serverless computing is used to implement chatbots which respond to user queries. The chatbot's backend relies on modular blocks to engage and collaborate with cognitive services. Each interaction within the chatbot is treated as an independent serverless function that executes valuable tasks such as processing user input and fetching information [27, 115]. This makes serverless architecture well-suited for such workloads, enabling a chatbot employing numerous interconnected functions to effortlessly scale, managing thousands of simultaneous interactions without the need for a continuously running server [47, 63].

APIs and Backend Services: serverless computing is frequently employed to facilitate APIs that foster communication between front-end applications and back-end services [83, 110]. It manages API requests, executes essential computations, fetches data, and interacts with databases or additional services. Within a backend service, serverless computing can be applied to establish particular functions, rendering them well-suited for managing specific tasks within the backend infrastructure.

IoT and Microservices Applications: serverless computing is well-suited for managing sporadic and unpredictable workloads linked to IoT devices and for deploying microservices [24, 65]. Functions within this model can efficiently process and analyze data produced by devices, responding to real-time events triggered by IoT devices. In the context of microservices, each function encapsulates distinct business logic or acts as a self-contained service. This methodology streamlines independent development, deployment, and scaling of diverse components within a broader application.

2.2.1.3 Serverless computing providers

Many serverless providers deliver computing services to users without involving the burden of user level management operations of the infrastructure. In this section, we list the most popular providers and the key differences between their policies.

AWS lambda [13] was introduced in 2014 by Amazon. It came with a price model that mainly relies on allocated memory size and number of invocations [112]. AWS Lambda offers elastic options for allocating the memory of function instance starting from 128 MB to 3008 MB with an attached CPU whose performance in-

creases proportionally to the allocated memory. It also allows providing, by default a thousand, concurrent function instances to serve deployed functions [65]. However, the number of instances could vary depending on the region where the function is deployed [68]. AWS Lambda introduced resource limits to ensure dealing with intensive workload efficiently. These include a timeout of function execution set to 15 minutes [112] and the maximum time of a function instance to be idle before it terminates is 5 minutes [79].

Microsoft azure functions was officially released in 2016 to deliver serverless services in the same manner as AWS Lambda [70]. But the price model depends on the average consumed memory of serverless functions instead of the memory allocated. Azure functions introduced three hosting plans to run serverless applications, namely, consumption, premium and dedicated plans [112]. Each has different resource configurations and costs. It also uses the function app concept to accommodate one or more individual functions sharing the same runtime configurations. Each app can scale up to 200 instances with a maximum memory of 1.5 GB [68]. Thereby Azure app service limits the runtime of function to 10 minutes, and function instance will be terminated after staying 12 minutes in an idle state [79].

Google cloud functions was released in 2017 as serverless platform [70]. The price model of Google Cloud Function relies on provisioned memory and CPU. This platform offers users a for memory options like 128 MB, 256 MB, 512 MB, 1024 MB and 2048 MB. Google Cloud Functions does not highlight the number of instances it support a single function to be scaled to, but it allows up to 1000 functions to be executed concurrently per session [68]. This platforms determined 9 minutes for the function execution timeout [112] and 15 minutes idle time for function instance to be terminated.

IBM cloud functions is built on the top of open-source project "OpenWhisk" developed by IBM. It was released in 2016. The price model of IBM offers memory sizes, namely, between 128 MB to 2045 MB [68] and it charges for the GB-seconds used [62]. This platform automatically scales up to 1000 concurrent function instances to serve intensive workload. The maximum function timeout is 10 minutes in IBM Cloud Functions [112] and it allows 10 minutes for function instance to be idle [79].

In summary, although the aforementioned providers have common features and offer similar services, each one has different policy which relies on several factors such as cost model, auto-scaling mechanism, predefined configurations and constraints on resources.

2.2.1.4 Open-source serverless computing frameworks

Open-source serverless frameworks offer more flexibility to build and deploy serverless applications compared with using proprietary serverless computing providers. They are not introduces new cost models or policies but address vendor lock-in with FaaS in commercial providers [78]. This enables developers to execute FaaS on multiple providers or local machines. In this section, we list some popular open-source frameworks.

OpenWhisk [1] framework allows developers to build and deploy serverless applications and functions in a cloud-native, event-driven, and scalable manner. It enables written functions to be invoked via events, which are referred to as "actions". OpenWhisk came with distinct features that make it a compelling choice for researchers and developers, such as supporting various programming languages, providing automatic scaling of resources to handle differing workloads, and its architecture is designed around event-driven.

OpenFaaS [6] framework designed to streamline the deployment and administration of functions or applications within a serverless architecture, harnessing the capabilities of containers. It offers flexibility, extensibility, and language-agnostic support in a containerised environment that utilizes containers as the fundamental deployment unit for functions, making it compatible with various container orchestration platforms.

Kubeless [5] is a portable framework that allows developers to deploy and manage serverless functions on Kubernetes clusters. This framework extends the serverless computing model to Kubernetes to simplify the deployment, scaling, and management of serverless applications within a Kubernetes environment. In Kubeless, serverless functions can be triggered by diverse events, such as HTTP requests, and are compatible with multiple programming languages.

Fission [3] is framework seamlessly integrated with Kubernetes, enabling the execution of serverless functions within the Kubernetes ecosystem. Fission simplifies the operational intricacies associated with managing serverless applications, delivering a serverless experience atop Kubernetes infrastructure. This integration amalgamates the advantages of serverless computing with the robustness and adaptability inherent in Kubernetes. It supports integration with external services and APIs, facilitating the creation of serverless applications that effortlessly interact with databases, messaging systems, and various third-party resources."

IronFunctions [4] framework meticulously crafted to streamline the development and execution of event-driven, on-demand applications. It is particularly suitable to build and run serverless functions using containerisation technology, where each function runs inside its own lightweight container and provides consistency.

2.2.1.5 Serverless and other computing paradigms

In the cloud computing paradigm, the processing of data happens within centralised data centres that are often located at a significant distance from the end-users, whereas edge computing and fog computing are both characterised as technological paradigms that relocate computing processes to be in close proximity to the locations where data is generated and collected [55].

Traditional serverless computing was originally designed for cloud environments to automatically manage the underlying infrastructure and operations while functions (FaaSs or events) are executed in datacenters using VMs or containers [12]. However, Serverless is also establishing its presence in fog and edge computing by resolving FaaSs request on the nearest point of devices, the edges of the network, or nodes, enabling low-latency processing and efficient utilisation of fog or edge computing infrastructures [26]. That is, serverless transcends cloud boundaries to evaluate its advantages in fog and edge computing to serve their associated applications, such as IoT applications, which are based on events triggered through sensing/actuating in the same way functions are triggered in serverless [24].

Although our serverless environment is generic, our dissertation focuses on cloud computing as evaluation scenarios will be performed on centralized datacenters. However, it can also support typical scenarios on fog and edge computing that we intend to present in the future by deploying and simulating FaaSs on multiple nodes and edge devices.

2.2.1.6 Virtualization, containerization and serverless computing

Virtualisation is a technology that turns a single physical hardware's resources such as a server or Physical Machine (PM) into multiple virtual environments (VMs) to enable the running of several applications on a single server [100]. It uses a hypervisor to decouple physical hardware from virtual environments and allows each VM to have its own operating system and applications [75, 95]. Virtualization has several advantages, such as reduced operating cost, simplified development, efficient utilisation of resources, and multi-tenancy.

Containerization is a technology that virtualizes the entire operating system, applications, and their dependencies in containers managed by the underlying operating system kernel [41]. Containers are lightweight, scale quickly, and run faster than VMs because they consume much fewer resources and share the same operating system kernel [95, 100]. Containers do not require a hypervisor or guest operating system and enable the running of multiple applications independently and securely on a single operating system.

Serverless computing does not offer a completely new technology because containers and VMs are still used in the serverless underlying infrastructure [13, 41]. Instead, it provides an architecture style that achieves better efficiency for written functions by creating short-lived and lightweight containers or VMs, as well as caching and reusing them by the provider for multiple function calls within a particular duration. The efficiency behind serverless computing comes from mastering the entire underlying infrastructure, provisioning resources wisely, and special configurations of resources and cashing that reduce additional overhead [9].

Table 2.1 summarises features of traditional VM processing, containers and serverless computing.

2.2.2 Traces

Tracing is an effective way of recording detailed information about application activities in a real system such as a cloud services provider [85]. These applications are composed of tasks could work together to fulfil the desired purpose. During executing these tasks within a provider, it captures information related to these tasks, such as consumed resources and a user who executes them. This information is extracted as a text file called a trace file. Thus, a trace file represents the execution history of tasks by users on a specific infrastructure. Each trace record fully represents the behaviour of a task within a provider in terms of the amount of consumed resources (i.e., memory), the total time spent in execution, the user who invoked the task, and so on. The type of trace is distinctly defined by two main factors. First, the type of application that determines the characteristics of executed tasks within a provider. Second, the type of provider in which trace is collected [30].

Each provider has its own restrictions on executing tasks, which eventually affect the characteristics of the tasks in the trace. Collecting trace reveals the total behaviour of applications and users' activities within a provider that responds to them according to its policy. This helps researchers interested in the evaluation of real systems and to know their behaviour.

There are several useful traces collected from large-scale systems around the world, such as grid workload traces [45], parallel workload traces [7, 36], and google workload traces [86].

2.2.3 Simulation environments

Simulation is a technique used to turn a real environment into a computer environment where we are able to conduct experiments in similar behaviour as if we would

Table 2.1: Comparison of virtual machines, containers and serverless computing features

Feature	Virtual machines	Containers	Serverless
Isolation	Strong isolation because each VM has a separate operating system	Light isolation as an operating system is shared among containers	High isolation at the function level only
Resource Overhead	Higher, because of running a complete operating system for each VM	Lower, because of sharing the host operating system kernel	Fairly high overhead as resources are allocated and deallocated dynamically
Deployment and Scaling	Management, provisioning, and scaling can be performed manually or automatically with the help of cloud services	Horizontal scaling, deployment, and management are simplified compared with VMs	Automatic scaling with easy deployment as the management of the infrastructure is abstracted away
Portability	Portable because it is executable on every virtual machine monitor that supports the same hardware architecture	Less portable because it is only usable on exactly the same host operating system	Complete lockdown because the serverless platform has vendor lock-in
Cost model	Continuous cost, as VMs remain active continuously regardless of usage	Resource-effective, cost scales with the number of containers	Cost-efficient due to pay for actual usage
Resource Allocation	Resources are allocated manually or automatically for each VM	Resources are allocated manually or automatically for each container	Automatically scales resources based on demand
Use Cases	Suitable for applications with long-running processes and consistent workloads	Well-suited for microservices and distributed systems applications	Ideal for event-driven and short-lived tasks such as IoT applications.
Examples	VMware and VirtualBox	Docker and Kubernetes	AWS Lambda, Azure Functions, Google Cloud Functions

on a real system [80]. There are three major types of simulations that determine the behaviour of experiments within the simulation environment, namely: discrete which the state variables change only at a countable number of points in time, continuous which the state variables change in a continuous way, and not abruptly from one state to another, and combined which involves both [87]. Thus, the simulation state will be changed frequently in time fashion according to changing of involved variables. The Discrete Event Simulation (DES) type is massively used in cloud field [99], as it requires capturing only the state changes over time, and no need to record the complete state. In the cloud research community, simulations have gained popularity [80], due to the following advantages:

1. Cheap environment compared to conducting experiments using the real system.

2. High-level of elasticity to configure scenarios and experiments in a reproducible environment.

3. Easy-setup and control even regarding the underlining layers that can be hardly to be performed in a real system.

4. Producing accurate result undistracted by other external variables such as different number of users present at any particular time in real system.

2.2.4 Evaluation methodologies

In our dissertation, several evaluation methods were used to validate the final results. In chapter 3, the coefficient of determination (R^2) was used to validate most results, as it measures the goodness of fit of extracted percentiles and averages as predicted against the original values in the dataset. This ensures that a single measurement considers all the extracted values to find the correlation between predicted percentiles and average in the dataset for each function and the actual percentiles and average in the dataset. Percent difference was also used to determine the range of change for the output of simulators when they have the same input. In chapter 4, the obtained results, such as estimating cost, have been validated against the AWS provider to show the accuracy of the results. Finally, in chapter 5, the results of the simulation for sequential and parallel versions were compared to demonstrate the ratio of improvement by the parallel version.

2.3 Related works

Our methodology to find relevant work was divided into two parts; we, first, investigated recent published surveys of cloud computing in a simulation environment. Second, we looked for current serverless simulators by delving into the drawbacks of the existing solutions and revealing neglected features. During the literature research, we focused on the relevant aspects and features to our research aims by highlighting common features, limitations, and research trends.

2.3.1 Cloud simulators

Over the last two decades, several IaaS simulation frameworks have been built to offer high levels of freedom to experiment. These frameworks introduced many features, functionalities, and concepts to handle the challenges that accompanying the cloud computing field. We conducted literature research on survey papers to reveal different aspects of cloud computing in a simulation environment, as the results show in Table 2.2. Each one provides a critical and comparative analysis of several simulators, and sheds the light on diverse evaluation criteria.

Based on the literature, we have concluded with 51 unique cloud simulators. Each simulator was built for a specific purpose. On one hand, some IaaS simulators have been built to focus on specific aspects such as energy-aware provisioning, and middleware supervision as the best solution in this sub-field. For example, GreenCloud [60] is a simulator specifically built for estimating the energy consumption of cloud data centres. Or, SPECI [104] is a simple simulator that was built to investigate middleware supervision protocols of data centres. Finally, GroudSim [82] is a platform mainly focused on scientific application modelling (e.g., workflows) in cloud and grid computing.

On the other hand, some IaaS simulators have been built to suit wider cloud modelling scenarios with extensibility in mind to support comprehensive features. Thus, they provide essential architecture and significant concepts as foundation to others. For instance, CloudSim [19] is one of the well-known frameworks designed to mimic general cloud behaviour. CloudSim built with the ability to introduce new features that support modelling and simulation of cloud computing environments and its extensibility has been demonstrated with numerous cases over the years.

DISSECT-CF [57] is a simulation framework that improved the modelling of resources, network utilisation, power consumption, and data centre configurations, by providing the capability of simulating IaaS internal behaviour. DISSECT-CF demonstrated its capability towards supporting DES environment in various aspects

Table 2.2: Review articles for cloud simulators

Authors	**Year**	**Main goal**
Ahmed et al. [10]	2014	Provide a comprehensive overview of 11 simulators and highlighting their important features, and analyse their pros and cons.
Ettikyala et al. [35]	2015	Present a comparison of 15 simulators based on the type of simulators, developed programming languages and other characteristics.
Kaleem et al. [54]	2015	Study 12 simulators and provide a better understanding of attributes for selecting appropriate simulators for new users.
Suryateja. [106]	2016	Give an overview of 17 simulators based on diverse criteria such as programming language and base platform.
Byrne et al. [16]	2017	Provide a review of 33 cloud tools focused on plugins and extensions proposed by researchers to support different aspects of cloud, edge, and fog computing. Additionally, it reveals the lack of distributed execution and parallel execution.
Khalil et al. [59]	2017	Investigating the common architecture of 33 simulators and provides evaluation cloud services and modelling equipment. Moreover, it shows the capabilities, challenges, and extendibility of cloud simulators.
Mansouri et al. [72]	2020	Providing a multi-level feature analysis of 33 frameworks with a practical way for evaluating their performance based on various scenarios.
Ismail. [46]	2020	Study 11 simulators based on developed types such as general purpose or specific simulators with energy concern. It also shows the challenges that e.g., could face these simulators.

of computing such as fog [73].

While investigating, we observed several properties that reveal trends, common-aspects, and current challenges. These properties in most cases required going in the deep of concerned simulators to obtain more details. We have categorised them as the following.

2.3.1.1 Implementation language

Cloud simulators have been developed by researchers to be used as reliable tools to conduct experiments and modelling various types of applications. Designing evaluation scenarios require sufficient knowledge in the underlying programming language of simulators. In our study, it was observed that Java dominated the programming language category for 80% of investigated cloud simulators.

2.3.1.2 Simulation type

Cloud simulators have been built to imitate the real providers by simulating events in the same behaviour that occurred in the original providers. Event-driven simulators are based on the sequences of events that lead to changes in the state of the system. We have seen in this study that 76% of the investigated cloud simulators fall into the discrete event-based category.

2.3.1.3 Simulation input

To imitate the behaviour of computing providers with respectable precision, simulators have to use realistic traces that were collected from real providers. These traces represent users' behaviour by showing the type of executed tasks and consumed resources. During this study, we found that 58% of cloud simulators supported realistic traces for simulating scenarios, while the others used synthetic workloads.

2.3.1.4 Support extension

While designing cloud simulators, it is beneficial to consider the extensibility to include additional features and functionalities for supporting future not yet foreseen technologies. We have observed that cloud simulators designed for general purpose such as CloudSim and DISSECT-CF have been used as foundations to support other aspects of computing such as fogs. The Table 2.3 shows the extensions that were built based on the core of these simulators.

Table 2.3: Extensions of various aspects that based on cloud simulators

Simulator	Serverless	Fog	Edge/IoT
CloudSim [19]	DFaaSCloud [50]	iFogSim [42], MyiFogSim [66]	EdgeCloudSim [103], PureEdgeSim [76], IOTSim [116], IoTSim-Edge [51]
DISSECT-CF	Our model [92, 94]	DISSECT-CF-Fog [73]	DISSECT-CF-IoT [74]

2.3.1.5 Lack of parallelsation

Although IaaS simulators offer many features that fulfil researchers' intent, we have observed that the majority were built in a sequential fashion. Parallel Discrete Event Simulation (PDES) approach has been applied in various fields with the primary goal of performance. For example ROSS [21] and GWT [28] are parallel discrete event simulators that execute on shared-memory multiprocessor systems. They mostly used in large-scale networking simulation models and telecommunication networks. DaSSF [64] is also parallel simulator targeting network simulation and it achieves high performance through parallel processing.

Unfortunately, cloud (IaaS) simulation frameworks have limited parallelisation. Parallelising existing systems remains a challenge. However, one of the frameworks called Cloud2Sim [56] supports concurrent and distributed simulations of clouds, based on the following libraries: Hazelcast, Infinispan and Hibernate.

In [39] the author raises many challenges that researchers could face in a PDES. One of these challenges is the complexity of using a parallel implementation correctly and simplifying code to understand it easily. Agreeing with this, our approach aims to keep the original sequential APIs while making a parallel solution in the background. In [40] the authors suggested the initial steps towards cloud supporting PDESs, unfortunately these steps were not yet adopted by current simulators. Introducing parallel execution to simulators needs easy simulation control as well as repeatable tests. In [32] the authors explained that sequential execution can be insufficient for modelling real complex systems, and parallel execution could manage resources efficiently. Sequential approaches are unable to fulfil many requirements, and lead to trade-off between the cost and performance.

2.3.2 Serverless simulators

Recently, few serverless simulators have been developed either by extending the IaaS simulators' functionalities or from scratch. To fulfil the researchers' purpose towards simulating FaaSs, serverless simulators have to mimic a real provider by offering the foundations and features of this new technology. Unfortunately, serverless simulation is in its infancy, frameworks only partially support the features (e.g., auto-scaling) of the new model. Thus, using them for evaluating new approaches to manage FaaS systems could still lead to potentially misleading research results.

DFaaSCloud [50] simulator extends CloudSim [19] to support some serverless features. It allows defining FaaS functions with various profiles and characteristics that determine their behaviour during simulation. However, it has the following major limitations for its support of serverless environments: (*i*) establishing and managing the virtual infrastructure backing functions is not fully supported because it does not consider the providers' policies, (*ii*) unable to utilise large scale generated realistic serverless traces due to use synthetic workload, and (*iii*) extracting serverless performance metrics (e.g., probability of cold-start and average utilization) from the simulation are missing.

OpenDC serverless [53] is a framework that was introduced based on the OpenDC simulator. It allows to model and test custom FaaS patterns. It introduced the essential architectural component, instance routing policy such as that imitates the basic serverless platform. However, it lacks an important concept of serverless, namely auto-scaling resources that responds to workload. It also doesn't provide information about the internal behaviour of infrastructure and function status.

SimFaaS [67] is a simulation platform that introduces a serverless environment to enable researchers to develop and optimise FaaS applications. It is designed to extract performance metrics from simulation. However, one of the missing features that make simFaaS, not-a-comprehensive-simulator, is the trigger, which defines how FaaS functions will invoke during simulation. SimFaaS also doesn't provide a foundation for a cost model of real providers. Moreover, the provider's resource constraints cannot be applied to simFaaS. Extracting performance metrics in simFaaS mainly relies on the already existing attributes of functions in the author's proprietary trace format, such as cold-start probability. Additionally, it calculates some metrics based on the traces, not actually evaluating the results from the simulation session.

Table 2.4 summarises research gaps and missing features of existing serverless simulators that need to be addressed to fully support researchers, developers, and end users. Providing such features enables researchers to experiment and evaluate various serverless scenarios in the same manner as commercial providers, as well as having full control over the simulation environment.

Table 2.4: Comparison of the examined serverless simulators with servereless features

Feature	**DFaaSCloud**	**OpenDC**	**SimFaaS**
Auto-scaling	N/A	×	✓
Cost model	×	×	Partial
Function triggers	×	×	×
Extracting performance metrics	×	×	Partial
Applying the policy of commercial providers	×	×	×
Revealing internal behaviour of infrastructure and provisioning resources	×	×	×
Concurrent provisioning resources	×	×	✓
Supporting trace file	N/A	✓	✓
Establishing and managing virtual infrastructure	N/A	×	×
Parallel simulation and execution	×	×	×

2.3.3 Discussion and concluding remarks

In the past decade, we have seen how simulators play an important role in computing by enabling the evaluation of desired scenarios and algorithms in an easy-setup and reproducible environment. Thus, the research community has been using them in their evaluations as an alternative solution for services provided by real cloud providers.

We have concluded the following the key findings: first, the most favourable environment in cloud simulation is a discrete event simulator that has written using the Java programming language. Second, the general-purpose cloud simulators such as CloudSim and DISSECT-CF, demonstrated their ability to support other technologies and are potentially suitable to provide solid-foundation to support serverless model. However, as demonstrated by [71] CloudSim's performance is not sufficient, to provide a robust-extension able to analyse the internal behaviour of function instances in serverless computing. Third, the majority of cloud simulators use traces to mimic the behaviour of real providers. However, workload traces collected from past providers are not well adoptable for modelling infrastructure workloads of serverless frameworks, due to their resources not streamed from serverless platforms and their attributes not fit the serverless policy. Fourth, cloud simulators lack of parallel execution, which could struggle to address the scale increases observed in serverless. Finally, there is a lack of features in existing serverless simulators that are not covering this new computing paradigm's needs.

We can conclude that the need to enrich the research community with a comprehensive serverless environment is crucial to fulfil researchers' expectations towards this technology, by offering the services behind serverless computing. Therefore, in this dissertation, we addressed the aforementioned limitations by introducing an integrated serverless environment (dubbed as DISSECT-CF-FaaS) able to generate and simulate large-scale serverless workloads. This environment is capable of supporting several real providers and mimicking their distinctive policies. It also provides parallel execution to foster analyzing the internal behaviour of our environment. The introduced serverless environment was built based on DISSECT-CF simulator. The rest of this dissertation is going to focus on these extensions.

2.3.4 Overview of DISSECT-CF simulator

DISSECT-CF [57] is a discrete-event simulation framework that offers insight into advanced cloud concepts. DISSECT-CF is written in Java and provides an amalgamation of several features that hardly exist in any previous simulators such as capturing low-level resource sharing behaviour and introducing an extensible energy

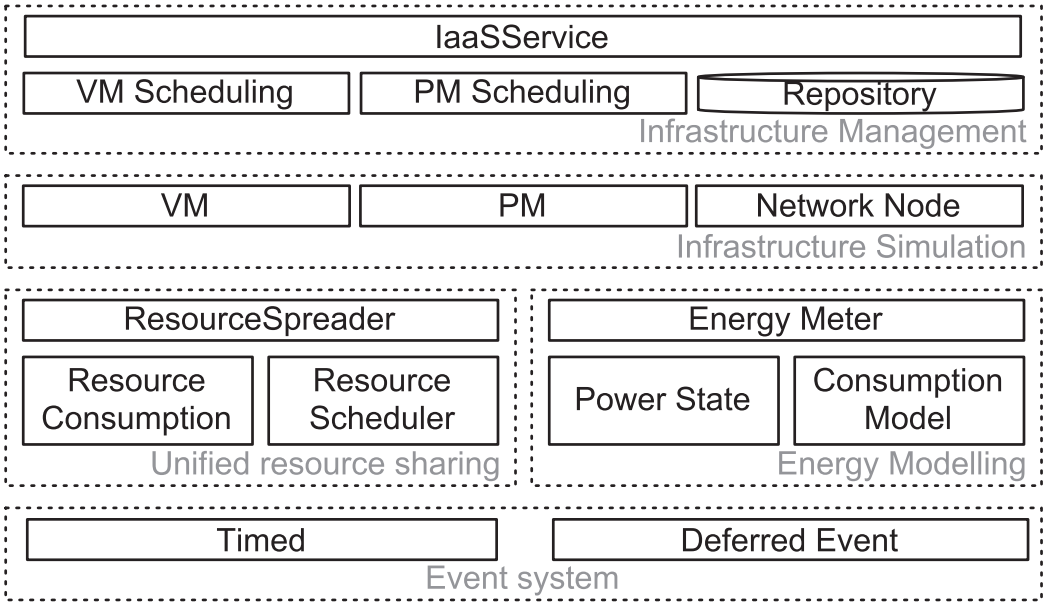


Figure 2.1: Architectural view of DISSECT-CF [57]

consumption model. This aims to support previously problematic IaaS simulation scenarios. In DISSECT-CF, time is measured in ticks and users of the simulator are free to interpret ticks the way they want.

The extensible core of the DISSECT-CF simulator consists of five major subsystems in a layered fashion as shown in Figure 2.1. Generally, each layer attempts to provide a comprehensive implementation for a particular concept without being dependent on the rest of the framework. The lowest subsystem the **event system** provides a mechanism to manage the behaviour of regularly and irregularly occurring events as well as controlling the basic state of the simulation in a given time instance. This subsystem is the foundation of all layers and introducing new features here has the highest potential impact on higher level subsystems.

Next, the **unified resource sharing** subsystem introduces a holistic approach to establish a central resource provider able to share behaviour among low-level computing concepts. Then, the **energy modelling** subsystem provides a unique approach that allows monitoring and analysing energy usage of all simulation resources by decoupling energy modelling from resource simulation (this allows performance gains by only offering selective energy monitoring). On a layer above, the **Infrastructure simulation** subsystem deals with modelling the behaviour of typical distributed sys-

tem components like VMs, PMs, storage and networking. Finally, the highest layer of abstraction is provided in the **Infrastructure management** subsystem which contains major IaaS components such as VM and PM scheduler that simulate the management of users requests and fosters the creation of custom internal IaaS behaviours. It also provides components such as Repository and the IaaS service to interact with users of the simulator.

Along the core DISSECT-CF, there are several relevant extensions and projects that enrich the simulator's feature set towards serverless simulation. The most relevant to this research is the auto scaling framework presented in the dissect-cf-examples project[1]. This framework enables the modelling of virtual infrastructure management tools and job schedulers on these virtual infrastructures. This allows simulations where the virtual infrastructure built up on top of DISSECT-CF core components are following the workload patterns pushed to the job scheduler.

The other relevant extension to the simulator is its workload representation focused DistSysJavaHelpers project[2]. This project provides abstractions to represent arbitrary workloads and offers ways to produce even generated workloads. Finally, it also enables the loading of several well known workload trace formats to foster realistic simulations.

2.4 Summary

In this chapter, we provided a background about serverless technology. We then analysed the common features and limitations of the most popular cloud simulators. Additionally. we revealed the researchers' trend towards the simulation environment and their favourable aspects. Finally, we showed the limitations of current serverless simulators and proposed this study to handle these limitations.

[1] https://github.com/kecskemeti/dissect-cf-examples
[2] https://github.com/kecskemeti/DistSysJavaHelpers

Chapter 3

Generating Realistic Serverless Traces

3.1 Introduction

Trace-based simulators are favourable in research [35], due to their elasticity to performing evaluation scenarios that could hardly be accomplished in real cloud providers. Over the past few years, numerous cloud simulators have been built to support the IaaS model [16, 72, 106]. These simulators use traces as a vital input to experiment the desired scenarios with respectable precision. These real traces are collected and offered by commercial or academic providers and represent comprehensive information of executed tasks or events that reflect their users' behaviour. Trace types are uniquely defined by first, their real provider where they were collected from. Second, the activity of users within that provider. Moreover, the provider's policy in terms of resource constraints and the nature of task leave a fingerprint in the behaviour of collected traces.

Due to serverless computing's recency, typical workload traces employed by IaaS simulators are not well adoptable to the new computing model due to trace's attributes not fit the serverless policy, such as the execution time of the task could exceed the timeout threshold of provider, and the trace does not contain information about how the task has been invoked (e.g., triggers). Moreover, the trace do not represent new kind of users. Thus, to simulate and predict the behaviour of functions in a serverless simulation environment, realistic workload needs to be used during the research scenario evaluation process.

In this chapter, we propose a novel approach for generating realistic serverless traces as a foundation for serverless and other simulators. We introduce two ways to implement the proposed approach to enrich current scenarios that desired by the

research community. The first manner is integrated with DISSECT-CF simulator to directly and internally generate realistic traces within the simulator's environment. The second manner involves an independent architecture that does not link to any simulators. It can be used to enrich various types of simulators with new traces types. The traces produced by our approach are based on the Azure Functions dataset (as it contains large number of functions), but its underlying concepts and mechanism are likely to be applicable to the other datasets as well.

The remainder of this chapter is as follows: Section 3.2 introduces the architecture of the proposed approach that is built on the top of DISSECT-CF simulator. Section 3.3 shows the mechanism of generating traces. Alongside this trace generation approach, it presents a simple evaluation framework to simulate generated traces. Section 3.4 demonstrates the improvement of the generator approach in Section 3.2 by introducing several concepts such as genetic algorithm to enhance outcomes, scaling workload, and supporting the reusability of generated traces. Moreover, it covers the evaluation of introduced concepts using various methods and simulators. Finally, Section 3.5 concludes the chapter and highlights the main accomplished results.

3.2 Architecture of the trace generator and serverless model

Supporting serverless computing model by IaaS simulators necessitate the introduction of additional features. One of the significant features is to enable the simulator to generate realistic-trace and predict the behaviour of such realistic serverless workloads. We introduced our serverless architecture based on DISSECT-CF that is aimed at offering automated management of FaaS workloads (by building on top of the pre-existing auto-scalers, see Subsection 2.3.4), as well as providing realistic models for such serverless models in order to enable building large scale simulations. The proposed architecture consists of three layers built on the top of DISSECT-CF, as shown in Figure 3.1.

The first step towards generating and executing serverless functions, is selecting dataset file and type of provider by a user of simulator, which will then be passed to the FaaS creation and configuration setup layers.

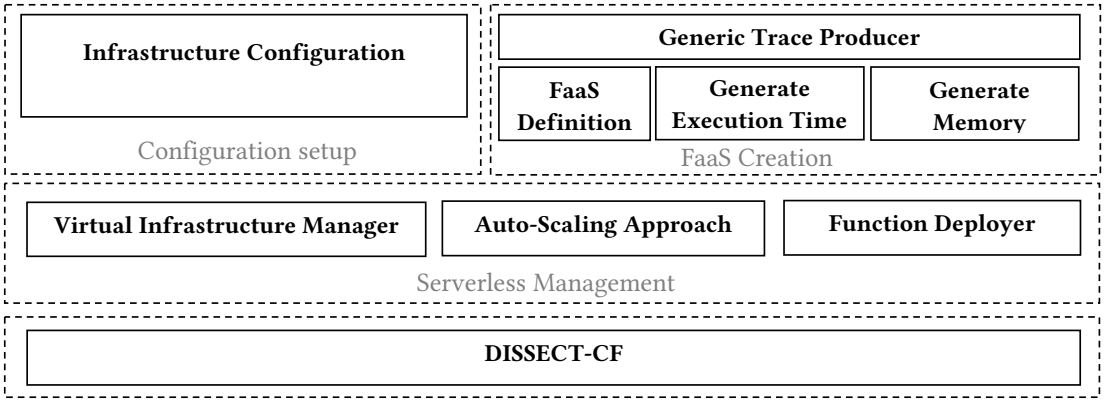


Figure 3.1: Architecture of proposed model

3.2.1 Configuration setup layer

It handles the user options by establishing proper infrastructure based on the selected provider's plan. It consists of `Infrastructure configuration` component that contains preset configurations that mimic a particular functions provider (in the current case, it was focused on Azure Functions [8]). The configurations offered focus on resource options (e.g., storage and memory) for functions as well as pricing and energy model details.

3.2.2 FaaS creation layer

It is responsible for generating serverless traces form a selected dataset file. It consists `Generate execution time` and `Generate memory` components that connect with the `Generic Trace Producer` component, which master the mechanism of generation. These components with help of `FaaS Definition` allow the generation of function invocations follow the desired amount and distribution specified by the simulator's user. The `FaaS Definition` component is responsible for populating the corresponding function definition in the simulation. The way function invocations are generated and governed by dataset file (in our case the Azure Functions dataset). The `FaaS Definition` component holds attributes that specifically define each function and its behaviour during simulation. These are either directly represented as individual invocation properties such as runtime, utilisation metrics; or they are represented through probability distribution functions of each metric for scalable reproduction of the original trace. The mechanism of the components of this layer are

further detailed in the Subsection 3.3.1.

3.2.3 Serverless management layer

It is responsible for managing virtual infrastructure (that backs the serverless computing platform), as well as providing just-enough resources for all the function invocations. This layer consists of following main components.

Virtual infrastructure manager is responsible for providing and managing the virtual infrastructure that backs the function invocations. It is used to offer a unified interface towards the auto-scalers, and abstract the cloud infrastructure so the VM requests or destruction requests are handled uniformly. This component is also responsible of collecting information on the provisioned resources during the simulation (e.g., the total amount of memory used at a particular moment). Finally, it enables calculating the number of provisioned instances and observing their status to determine the average of idle and utilisation of instances as well as the probability of cold-start.

Auto-scaling approach provides several, extensible mechanisms that observes the previously collected information, and makes sure the virtual infrastructure is increased or decreased in size according to the ongoing and future function invocations. When a decision is made to change the infrastructure, the `Auto-scaling approach` will notify the `Virtual infrastructure manager` component to request or destroy VMs.

Function deployer gets all generated functions and dispatches them to available VMs for execution. For example, when the number of functions needs to be dispatched simultaneously is increased, `Auto-scaling approach` component will observe the utilization of VMs and request more.

3.3 Generating realistic traces from Azure Functions dataset

The first official real-world FaaS workload dataset was publicly released on June 17th 2020 [97] by Microsoft Azure Functions on Github[1]. In this section, we demonstrate our approach of generating realistic trace from the Azure dataset. The Azure workload consists of 14 sets of three files representing 14 days of execution history. Each a day came with three files include real detailed information regarding a particular aspect of Azure Functions provider.

[1] https://github.com/Azure/AzurePublicDataset

The first file contains a history of invocations per function. This file contains 1440 columns (a full day) per function due to the invocations were binned at 1-minute intervals. Thus, it offers per minute details of function invocation counts. The second file includes the median, the minimum, maximum and average values for each function invocation kind's runtime and memory use.

The third and last file contains distributions of allocated memory per application, which is able to host the execution of individual functions. For this metric, each application's memory was sampled every 5 seconds, which they then averaged every minute. Alongside these, it also characterises the runtime and memory utilisation characteristics of a particular function invocation type. This characterisation is done in terms of disclosing the distribution of the utilisation via a few percentile values (i.e., they specified what was the runtime/memory value for 0^{th}, 1^{st}, first quartile, median, third quartile, 99^{th} and the 100^{th} percentile). The percentiles were provided on a daily basis showing the distribution execution time and memory utilisation for invocation count that were binned at 1-minute intervals.

To access holistic details for a one-day dataset simply, we developed code[2] that combined the three files into a single file, representing one day, by matching hash of the owner's ID, the function's ID and the application's ID.

3.3.1 Generating invocations, execution times and allocated memory

When a user selects the Azure dataset file, it will be processed by the `Generic Trace Producer` component. This loads the file and analyses its contents in terms of the number of lines existed in dataset file (each line represent a unique function and its detailed information regarding invocations over a 24 hour period, the distribution of execution time and memory utilisation and other details).

The `Generic Trace Producer` component reads each line from the dataset file and produces the attributes of existed function according to invocations (how many times current function is invoked in one day) as shown in Figure 3.2. This is done by extracting the percentiles of each function to be passed for generating execution time and memory utilisation values.

Generating the execution time and amount of memory for each function and its invocations, is mainly based on the values of the dataset file's percentile related columns. The minimum and maximum values determine the range for creating the invocations.

[2] https://github.com/dilshadsallo/DistSysJavaHelpers

The count value specifies the total number of invocations for single function in a day of the original dataset's recording. Our approach offers customisation options for the count value in order to allow the simulator's user to generate traces which are similar but have different number of invocations. The percentile values that existed under percentile ranks, which shows how execution time and memory are distributed over one day in the original dataset. The execution time came with 1, 25, 50, 75, 99 and 100 percentile ranks, whereas the percentile ranks of memory are 1, 5, 25, 50, 75, 95, 99 and 100.

When the functions are defined, as a first step we calculate how many invocations should fall in each percentile rank by using the following equation.

$$Invocations_{number} = \frac{count * percentileRank}{100} \tag{3.1}$$

Thus, the total number of invocations will be divided into six and eight values for execution time and memory, respectively, according to the percentile ranks. As the percentile is a value score below which a given percentage of scores in its frequency distribution falls, each calculated value will be subtracted from its previous one, except the first value that falls under percentile rank 1. Thus, the total number will be equal the count value (total number of invocations).

The second step of our approach is generating execution time and memory values from the percentile values according to the number of invocations for each percentile rank. Here, `Generate Execution Time` and `Generate Memory` componentes take percentile values provided by the original dataset file, and then generates their corresponding values within the range (considering minimum and maximum values for each function) of percentile values. When the count value is customized the characteristics of the invocations are maintained from the original dataset.

Let us take an arbitrary number of invocations such as 89434 (count) with percentile values 10, 230, 550, 650, 800, 950 for percentile ranks 1, 25, 50, 75, 99 and 100, respectively. As the first step, our approach calculates the invocations fall within each percentile rank using equation 3.1. As result of first step, we will have 894, 21464, 22359, 22358, 21464 and 895 for percentile ranks 1, 25, 50, 75, 99 and 100 respectively. After getting total invocations for each percentile rank, the second step is generating execution time values for invocations. Our approach will generate randomly (with a uniform PRNG) 894 execution time values within the range of min (that came with original dataset file) and 10 that located under percentile rank 1. Then, it generates 21464 execution time values within the range of 10 and 230 located under percentile ranks 1 and 25, respectively, and so on. This process will continue till generating execution values for all invocations of selected function, and the same approach will be used for memory values.

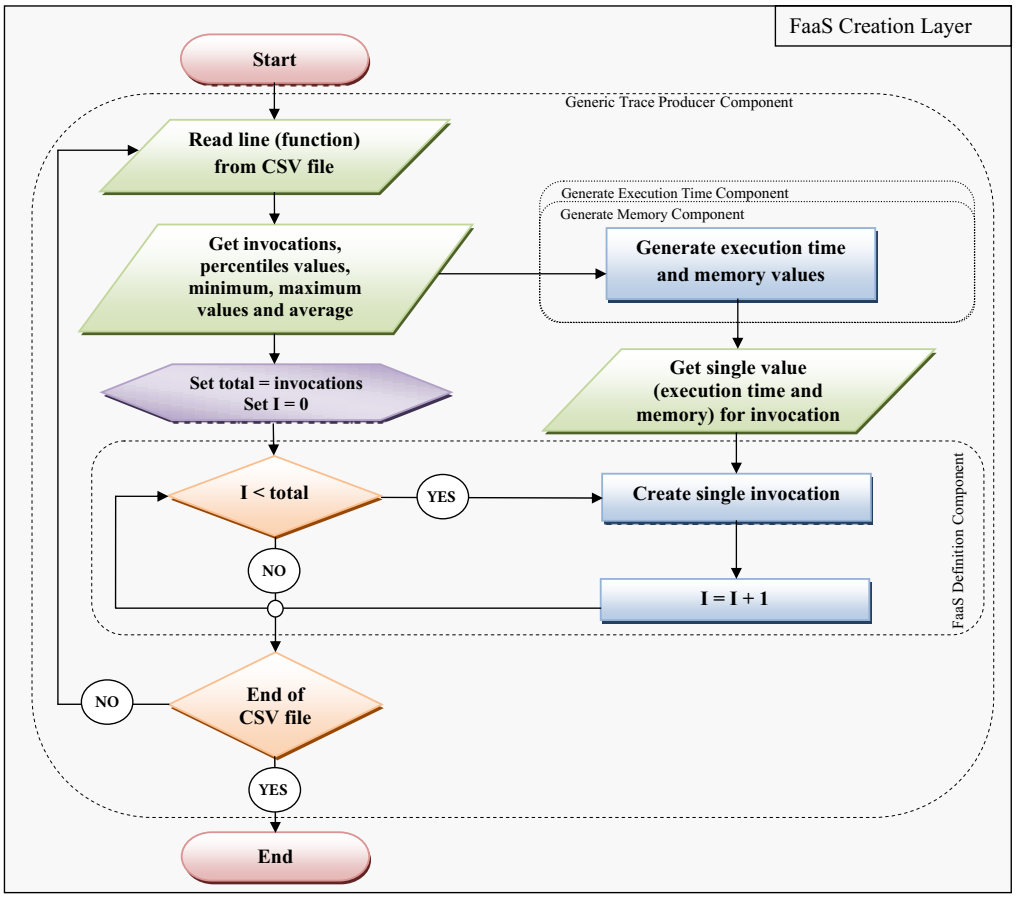


Figure 3.2: Flowchart of generating trace by Generic Trace Producer and other components

Finally, for each invocation, the `FaaS Definition` will be instantiated with the previously extracted values (e.g., amount of memory), unique ID and submitted time that determine the behaviour of function's invocation. `FaaS Definition` will continue to instantiate invocations according to the total number of function's invocations in the original dataset.

After generating all required function invocations, the `Generic Trace Producer` proceeds reading the next line from the dataset. This process continues till we finish generating all requested functions and their invocations. Once process is finished, the trace will be passed to serverless management layer to be used by the simulator's further components (this is achieved with the `Function Deployer`).

3.3.2 Evaluation of generator approach

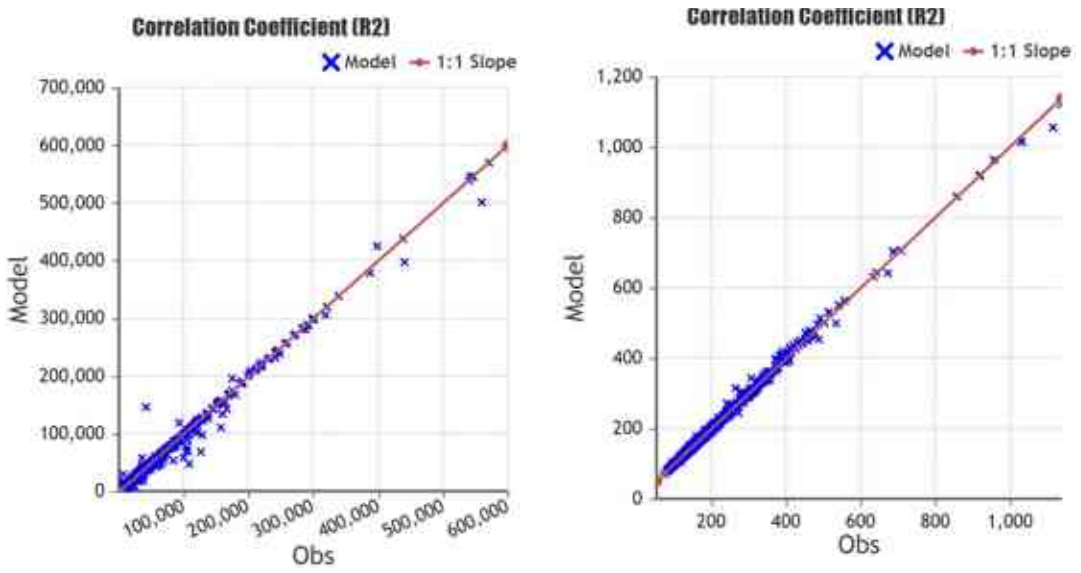
In all the forthcoming experiments in this chapter, a laptop (Intel (R) Core (TM) i7-4600U CPU @ 2.10GHz (4 CPUs), 2.7GHz, 8 GB) was used for the evaluation of our approach and model for generating and simulating FaaS.

To validate our approach that will act as a foundation for evaluation serverless model, we have chosen a one of dataset files that contains around 36000 unique functions. Then, we picked randomly 5000 functions and we have generated 5000 invocations for each function. In total, we have generated 25,000,000 invocations as medium-scale compared to total dataset's invocations. For each function that has generated 5000 invocations, we calculated the percentile values and average for both execution time and memory utilization. Then, we measured (R^2) between the generated and original values to show data accuracy.

For execution time, R^2 was 0.8438 for percentiles and 0.9956 for average, which indicates the accuracy (and realism) of our approach and how close the data points fall to the fitted regression line as shown in Figure 5.6a. For memory, R^2 were 0.8999 and 0.9977 for percentiles and average, respectively. The generated percentiles of memory is more adequate as shown in Figure 5.6b, compared to the execution time and the reason is the distribution of execution time values is significantly wider than the distribution of memory usage.

There are many application areas that are able to use our generated realistic traces, but we used Azure Functions provider as scenario to demonstrate the advantage of our introduced model. The configurations to simulate serverless functions are different in FaaS providers. Azure Functions provider offers several plans to be selected by a user for its workload. Our model is able to support consumption and premium plans, which have different resources such as the memory, price and storage. We have provided a simple measurement using the consumption plan, to simulate our generated serverless functions and predict the internal behaviour of Azure provider in our model.

We have conducted an experiment that directly generates (using FaaS creation layer) and simulates (using serverless management layer) around 2 million serverless invocations using our model. This scenario demonstrates how our model responds to rapid increase or decrease in demand of invocations during the simulation. The Figure 3.4 shows how our model imitates the Azure Functions provider by utilising dynamic allocation of memory based on the number of invocations at each time instance.



(a) Execution time

(b) Memory

Figure 3.3: Coefficient of determination evaluation for generated execution time (left-figure) and memory utilisation (right-figure)

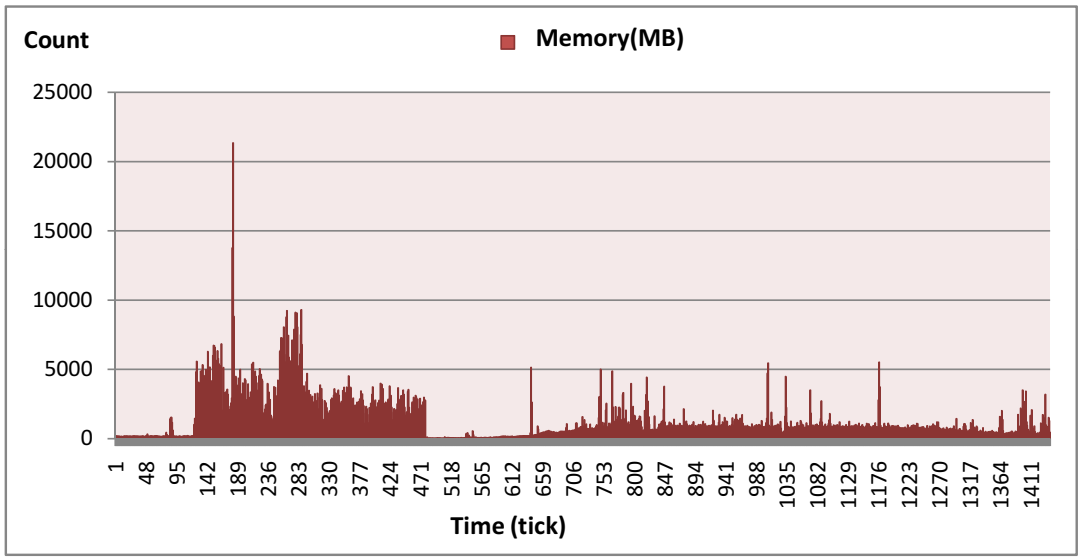


Figure 3.4: Dynamic memory allocation of our model

3.4 Improving our previous generator approach

It is often difficult to predict scenarios that are desired by the research community in the serverless field. Thus, it is always better to produce customised traces based on a particular time, homogeneous events, users' behaviour and invocation patterns. Moreover, giving an opportunity to exploit these scenarios by non-serverless simulators is essential to provide an alternative way to simulate serverless behaviour with simulators that do not natively support the serverless model.

In this section, we address the limitations of our previous approach by proposing a new architecture (see Subsection 3.4.2). It applies a genetic algorithm to obtain the best generated functions' attributes (the mechanism of the work is further detailed in Subsection 3.4.2.1). It also generates customised traces (see Subsection 3.4.2.2). Finally, it supported the reusability of the generated traces in other computing simulators by converting them to popular formats (see Subsection 3.4.2.3).

3.4.1 Limitations of previous approach

Our approach that introduced in Section 3.2, aimed at generating realistic traces based on Azure dataset. This approach allowed a user (who requires to produce trace) to select a particular function invocation kind and time period of a particular day to act as the model to generate traces. During the generation, the approach reused the submitter and function related information in the generated trace, while it provided new values for not-explicitly disclosed values like arrival time, runtime and memory utilisation. To generate these, we used the disclosed percentiles and minimum and maximum values. We validated this approach by generating several traces containing functions that were selected randomly from the complete dataset (i.e., we asked the generator to try to mimic the behaviour of a particular function on a particular day during a simulation). We then collected the average runtime and memory utilisation values, and shown that the coefficient of determination (R^2) was > 0.99. This suggests that we have produced an approach that is providing realistic average runtime and memory utilisation values. We also calculated the percentiles for the generated trace and the percentile values were having significantly weaker R^2 values due to the dataset's way of disclosing the percentiles (i.e., they did not disclose the actual percentiles but they first calculated average runtime and memory utilisation values over 30 second periods throughout the day, then this was used as the basis of the percentile calculation).

Dataset that is collected from the Azure Functions provider reflect users' behaviour (a user has unique HashOwner in dataset) by showing the number of invo-

cations (tasks), type of service and how frequently particular functions are invoked. Enclosing the real information is crucial for further studies such as predicting the consumption behaviour of users to stimulate resource usage awareness. The Azure dataset contains a huge number of functions and services, which could not meet several researchers' scenarios that require small-scale workload with real users' behaviour. Unfortunately, this previous approach did not consider this feature to support this scenario. It also neglects generating traces based on triggers that came with original dataset such as http, timer, event, queue, storage, orchestration and others.

One of the observed limitations of the previous approach is the lack of trace reusability. As it is integrated internally with a serverless model of DISSECT-CF simulator, every time there is a demand of serverless traces, we must go through the process of generating realistic traces completely. Moreover, other simulators have no opportunity to advantage from the generated realistic serverless traces as they are not adapted to this approach.

3.4.2 Architecture of enhanced approach

To remedy the aforementioned limitations, we introduced independent architecture that holds FaaS creation layer from the previous approach as well as adding three further components, namely `GA`, `User Behaviour` and `Standard Format` as shown in Figure 3.5. They are introduced to improve quality, enabling scaling workload and providing reusability of generated traces respectively. These components are compatible with the previous generator approach and distributed over the following two layers. **FaaS creation**, which is responsible for generating traces based on a selected dataset. **Processing layer**, which is responsible for detecting the users' behaviour and converting the generated traces to standard formats.

Our new architecture relies on the `DistSysJavaHelpers` project (see Subsection 2.3.4) that provides abstractions to represent arbitrary workloads as well as enabling loading of several well-known workload trace formats.

Here we will give an overview about introduced components and in the forthcoming sub-subsections, we will explain them in details. The `GA` component includes the concept of genetic algorithm, and it works with other components of the same layer to improve the process of generating traces based on averages and percentiles. The `User Behaviour` component enables the generator to scale-workloads with real user's behaviour. Finally, the `Standard Format` component supports the reusability of the generated traces by converting them to other formats that are supported by most simulators.

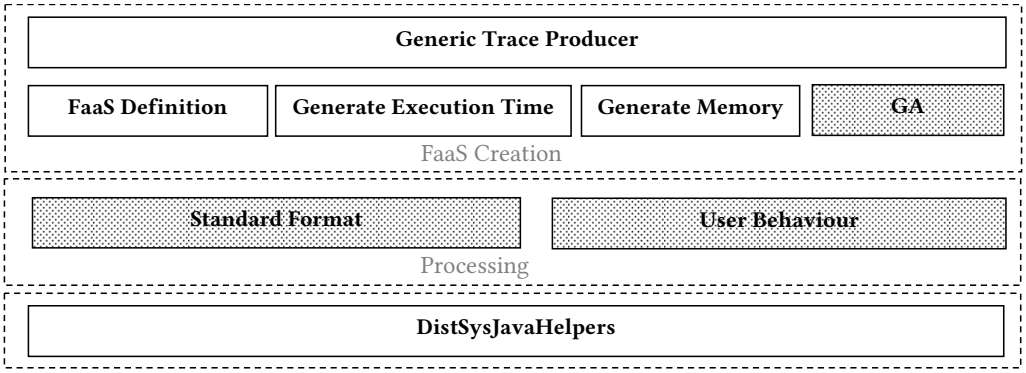


Figure 3.5: Architecture of generator approach

3.4.2.1 Improving the percentiles of generated traces

The previous approach generates one set of values (representing one individual in genetic algorithm concept) for each unique function and its invocations. This was based on the percentiles of execution time and memory utilisation, which determine how the values are generated and distributed. Although our approach generates values within the range of percentiles, good values of (R^2) between the generated and original could not be produced by a single iteration as they are generated randomly.

To address this issue, we introduce (`GA` component) to our previous approach, which applies the concept of a genetic algorithm. A genetic algorithm is a type of optimization technique based on the process of natural selection and genetics. It belongs to the larger category of evolutionary algorithms, which are computational techniques influenced by biological evolution. Genetic algorithm is frequently used for search and optimization problems, aiming to find optimal solutions within expansive and intricate search spaces [37]. It works in several steps: **Initialization** generates random populations (individuals or chromosomes) with each individual having the potential to be the best solution to the problem. **Selection** forms a new population from only the individuals that have good fitness. An individual's fitness is measured by its effectiveness in solving the problem at hand. **Crossover** combining pairs of selected individuals to create a completely unique individual (offspring) by exchanging genes in a manner similar to crossover in biological genetics. **Mutation** random changes are introduced in some individuals to simulate genetic mutations to maintain diversity in the population as well as assisting the exploration of new regions within the solution space. **Replacement** populations that include parents are substituted with their offspring to avoid overlapping between successive

generations. **Termination** extends the algorithm to perform these steps, whether the termination condition is met (finding optimal solution) or reaches the specified number of generations.

Thus, `GA` component produce several sets of values, and then it selects optimal values to constitute a single set for function and its invocations as shown in Figure 3.6. Our approach is able to generate a full large-scale trace or scaling-workload to the desired size (scaling-workload approach will be discussed in the following subsubsection). When the Azure dataset is selected, it will be processed by `Generic Trace Producer` to analyse dataset's contents. It reads each line from the dataset file (which represent one unique function and its invocations over a 24 hour period) and imitates the behaviour of the function according to invocations, as shown in Figure 3.6. This is done by extracting the necessary percentile information to be passed for `Generate Execution Time` and `Generate Memory` components via `GA` component. The `GA` component holds all concepts of genetic algorithm and enables a user to set configurations in terms of generation's number, individuals, mutation and crossover ratios, and fitness value that must be fulfilled, such as the value of (R^2) between the generated and original percentiles has to be greater than 0.99.

For each unique function and its invocations, the `GA` component produces sets of values according to the number of individuals. Each set of these represents generated execution time and memory utilisation values for the selected function and its invocations. For each iteration, the `GA` component will apply the selection, mutation, and crossover policies according to the configuration. This will lead to select the best amongst the whole selection of individuals that produce good R^2 value for the percentiles (meets fitness function's value) against the original percentiles of the Azure dataset. The process of iteration will continue, according to the selected number of generations that is determined by a user, which finally will concluded with optimal execution time and memory utilisation values for the selected function and its invocations.

As we explained in Subsection 3.3.1, `FaaS Definition` component will then instantiate invocation with the previously extracted values (e.g., amount of memory). Each single invocation involves several parameters such as unique ID, submitted time, execution time and memory utilisation, that determine the behaviour of the function. `FaaS Definition` will continue the generation of the invocations of the selected function according to its total number of invocations. After generating all the required function invocations for the simulator, `Generic Trace Producer` proceeds reading the next line from the dataset. This process continues till we finish generating all requested functions and their invocations.

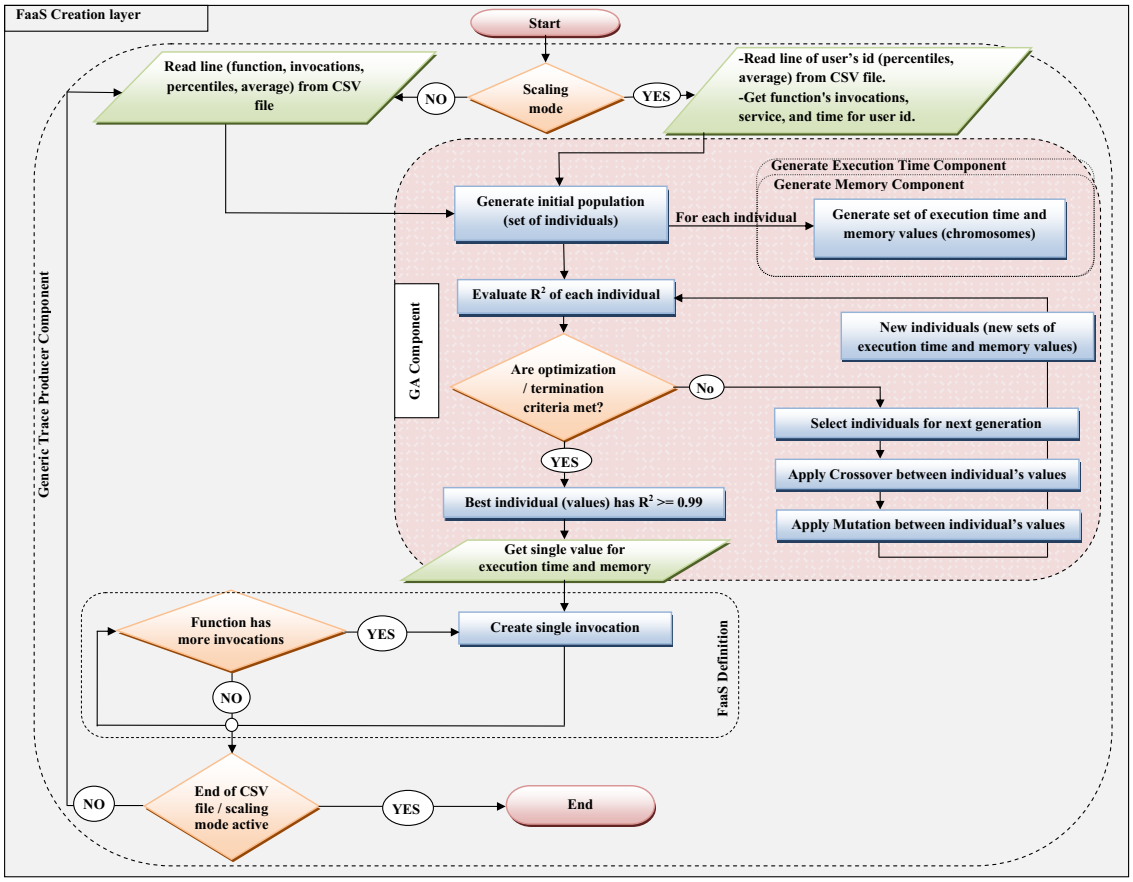


Figure 3.6: Flowchart for process of generating trace

3.4.2.2 Scaling workload with real users' behavior

Algorithm 1 Scaling workload with users' behaviour

1: **procedure** ScalingWorkload(dataset, workloadSize, service, time)
2: $totalTasks \leftarrow \sum_{i=0}^{N} nt_i$
3: $workload \leftarrow \emptyset$
4: **for** $id_{u_x} \in ID$ **do**
5: $involvedPerc \leftarrow ut(id_{u_x})/totalTasks$
6: $scaledTasks \leftarrow \lfloor involvedPerc \cdot workloadSize \rfloor$
7: $workload \leftarrow workload \cup genericTraceProducer(id_{u_x}, scaledTasks, service, time)$
8: **end for**
9: **return** $workload$
10: **end procedure**

Scaling-workload is essential to support researchers' scenarios that require small infrastructure to achieve their desired purposes, such as predicting the consumption behaviour of users to stimulate resource usage awareness. Therefore, we introduce `User Behavior` component that enables scaling workloads by detecting users' behaviour in terms of calculating the percentage of participation for each unique user in a dataset. The main logic behind this component is explained in Algorithm 1.

Before we introduce the algorithm, we discuss its foundational notation. It is based on the *dataset* that is read in the beginning to identify each user and its behaviour. The complete dataset of N lines is represented as: $dataset = \{line_1, line_2, \ldots line_N\}$. During scaling, we model each line as a tuple: $line_i = (id_i, nt_i)$, where id_i depicts the hash owner identified by the i^{th} line, and nt_i specifies the number of tasks (function and its invocations) listed in the particular line. User hashes can repeat over several lines in the dataset when the same user returns and uses the infrastructure for multiple functions. Thus, it could happen that $id_i = id_j = id_{u_x}$. To ensure correct user representation in our scaling approach, we collect the unique hashes (depicted as id_{u_x}) in the set of $ID = \{id_{u_1}, id_{u_2}, \ldots id_{u_K}\}$. During the loading of the dataset, we also determine each individual user's overall workload in the dataset. We use the following notation for this calculation: $ut : ID \rightarrow \mathbb{N}$, and we calculate the value of this function as follows: $ut(id_{u_x}) = \sum_{i:id_i=id_{u_x}} nt_i$, where id_{u_x} is a unique hash that we previously collected in the set ID.

Based on these definitions, our algorithm, first collects the total number of tasks in the complete dataset - see line 2. This goes through all the trace lines and sums up their individual task counts. Next, the algorithm also starts by initialising the

generated *workload* completely empty. This allows us to generate individual sub-sets representing particular users from the original dataset and then merge them together in the next phase.

After the preparatory steps, our algorithm goes through all unique user ids found in the original dataset and generates a sub-scaled workload for each of them. This is done by first calculating the involvement percentage of each user based on its total number of tasks (*ut*) and the total number of tasks in dataset (see line 5). After that, we calculate the expected number of invocations in the scaled workload for each user based on their participation percentage and the expected *workloadSize* (see line 6). Note, that we round the result down to the nearest integer ensuring that our scaled trace only contains user ids that would result in at least one generated invocation. Finally, in line 7, we generate the required number of invocations following our previously discussed `Generic Trace Producer` component as shown in Figure 3.6. This, then produces a single user focused statistically correct generated set of invocations that fits the desired scenario of the user of our scaler. These, unique-user-specific traces are then merged into the finally returned generated workload. To allow further customisation of the scaled traces, we enabled customising the scaled workload based on a particular time range or specific services. These are both passed in as the *service* and *time* parameters to both the scaler as well as the trace producer.

3.4.2.3 Converting generated trace to standard formats

`Standard Format` component enables the reusability of generated traces by converting them to other formats as standalone traces, and getting rid of the process of generating traces repeatedly. When the serverless functions are generated, this component obtains each function's invocation with all its attributes (e.g., execution time, tasks' id and amount of memory) to store in its repository. Then, for each invocation, it arranges its attribute values based on the desired format, as each one has its own ordering. Finally, it goes through this process till the end of functions, then it produces standalone trace file.

`Standard Format` component supports converting generated traces to several standard formats. This provides an alternative solution to other simulators that don't natively support the serverless model to take advantage from the generated realistic serverless traces. It also allows a user of the model to write its own format by arranging all attributes of generated functions based on the requirements of desired format.

`Standard Format` component also enriches serverless frameworks with FaaS work-

load by converting generated traces to AWS Lambda traces and other trace formats. However, some AWS trace attributes are not available in Azure dataset, such as cold-start, which demonstrate when a provider has requested a new instance for an invocation. To collect these attributes in real run time, our serverless model (see Section 3.2) that was devised by extending the DISSECT-CF, uses converted AWS Lambda trace as input for simulation to produce the necessary attributes that are unavailable. For each function's invocation in trace, our model will submit the function to the infrastructure to be simulated. If there is an instance ready to accommodate this function, it will simulate directly. Otherwise, the model will request a new instance (cold-start) for this function. The mechanism of requesting and terminating instances that affect producing cold-start attributes depends on the configurations that is selected by a user of model. When the simulation is finished, the model reproduces the AWS trace with complete attributes to be simulated by other serverless frameworks.

3.4.3 Evaluation of improved percentiles

To validate our approach of generating realistic traces with the help of the genetic algorithm, we configured the `GA` component setting according to an analysis provided in [107] that obtained by conducting a deep study to determine the best setting. We used 100 individuals and tournament selection with size equal to 10 individuals. Regarding crossover and mutation rates, they set to 0.9 and 0.05, respectively. Finally, the elitism strategy is exploited to copy the best individual from the current population into the next one without undergoing the genetic operators.

After that, we used the first day of the Azure dataset that contains around 36000 unique functions. We then used uniform randomly generator to pick up 5000 functions (the same numbers as in previous generator approach in Subsection 3.3.2), and we have generated 5000 invocations for each unique function based on the percentiles that disclosed in Azure dataset. As a result, large-scale of execution time and memory utilisation values were generated. For each function, we calculated those percentiles from generated 5000 invocations to see how close they are to the original percentiles of the Azure dataset. We used the coefficient of determination (R^2) between generated and original percentiles as fitness value to show data accuracy.

Although the selected functions produced wider range of values for the model and observed data compared to result in previous approach (see Subsection 3.3.2) as they have chosen randomly (each has different distribution values), we concluded with a very good result of R^2, which was 0.9994 for execution time and 0.9995 for memory utilisation as shown in Figure 3.7. This indicates the accuracy (and realism)

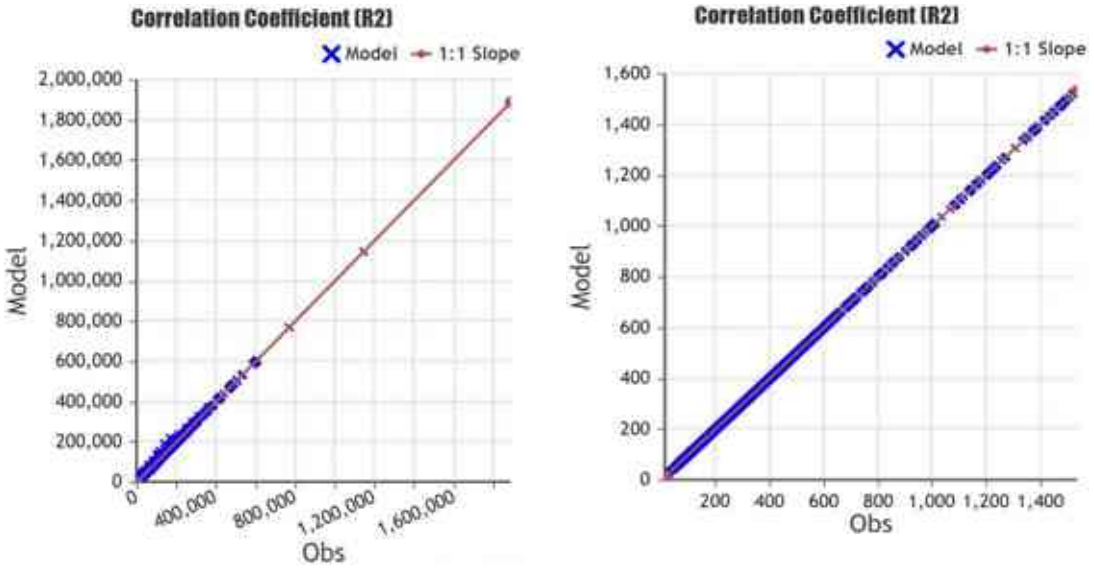


Figure 3.7: Coefficient of determination evaluation for generated percentiles with help of GA, for execution time (left-figure) and memory utilisation (right-figure)

of our approach and how genetic algorithm has enhanced the generated percentiles that affected the result of the previous generator approach.

3.4.4 Evaluation of users' behavior

To validate users' behaviour, we have chosen the second day of the Azure dataset, which contains comprehensive functions that executed in the Azure provider. Then, we invoked `User Behavior` component to analyse the selected file statistically. The `User Behavior` component demonstrated that the file came with around 853 million invocations, 36,456 services and 8,590 users. Moreover, it provided detailed information regarding each user's invocation number and the percentage of participation, as shown in Table 3.1. After obtaining these details, they will be passed to `Generic Trace Producer` component to be used for providing different workloads.

We validated the scaling approach by producing different workload sizes that fit small and large infrastructure configurations. We, then, invoked `User Behavior` component for statistical analysis of each generated workload. Finally, we compared the percentage of users' participation in all different workloads, with the original dataset by using R^2 as shown in Table 3.2. We also measured R^2 between the average of percentiles for execution time and memory utilisation for all generated workloads against the original one to show data accuracy. The results show that our

Table 3.1: Top 10 users by number of tasks submitted to the real provider

Rank	UserID	Number of Jobs	Percentage
1	U4932	127471686	14.94%
2	U376	98867247	11.59%
3	U5660	51372036	6.02%
4	U1746	49036404	5.75%
5	U3387	37036087	4.34%
6	U8104	22457567	2.63%
7	U6940	16743959	1.96%
8	U6143	15937341	1.87%
9	U2488	15808936	1.85%
10	U6175	14780372	1.73%
11	Other	403617380	47.31%
12	Total	853129015	100.00%

Table 3.2: Scaling workloads with real users' behaviour

Workload Size	R^2 (User's percentage)	R^2 (Execution time)	R^2 (Memory)
10^3	0.9999	0.9969	0.9986
10^4	1	0.9986	0.9984
10^5	1	0.9993	0.9989
10^6	1	0.9993	0.9981
10^7	1	0.9995	0.9997
10^8	1	1	0.9998

approach enables scaling workloads efficiently with the real users' behaviour. It also shows that the generated execution time and memory utilisation percentiles resemble the original values during scaling workloads.

The `Generic Trace Producer` component also enables a user of a simulator to generate workload with customized options, while maintaining real users' behaviour with help of `User Behavior` component to assist researchers to explore the behaviour of a particular service at a specific time. We generated a trace, as shown in Figure 3.8 for orchestration trigger service that contains numerous invocations during one day. We also generated a trace for the timer trigger, and we showed the participation percentage of users at the first and last minutes of the day, as shown in Figure 3.9.

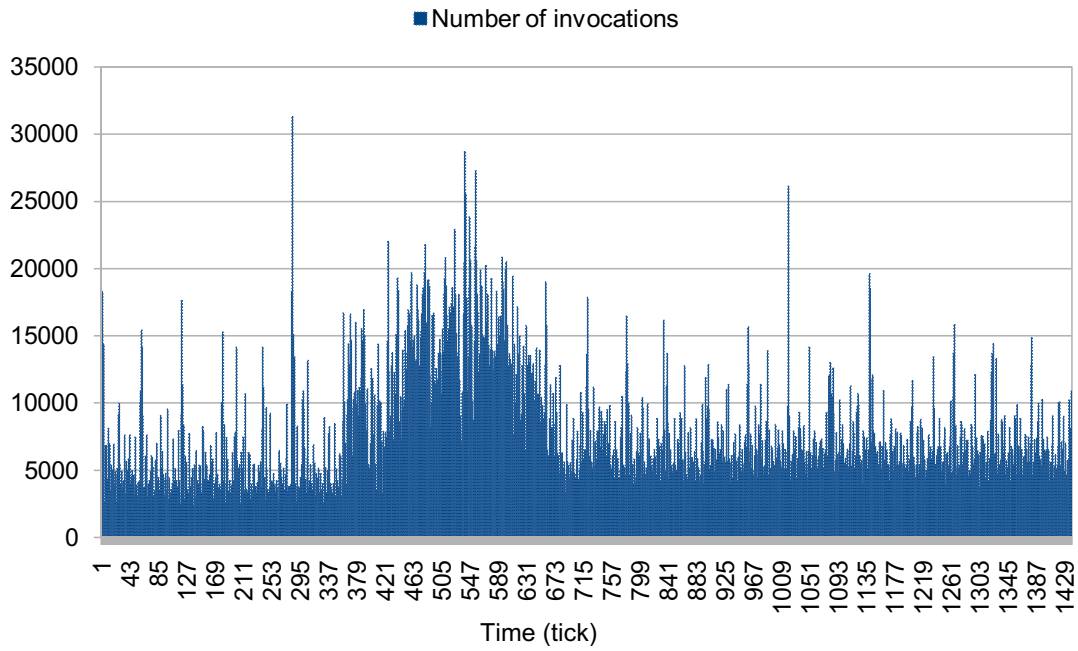


Figure 3.8: Number of orchestration triggers invoked in one day

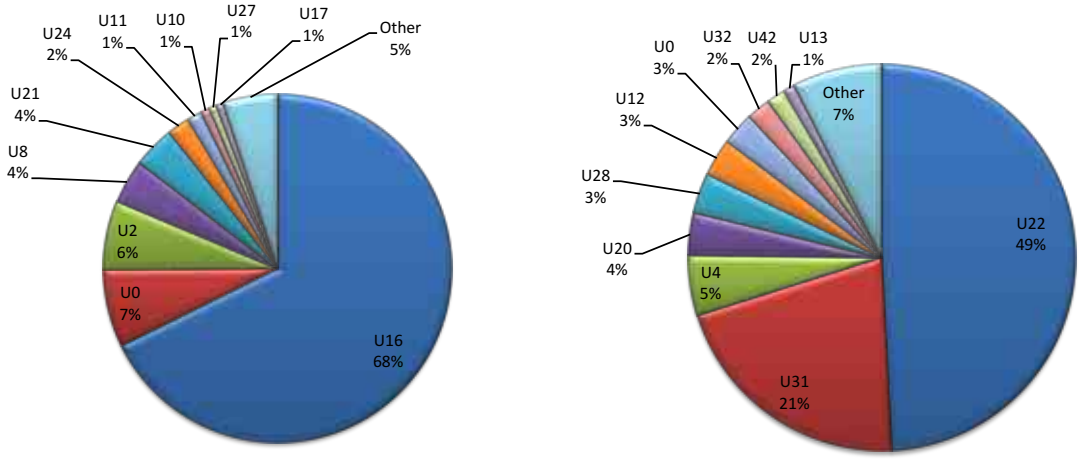


Figure 3.9: Percentage of invocations for timer trigger per user, Left-figure: first minute of the day, Right-figure: Last minute of the day

3.4.5 Evaluation of converting approach

Converting generated traces is beneficial to computing simulators that only support real traces and consider all the function's attributes in trace. To validate our converting approach, we have generated traces with different formats and simulated them by simulators that belong to different fields, namely, DISSECT-CF (cloud simulator), GridSim (Grid simulator) and simFaaS (serverless simulator).

3.4.5.1 DISSECT-CF

One of the approaches to verify the reliability of converting generated traces is monitoring the internal behaviour of the simulation while simulating the same tasks from different formats. Therefore, we have selected the DISSECT-CF simulator as it is enabling observing the internal behaviour of the infrastructure during simulation as well as offering precise results. DISSECT-CF will also be used as a standard to compare with other simulators in the forthcoming sub-subsections.

To show the accuracy of converting traces to other formats, we first configured the simulated infrastructure of DISSECT-CF for all experiments by setting up a homogeneous cloud with 100 PMs (each configured with 32 CPU cores, 256 GiB of memory and 256 GB of storage) and configured central data storage of 36 TB. Moreover, VMs were set to auto-scaling, and each one will be requested based on function attributes. Thus, the VM cannot process more than one invocation at the same time but can be used for the same function type if it is available.

Second, we generated 20 thousand functions (medium-scale trace that is suitable to above configuration) from the third day of original Azure dataset (Comma-Separated Values (CSV) file). We then asked to convert the same generated trace to Grid Workload Format (GWF) and Standard Workload Format (SWF) traces. First, we generated a workload from the Azure dataset (CSV) that was directly simulated by DISSECT-CF. In the second round, we simulated the transformed GWF and SWF traces. Finally, in order to prove that the two approaches are the same, they had to achieve our expectation of having less than (5%) percent difference in the internal behaviour. This ratio of differences is acceptable for the internal behaviour of the same simulator to obtain reliable results when we validate the conversion of traces using more than one simulator.

We have observed the internal behaviour of the simulated infrastructure in terms of simulated timespan (time that trace takes in simulated work not in real life), number of used virtual machines, average utilization of PMs and total power consumption for all experiments. Table 3.3 shows that simulation timespan and the number of used VMs are identical between the original trace (CSV) and converted

Table 3.3: Simulating twenty thousand functions with different formats

Metrics	CSV	GWF	SWF
Average utilization of PMs(%)	0.8119	0.8014	0.8014
Total power consumption (kWh)	73.5076	73.4451	73.4451
Simulated timespan (ms)	87792001	87792001	87792001
Number of used VMs	301	301	301

traces (GWF and SWF). However, there is a difference of 1.3% and 0.08% for the average utilisation of PMs and total power consumption, respectively. This meets our expectations and shows that our approach of converting is properly and realistic.

3.4.5.2 Other simulators

Enriching simulators with serverless traces offers an opportunity to simulate the behaviour of FaaSs, which enables researchers to conduct various experiments by simulators that belong to different fields. To validate our approach of converting traces and demonstrating their usability by other simulators, we explored the most popular open-source simulators in the computing field, as shown in Table 3.4, that could use these traces. We have considered two factors while exploring these simulators. First, the recent year these simulators were used by researchers for evaluating real computing scenarios. Second, the types of workloads that are supported by these simulators.

We have selected the most recent simulator used in the evaluation CloudSim [19] to validate our converted traces against the validated simulator DISSECT-CF. It is popular simulator and supports both real traces and generated synthetic workloads for simulation. We used the same cloud configurations mentioned in Subsubsection 3.4.5.1, for both DISSECT-CF and CloudSim. We generated traces in GWF and SWF formats originally for DISSECT-CF, and we then reused them for CloudSim. Unfortunately, the result was very different in terms of simulated timespan. CloudSim could not even be close to it. After investigating the factors that influenced the result, we have concluded that CloudSim does not consider the submitted time of a function (which represents the time of dispatch function to the real provider), and it dispatches all functions at the same time in the beginning of the simulation. This renders it unsafe for our expectations.

We then have selected the second recent and popular simulator GridSim [15] for simulating and validating our converting approach. GridSim considers all attributes of a task, such as execution time, submit time, memory utilisation and others, as

Table 3.4: Open-source simulators used recently by researchers in their evaluation process

Simulator	Type	Year Used	Workload supported	
			Trace file	Synthetic
CloudSim [19]	Cloud	2022	✓	✓
GridSim [15]	Grid	2022	✓	✓
iCanCloud [81]	Cloud	2021	✓	✓
DISSECT-CF [57]	Cloud	2021	✓	✓
WorkflowSim [25]	Cloud	2021	✓	N/A
CloudSched [109]	Cloud	2021	✓	N/A
CloudAnalyst [113]	Cloud	2021	N/A	✓
GreenCloud [60]	Cloud	2021	✓	✓
GPUCloudSim [101]	Cloud	2021	N/A	✓
SimGrid [23]	Grid/Cloud	2021	✓	✓
DFaaSCloud [50]	Serverless	2021	N/A	✓
BigHouse [77]	Cloud	2021	✓	✓
simFaaS [67]	Serverless	2021	✓	✓
CloudSimSDN [102]	Cloud	2021	✓	N/A
CEPSim [44]	Cloud	2021	N/A	✓
OpenDC Serverless [53]	Serverless	2020	✓	✓
FederatedCloudSim [61]	Cloud	2020	✓	N/A
EMUSIM [18]	Cloud	2019	N/A	✓
CloudReports [108]	Cloud	2019	N/A	✓
DCSim [58]	Cloud	2018	✓	N/A
ElasticSim [17]	Cloud	2017	✓	N/A
Cloud2Sim [56]	Cloud	2016	✓	✓

Table 3.5: Comparing simulating functions using DISSECT-CF and GridSim in terms of simulated timespan

Simulator	**1k**	**5k**	**10k**	**50k**	**100k**	**500k**	**1m**
DISSECT-CF	33.6s	2.79m	5.58m	27.9m	55.9m	4.65h	9.31h
GridSim	34.3s	2.81m	5.59m	27.9m	55.9m	4.65h	9.31h
Difference	1.95%	0.4%	0.2%	0.04%	0.02%	∼0%	∼0%

the same as DISSECT-CF does. We have simulated different workload sizes starting from small-scale trace (1 thousand functions) to large-scale trace (1 million functions) using DISSECT-CF and GridSim. As both were set to the same configuration and have simulated identical workloads, the timespan reflects the overall execution time of the simulation session. We have calculated the percent difference of simulated timespan for each workload, as shown in Table 3.5. The result shows the average percent difference is 0.37% between both simulators, which shows how accurately the traces are generated to match the original one and converted to standard formats.

3.4.5.3 SimFaaS

Our approach supports converting generated traces to various serverless workload formats that used by serverless simulators. To demonstrate that, we have selected simFaaS to simulate converted traces as it supports real traces and it was recently used for evaluating research as shown in Table 3.4.

SimFaaS [67] is a serverless framework built to optimize the performance of FaaS applications by predicting several Quality of Service (QoS) metrics accurately. SimFaaS used real-world traces from Amazon AWS Lambda to conduct experiments and extract performance metrics from simulation. These metrics include calculating the probability of cold start, the average number of idle instances and average utilization of instances.

As our generator approach can produce traces in different standard formats that meet a user's and a simulator requirements, we converted Azure dataset to AWS Lambda traces to be used as input to our rudimentary model (see Section 3.2). It then reproduces the AWS traces with complete attributes to be simulated by simFaaS framework.

Our model also provides performance metrics likes simFaaS, but it extracts them from simulation session. These include calculating the arrival rate of requests and counting the idle instances by calculating the difference between the total number of instances, and the number of instances that running at each second. Moreover,

it measures the probability of cold start by dividing the number of requests causing a cold-start, by the total number of requests made during the simulation. Finally, it measures the average of instance utilisation by counting the number of unique instances in the warm pool at each second.

To check the accuracy of our model output, first, we simulated the real-life traces from AWS Lambda (enclosed with simFaaS) using simFaaS simulator and our model. These traces contain around 500 thousand functions' invocations. As both simulators extract performance metrics from simulation, we measured the coefficient of determination (R^2) between the simFaaS and our model results to show data accuracy. For arrival rate, R^2 was 0.9999 and 0.9964 for cold start probability. R^2 were 0.9962 and 0.9982 for average utilisation and idle instances, respectively.

Second, we chose the third day of the Azure dataset. We then generated 15 AWS Lambda traces from this Azure dataset. For each trace, we have randomly selected thousand unique functions with their invocations. As a result, the generated traces contain around 500 thousand requests. After that, we simulated these traces with our model to enclose the attributes that are not available, as well as to extract the performance metrics from the simulation. Finally, we simulated the AWS traces that reproduced by our model as output, using simFaaS framework.

The performance metrics extracted from the simulations as shown in Figure 3.10, for both simFaaS and our model. To validate the converting approach, we measured R^2 of performance metrics for both. The results of R^2 were 0.9999, 0.9960, 0.9177, and 0.9525 for arrival rate, cold-start probability, average utilisation and average idle instances, respectively. This indicates the accuracy of our approach for producing traces that used by the simFaaS serverless simulator.

3.5 Summary

Simulators play a crucial role in the cloud computing, grid computing, and serverless computing fields by providing a flexible environment that could replace real providers in the research area. Imitating such serverless environments requires realistic traces that reflect users' behaviour in terms of execution history. Due to problems with existing IaaS workloads and their adaptability for use by serverless simulators, in this chapter, we proposed a novel approach to generate realistic serverless traces for enriching simulators that belong to different types of computing paradigms.

We have generated several traces to evaluate our approach. These traces involved functions with arbitrary invocation numbers to demonstrate the usability of our approach under different circumstances. We then validated our generator approach using the coefficient of determination (R^2) of the reported execution time and mem-

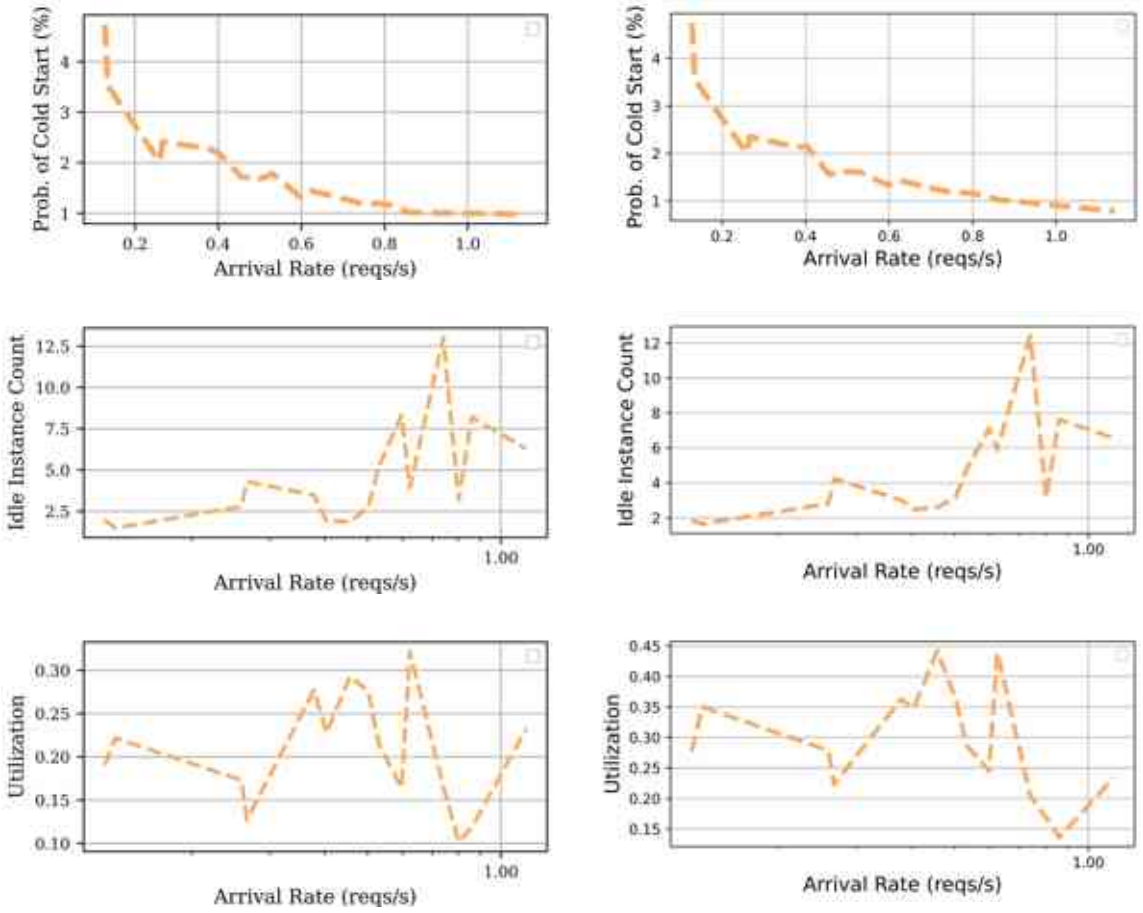


Figure 3.10: Performance metrics extracted from simulation, left-figures were generated by simFaaS, right-figures were generated by our model

ory utilisation averages and percentiles, between both the original Azure dataset as well as our generated ones. Our approach provided very good R^2 (> 0.99) values for predicting averages and percentiles.

To demonstrate the benefits of the generated traces, we introduced a rudimentary model for serverless systems based on DISSECT-CF to imitate the behaviour of Azure's Functions provider, and made several estimates and predictions of cost and utilisation. Our evaluation shows that the generated workloads are realistic and closely follow the behaviour of Azure's Functions as a service component.

In terms of scaling-workload with real users' behaviour, we have compared the generated traces to the originals in the following way. First, we calculated the percentage of users' invocations that reflect their behaviour over one complete day. Second, we randomly selected different numbers of functions to generate the different size workloads. Third, we calculated the percentage of users' invocations and percentile values of execution time and memory utilisation for generated workloads. Finally, we compared the already percentiles and calculated the percentage of user's behaviour from the original dataset with the generated ones with the help of (R^2). Our approach provided very good (R^2) (> 0.99) values for users' invocations with relatively well matching behaviour to the originals found in the Azure dataset.

Finally, we have demonstrated the benefit of the reusability of the generated traces by converting and applying them in a diverse set of simulators (belongs to cloud, grid and serverless fields) and shown that they offer reproducible results independently of the simulator used.

Chapter 4

An Extension of DISSECT-CF to Simulate Function-as-a-Service

4.1 Introduction

Serverless is a computing paradigm that relieves users from the burden of managing infrastructure and operations [70]. In the research community, simulators are the most common environments for evaluating algorithms and scenarios, as they provide easy-setup, low-cost and reproducible environments [80]. To support the research community's needs on serverless simulator and enabling evaluating FaaS scenarios, a simulator has to support the freshly introduced features, computing style, and resources constraints of serverless providers. These include: (*i*) auto-scaling resources to meet the demand of invocations, (*ii*) function-level execution-time monitoring, (*iii*) simulating realistic workload for precise results, (*iv*) associate triggers to invoke functions, (*v*) customising configurations of functions in terms of allocated resources, (*vi*) calculating costs based on selected configurations and function-level runtime, as well as providing statistical information about executed functions and internal infrastructure.

Unfortunately, there are no established simulation frameworks that can support research on the challenges accompanying serverless computing. The few serverless simulators that exist focus on specific functionality or aspects, but they could not comprehensively support the above listed features [50, 53, 67]. Thus, a comprehensive serverless framework able to mimic the behaviour of real providers is essential towards evaluating applications and scenarios reliant on the concepts of the serverless paradigm.

As serverless technology is based on an underlying IaaS that is abstracted from a user, it is beneficial to extend an existing IaaS simulator to support serverless features. This chapter focuses on introducing a serverless computing model to the DISSECT-CF simulator to enable simulating realistic Function-as-a-Service solutions. Our serverless environment provides an integrated environment (dubbed DISSECT-CF-FaaS) that fully supports all aforementioned features, first by, providing a generic cost-model, able to imitate serverless providers' cost policies. Second by, extracting performance metrics such as cold-start and warm-start probability that occur during a simulation. Third by, introducing the trigger concept to offer different ways to invoke a function. Fourth by, enabling customising configurations of providers. Finally, producing statistical information regarding the internal behaviour of provisioning and simulated tasks.

The remainder of this chapter is laid out as follows: Section 4.2 studies the limitations of our previous model. Section 4.3 demonstrates the architecture of the proposed serverless simulation environment. Section 4.4 exposes the internal mechanisms of DISSECT-CF-FaaS. Section 4.5 deals with experiments and scenarios to evaluate the services that provided by our serverless environment. Finally, Section 4.6 provides a summary of DISSECT-CF-FaaS's outcomes.

4.2 Extending our previous model

The foundation of modern computing technologies such as fog computing, edge computing, and serverless computing, are built on the concept of cloud computing. Thus, extending IaaS simulators to support other computing models is an essential step towards offering a versatile solution for the research community. In Section 3.2, we introduced a rudimentary model for serverless systems based on DISSECT-CF. However, it had noticeable limitations such as (*i*) focused only on Azure Functions providers, which limited its use for other providers and theirs policies, (*ii*) the quintessential concept widely used by FaaS systems is not offered, (*iii*) its cost model was not generic enough to support the newer providers, and (*iv*) essential, serverless focused performance metrics such as the number of provisioned instances at a specific time, are complicated to acquire.

4.3 Proposed architecture

To remedy the aforementioned limitations, we introduced new layers and components to the architecture of the previous model, in addition we updated the internal

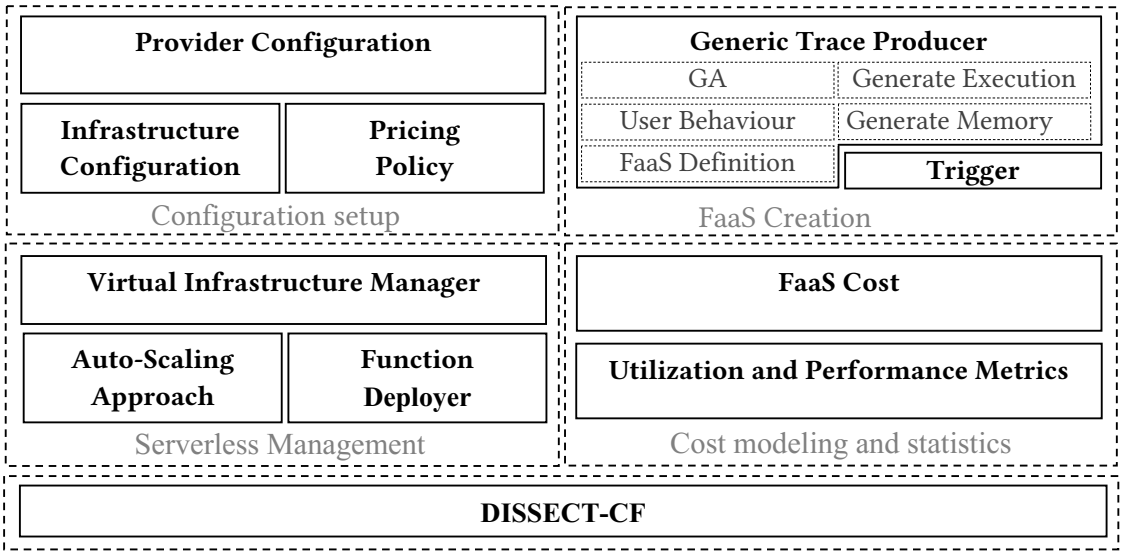


Figure 4.1: Architecture of our serverless environment

implementation of some components. This proposed architecture consists of four layers built on top of the core DISSECT-CF simulator (see Figure 4.1). Each layer is designed independently from others as possible. The architecture aims at enabling extensibility towards other serverless providers and triggers.

4.3.1 Configuration setup layer

This layer provides direct interaction with a researcher to establish underlying infrastructure and cost policies through provider selection and configuration. It consists of the three main components.

Provider configuration component allows choosing one of the providers with its policy that mimics the real one. In addition, it can be extended to include other providers or defining desired configurations. The provider's policy reflects the resource limits, regions, scaling approach, memory sizes and other constraints that will be applied during simulating functions.

Pricing policy component holds the cost of offered services for each serverless platform, such as AWS lambda [2]. This cost, is not only dependent on the provider, it varies depending on region selection, architecture, provisioned instance numbers, memory size and so on.

Infrastructure configuration component has updated (see Section 3.2) to include preset configurations of other real functions providers. It links with the in-

frastructure management subsystem of DISSECT-CF to establish relevant IaaS level components like data centres based on the selected configurations and provider.

4.3.2 FaaS Creation layer

In contrast to the configuration layer, FaaS creation focuses on the functionality running on top of the previously configured infrastructure. Primarily, this is done via the selection and customisation of a trace/dataset used for modelling serverless function behaviour. It also enables associating triggers to these functions. It consists of **Generate execution time**, **Generate memory**, **FaaS Definition**, **Generic Trace Producer**, **GA** and **User Behaviour** components that introduced in Section 3.2 and Subsection 3.4.2.

The process of generating is managed by **Generic Trace Producer** component, which is responsible for loading and reading the selected trace file and turns its content to functions and characteristic invocation patterns. This is achieved with the help of aforementioned components which are based on our previous research results on realistic trace generation (see chapter 3). The generation and trace loading is offered on several widely used datasets collected as traces of real-world infrastructure behaviour. These can have formats such as dataset file (CSV), standard grid (&), parallel workload formats (e.g., GWF), AWS Lambda traces and other.

Additionally, the **Trigger** component is introduced to provide various ways to tell what could cause the function to be invoked in the simulation. The mechanism of trigger to invoke functions is managed by event system of DISSECT-CF. Currently, apart from directly scheduling the function invocations as they are listed in the traces, it also supports timer and blob triggers. Timers triggers represent recurring function invocations with regular intervals. In comparison, blob triggers offer a capability to trigger a function invocation when there is a new blob deposited in the cloud storage. Such this concept enables researchers to conduct experimental patterns easily.

4.3.3 Serverless management layer

This layer represents the internal implementation that reflect the policy of serverless providers. This is achieved through the management of virtual infrastructures behind each serverless function created earlier. These virtual infrastructures provide just-enough resources for all function invocations throughout the simulation.

We upgraded the internal implementation of this layer from our previous rudimentary model (see Section 3.2). This update focuses on multi-provider support that let select one provider for each single simulation session, and then reflect its

scaling policy, limits, and mechanism.

Virtual infrastructure manager component is responsible for providing and managing the virtual infrastructure that backs the function invocations. It has a direct link with the Infrastructure simulation subsystem of DISSECT-CF to request resources such as function instances for simulating functions based on the configured FaaS provider's policy. Thus, it follows the selected provider and FaaSs configuration in terms of requesting resources for simulating functions and observing the internal behavior of the simulation.

Auto-scaling approach component provides a scaling mechanism that relies on a selected provider. It applies the provider's policy in terms of terminating function and function instances as well as keeping several function instances within the provider's expiration threshold. This component allows our environment to apply horizontal scaling to instances and other resources within the provider's policy.

Function Deployer component handles all generated functions to virtual infrastructures by dispatching each to available function instances inside the managed infrastructure.

4.3.4 Cost modeling and statistics layer

In our final layer, we provide components to estimate the cost of the workload that passed through the simulation. This layer consists of two components, namely, **FaaS Cost**, which follows cost model's policy of the configured providers to calculate the total cost of the simulated function invocations. As each provider has its own policy to calculate the cost, this component acts according to the selected provider and applies its cost model to simulated functions.

As the first step, this component obtains the cost of selected services and configurations from **Pricing policy** via **Infrastructure configuration**. It then calculates the number of invocations (requests) of all functions that occurring during the simulation session. Next, it gets the actual running time of functions from **Utilisation and performance metrics**. Finally, it collects the consuming resources (e.g., used memory) to apply the cost model of the selected provider. For instance, if the AWS provider is selected by a user, this component calculates the number of requests and their durations for the functions. Additionally, it will calculate the allocated memory for all functions, and then it obtains the cost of these to estimate the total cost.

Utilisation and performance metrics component provides statistical information and essential performance metrics about the internal behaviour of the IaaS, and virtual infrastructures that back the workload, to allow the in depth research analysis of the impact of the various scaling, invocation trigger end similar strate-

gies. In addition to support the introduced performance metrics in Section 3.2, this component enables calculating average concurrent number of instances, average life-time of instances, average running-time of instances, and average idle-time of instances. Finally, it can be extended by researchers to include other further metrics.

Analysing the internal behaviour of the IaaS and all provisioned resources in terms of starting and terminating time within simulation, is mainly managed through the event system of DISSECT-CF

4.4 An illustrative walk through of our extensions

Figure 4.2 demonstrates a typical simulation scenario with DISSECT-CF-FaaS. The flow of this scenario is marked by arrows and each arrow shows the sequence of steps that a user of the simulator would likely take.

The first step towards simulating FaaS is selecting a provider such as AWS. This leads our extended simulation environment (i.e., our model) to establish the provider's policy internally. Each provider has its own configuration and `Provider configuration` component stores these configurations that mimics the corresponding real providers. Our model also supports extending to another serverless provider via this component. In this step, we allow a user to customise resources for simulating functions by specifying expected infrastructure and virtual infrastructure configurations based on modelled functions, users, or just in general in the overall simulation session.

After selecting the provider and its configurations, in step (2), these are forwarded to the `Infrastructure Configuration` component. This obtains the cost model of the selected configurations and services from the `pricing policy` component to be ready for estimating the total cost at the end of simulation session, in step (3). The cost of services relies on selected criteria such as region/data centre, architecture and memory size. With this additional information, it is now ready to then establish the underlying infrastructure with help of DISSECT-CF, to accommodate the forthcoming virtual infrastructure of step (4).

At the same time, in step (5), `Generic Trace Producer` component begins loading the user selected trace/dataset file. This reads the trace line-by-line, then extracts and generates the distribution functions of each essential attribute (e.g., runtime, invocation frequency). These are then passed to the `FaaS Definition` component to create FaaS functions in step (6). If a user defined functions with trigger during step 1, the `Trigger` component will be associated to the functions in during step (7).

Whenever all the desired functions are defined, they are stored in the repository of the `Generic Trace Producer` component, as shown in step (8). Next, all

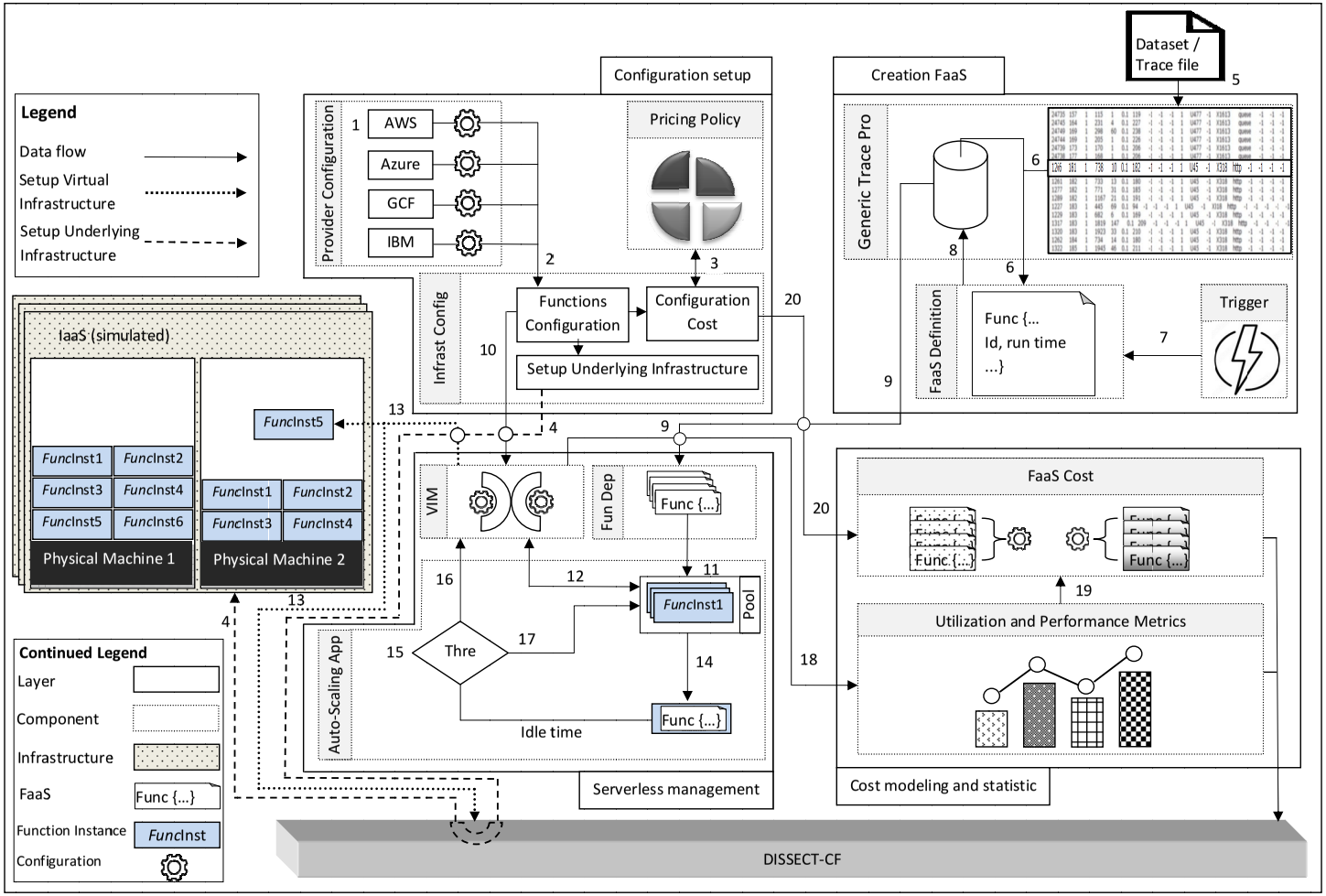


Figure 4.2: A demonstration scenario executed on our extended simulation environment

functions will be forwarded to `Function Deployer` component to be ready for simulation in step (9). Whenever the functions reach `Function Deployer` component, the `Virtual Infrastructure Manager` component gets the configurations for these functions from `Infrastructure Configuration` component, see step (10). Here the process of simulation inside the core DISSECT-CF will commence as well by dispatching each function to user selected `Auto-Scaling` component in step (11).

In this scenario, the modelled provider is not preparing warm function instances, thus at the beginning of the simulation, there are no available function instances in the pool to accommodate the dispatched functions. Thus, `Auto-Scaling Approach` will notify `Virtual Infrastructure Manager` to request a function instance based on the selected provider's policy and function's configuration (12). The `Virtual Infrastructure Manager` component will handle the instance requests with the help of DISSECT-CF (13) that mimics the provision of the instance. This manager is also responsible for destroying a function instance when it is expired (e.g., due to the instance not having invocations dispatched to it in the past 10 minutes). With the aforementioned policy, the first invocation will take significant time to dispatch (see step (14)) due to initialising, loading and registering function instance, which eventually leads to a cold-start. A user can avoid cold-start in some providers, such as AWS Lambda by selecting provisioned concurrency that warms-up instances to be ready before any invocations get dispatched. However, this service incurs an extra cost to the bill according to AWS provider's policy.

While the function invocations are executed, our model maintains the execution timeout and it will terminate any function instance that exceeds the threshold of the selected provider. After a function invocation is completed, the function instances might be idle for while and here `Auto-Scaling Approach` component calculates the idle time to determine the necessary state of the instance as shown in step (15).

If the idle time exceeds the provider instance expiration threshold, the simulator requests the destruction of it via the `Virtual Infrastructure Manager` component as demonstrated in step (16). Otherwise, in step (17), the idle instances will join the pool to allow their reuse and increase the ratio of warm start invocations. By adding instances to the pool, we also consider the per instance maximum scaling numbers defined by the provider's pre-configured policy. Once all function invocations complete, `Utilisation and Performance Metrics` component obtains detailed information about simulated functions and their instances from the `Virtual Infrastructure Manager` in step (18).

This component will start to calculate performance metrics such as running time, arrival rate, cold-start and warm-start probability, average life-time of instances, average idle-time of instances, and average concurrent number of instances with the

help of the event system of DISSECT-CF. I.e., it injects introspection points to the simulated infrastructure to get notifications about instance, invocation, utilisation state changes. In step (19), these stats are then passed to the `FaaS Cost` component to estimate the costs that would likely to incur if such simulated workload would happen on a real-life infrastructure. `FaaS Cost` will first get the price of the functions' configurations from the `Infrastructure Configuration` component as per step (20).

4.5 Experiments

A laptop (Intel (R) Core (TM) i7-4600U CPU @ 2.10GHz (4 CPUs), 2.7GHz, 8 GB) was used for the evaluation of our extension. In this evaluation section, we evaluate the effectiveness of our extension by producing services that resemble the AWS Lambda and Azure Functions providers, as they are most popular serverless providers [84]. Moreover, mimicking the providers' policies in terms of resource constraints and associating triggers to serverless functions to reflect the realistic internal behaviour of serverless computing.

4.5.1 Evaluation cost model

Estimating the cost of serverless workload sheds light on how a provider charges a user based on actual services used, which is the main concept behind existing serverless technology. It also enables researchers to evaluate several computing cost scenarios. As each serverless provider has its own policy to calculate the price, our serverless environment is able to mimic the behaviour of providers and reflect their policies to estimate the cost of simulated workload.

To evaluate our generic cost-model, we have imitated AWS Lambda provider by selecting AWS provider that leads our serverless environment to apply its policy internally. AWS Lambda provider offers users different memory sizes to simulate functions. The cost of services increases based on the allocated memory size that also leads to proportionally increased CPU allocations. Apart from memory size, the cost of service varies according to the selected architecture and region.

We have generated 100k realistic serverless function invocations by selecting the first day of Azure dataset as input to our serverless environment. We then have conducted experiments to estimate the cost of the same workload but using different memory configurations. We have used memory sizes, namely, 128 MB, 512 MB, 1024 MB, 2048, MB and 3072 MB, with architecture (x86) and region (frankfurt) as it's the closest one to our location. The Table 4.1 shows that we obtained similar cost

Table 4.1: Simulating 100k invocations using different memory sizes

Size	Cost($)	Average run time(ms)	Cold-start(%)	Warm-start(%)	Arrival Rate(s)
128	3.553	17329	1.059	98.941	0.0036
512	3.503	4321	0.614	99.386	0.0073
1024	3.517	2156	0.555	99.444	0.0103
2048	3.625	1114	0.523	99.477	0.0145
3072	3.504	717	0.510	99.490	0.0151

for all experiments but with different running time as expected. The reason is AWS Lambda provider offers a user trade-off between cost and running time. Increasing memory size is costly but leads to reduced running time and cold-start probability. Thus, a user can have a fast or slow run time with almost the same cost.

As each provider has its own cost model to calculate the cost, our environment applies the same cost model to simulate functions. For the AWS used here, the cost model is based on the number of invocations, selected memory size, and average running time of the simulated functions provided by our environment. As AWS [2] provides estimating cost based on the aforementioned parameters, we validated the obtained results in this way to show the accuracy of the results.

4.5.2 Evaluation of provisioned concurrency

The AWS Lambda provider offers a user option (provisioned concurrency) to minimise the cold-start probability that leads to sped up invocations. This option charges a user based on the selected number of provisioned concurrency. This prepares function instances in advance to be ready when a function is dispatched without causing cold-start. We conducted experiments to reflect the behaviour of AWS Lambda provider when this option was selected. We have simulated the same workload as previously. We limited the memory size to smallest 128 MB and evaluated the costs and performance with and without the provisioned concurrency option. We have set the number of provisioned concurrency to one for each function type to show minimum influence of this option.

We validated the results of the Table 4.2 against the AWS estimating cost model that shows our simulations can imitate the AWS Lambda policy to reduce cold-start and timespan with extra cost when the user chooses a provisioned concurrency to simulate workload. As demonstrated, the workload can benefit from warm-start, both in terms of execution time as well as reduced cold-start percentage.

Table 4.2: Simulating 100k invocations using the AWS provider with instances having 128 MB memory

Provisioned	Cost($)	Cold-start(%)	Warm-start (%)	Timespan(tick)
No	3.553	1.059	98.941	27861501
Yes	3.821	0.846	99.154	27361501

4.5.3 Evaluation of trigger

A trigger is used to provide different ways to invoke a function. The trigger type determines the situation of event that needs to be met before an invocation takes place. According to [97], the timer was one of the most heavily used triggers in the Azure dataset. The timer trigger is set based on an interval that defines how frequently the function is invoked.

To investigate the timer trigger in our serverless environment, we first imitate the Azure provider's policy by selecting a consumption plan for all functions. We, then, generated varying sized realistic traces and conducted an experiment that involves five different groups of functions. Each group consists of 500 types (medium scale) of functions and has different invocation intervals determined by their triggers as shown in Table 4.3. Each function (in each group) is invoked using timer trigger, these invocations then end up having function-typical realistic attributes such as execution time and memory utilization that was derived from the Azure dataset with the help of our generator.

We ran the functions of all groups for one simulated day. The result shows that the probability of cold-start for the group of functions that has the smallest interval is less than for other groups. As our environment observes the status of instances and measures their life-time, it allows more frequent reuse of their instances than those belonging to other function groups. This eventually reduces the idle time of these instances in warm-pool. Therefore, they will not exceed the expiration threshold of the selected provider to be terminated by our environment.

4.5.4 Evaluation of performance metrics

Our serverless environment enables extracting essential performance metrics from the simulation sessions based on users that existed in trace, type of simulated functions, or overall session. These metrics harder to come by in real serverless providers in smooth way. The performance metrics reveal the internal behaviour of the imitated provider in terms of applying its policy, constraints and backend provisioning resources for understanding internal mechanism.

Table 4.3: Using timer trigger for five groups of functions with different trigger intervals

	Interval (s)	Prob. of cold-start (%)	Prob. of warm-start (%)	No invocations
Group1	30	0.1945	99.8054	1440000
Group2	60	0.2461	99.7538	720000
Group3	300	1.0277	98.9722	144000
Group4	600	1.1013	98.8986	72000
Group5	1200	2.3555	97.6444	36000

Table 4.4: Average performance metrics of the instances were extracted while simulating different workload sizes

Workload size	Concurrent no	Lifetime(s)	Running-time(s)	Idle-time(s)
1k	65	1239	40	1199
10k	67	1381	197	1183
100k	98	1481	323	1158
200k	155	1541	408	1133

We have demonstrated our performance metrics through workloads sized. Each single session, we generated workload size between 1-200k function invocations from Azure dataset. We then simulated this workload by mimicking the Azure Functions provider and observed its average provisioned number of instances concurrently as well as average life-time of instance (when it starts for the first time and then eventually terminates), actual running-time and idle-time of instances for the overall simulation session.

The result shows that our serverless environment provisioned more instances when the workload size was heavy, as shown in Table 4.4. The reason is Azure Functions provider can scale up to 200 instances concurrently per function-app type. The results also show that the utilisation of instances (running time) is increased when workload size is heavy. Note that, even though the idle instance lifetime in Azure Functions bounded to 12 min, the average lifetime of instances exceeds the constraints. The reason is when the instance reused for simulating function, its starting time will be updated, and our serverless environment records the lifetime of the instance from the first time requested until it is terminated. Thus, the instance could have worked for hours within the provider's limits. Another reason is, according to [111] each provider has its own policy to keep several instances in warm-state for a while.

4.6 Summary

In this chapter, we extended the DISSECT-CF simulator to offer the services unique to current serverless technologies. We offer an extensible simulation framework in a comprehensive environment that enables realistic mimicry of the behaviour of commercial providers.

We have designed several experiments to evaluate our serverless extension. These experiments were conducted by imitating services and features of AWS Lambda and Azure function providers. In the AWS Lambda provider, our experiments evaluated the cost model by simulating the same workload with different memory sizes. They also show how the selection of provisioned concurrency reduces the cold-start probability and simulation time. In Azure Functions, we have investigated the timer trigger by simulating five groups of functions (each group contains 500 functions) with different time-span. We have also designed an experiment to extract performance metrics and internal behaviour of the serverless environment that occur during simulation.

In terms of cost-model, our serverless environment provided the expected results by estimating similar costs for all different memory sizes but with different running time. Our DISSECT-CF-FaaS properly reflected the AWS provisioned concurrency policy to reduce cold-start with extra cost. The trigger behaviour was successfully captured by our serverless environment and reflected in Azure Functions environment that avoided idle instances when functions were heavily invoked. Finally, our serverless environment extracted the performance metrics such as average concurrent instances, actual running-time and idle-time of involved instances from the simulation session.

DISSECT-CF-FaaS demonstrated its performance towards simulating the aforementioned small and medium-scale scenarios as its execution is limited to a sequential manner. However, large-scale scenarios such simulating millions of functions are required parallel execution.

Chapter 5

Parallel Event System to Reveal the Internal Behavior of our Serverless Environment

5.1 Introduction

As DES simulators gained significant popularity to support and evaluate cloud computing environments, there are several obstacles that stop increasing the performance of these simulators. First of all, most are designed to execute sequentially [16] that could not be appropriate for the state of the art scenarios, such as simulating millions of service invocations and their interactions in serverless computing situations.

Introducing a parallel approach to the core event handling in IaaS simulators, would be the first step towards scaling their performance efficiently to meet the newest challenges in the field. Moreover, it will benefit extensions that are built on top of them.

Always applying parallel execution to the simultaneously occurring events in the simulation does not necessarily lead to a well scaling DES though. When only a few events occur simultaneously, sequential execution is often times beneficial as we can avoid the overheads of parallel constructs. Otherwise, the parallel execution can lead to better performance. Thus, the necessity of determining at a specific simulated time instance, whether the events will execute sequentially or parallel, is crucial to increase the performance and to avoid unnecessary overhead.

This chapter introduces the parallel execution to its most abstract system of DISSECT-CF: the event system. The new event system automatically switches be-

tween the new parallel mode and the old sequential one based on the number of simultaneous events that occur at a given time instance. The parallel executor divides and distributes the simultaneous events equally over the available processors and balances the load across the system. These two operations reduce idle CPUs (or cores) behind the simulator. This parallel version is able to foster the execution time of our serverless environment by extracting the performance metrics in parallel manner.

The remainder of this chapter is as follows: Section 5.2 covers the existed challenges and issues of PDES. Section 5.3 looks to investigate the occurrence of simultaneously events. Section 5.4 presents our methodology of employing parallelism in the event system of DISSECT-CF. Section 5.5 deal with experiments that are designed to evaluate the scalability and performance of the new approach. These experiments focus on the core functionality and time management mechanisms of the event system in DISSECT-CF. Section 5.7 summarises the conducted results of our parallel approach.

5.2 PDES issues and challenges

The main goal of PDES [39] is to enhance the execution of discrete event simulations by efficiently exploiting the capabilities of high-performance computing platforms. This can be achieved by taking advantage of the natural parallelism that occurs in developed models, applications, or systems. Despite PDES field has gained significant popularity over the last decades, it faces several challenges and issues arising from the need to adapt and deal with the modern architecture of underlying hardware, the complexity of systems, and the development of software [49].

The PDES issues are arisen from applying various strategies and principles to decompose a simulation to be processed on available processors when there is an opportunity to exploit the parallelism available in a sequential program, underlying problem, or even the architecture of underlying hardware. These strategies have to be designed and developed to get the most out of them by ensuring loading balancing fairly among all processors, scheduling events properly, and offering synchronisation mechanism.

Besides the general PDES issues, there are other world-wide challenges that make developing parallel simulation efficiently is hard [38] such as creating large-scale simulations to support the behaviour of real-world irregular systems, developing scalable and efficient techniques to map PDES computation to modern hardware architectures such as Graphics Processing Unit (GPU) to achieve better performance, and

simplifying the development of PDES to encourage developers to shift towards creating parallel models.

As the subsystems of DISSECT-CF are designed to work independently as much as possible, this give privilege to simplify introducing PDES to the core of the simulator (event subsystem) as a foundation for parallel simulation. Based on the architecture of DISSECT-CF, several of the aforementioned issues can be considered when developing a new parallel event system of DISSECT-CF. The parallel version has to resolve fundamental issues, such as exploiting the parallelism available at each particular tick via detecting all simultaneously occurring events. Second, exploit all processors of the platform and offer initial load balancing by dividing the list of events into sub-lists to be distributed equally over available processors. Finally, foster the execution of regular and irregular events that occur during the simulation. In the forthcoming sections, we will explain in detail how a parallel event system is applied and the mechanism of work.

5.3 Prominence of recurrent events

Most cutting-edge technologies such as serverless computing have designed to allow multiple events such as serverless functions (FaaSs) and their invocations or computing tasks in cloud, happen at the same time to obtain better performance [62]. Its paradigm encourages users to execute events in a parallel by managing the backend infrastructure on their behalf. Moreover, it provides essential concepts such as a timer trigger that leads events to occur simultaneously within the serverless computing provider. Thus, the serverless technology provides an elastic environment to invoke functions in parallel.

In Azure Functions provider, the collected datasets reveal to which extent the serverless functions are happened simultaneously. We analyzed the available 14 days of Azure datasets [97] to estimate the average number of functions that were executed in parallel. For each dataset representing a one day, we collected the total number of functions' invocations, and we then divided the number by 1440, as invocations were binned at 1-minute. Finally, we divided the concluded number by 60 to calculate the average executed functions per second. The Figure 5.1 shows how intensively functions were executed in parallel during 14 days by users the worldwide.

When we come to simulation environment to mimic advanced technologies mechanism, the architecture of simulator is critical word to demonstrate its capability towards parallelism. It could be the architecture of a simulator is designed with parallel in mind, or having extraordinary features, such as ability to recurrent events based on specific time, to be used as a foundation towards parallel execution.

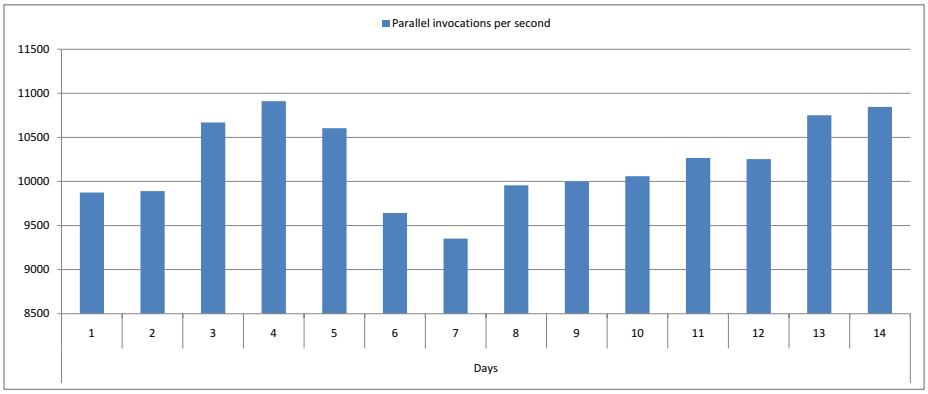


Figure 5.1: The average number of serverless functions executed simultaneously in Azure Functions provider

Although, the selected simulator DISSECT-CF reduces the execution time of equal quality/detail simulations done compared to several other simulators in the field, it still does so in a sequential fashion. Therefore, we aim to set the foundations to support simulations that require high performance.

The lowest (event) subsystem of DISSECT-CF has two main classes: (*i*) the `Timed` class, used for recurring events; and (*ii*) `DeferredEvent` class used for irregular events. Recurring events are invoked regularly based on a specified frequency. Thus, recurring events can subscribe notifications, when subscribing, an event frequency must be specified to determine how many ticks must pass to get repeated notifications. As the higher level subsystems of the simulator are built on the top of mostly recurring events, our target is enhancing the performance of this subsystem. We expect that this will positioning affect the performance over the rest of the framework and its extensions. The event sub-system had a sequential execution design. Based on the existing API of DISSECT-CF, parallelisation could happen for executing of simultaneously happening events (i.e., events that should happen in the same time instance or tick of a simulation).

To understand such simultaneous events, we have provided a simple example scenario with three event objects with various frequencies (this demonstrates events derived from the `Timed` class of DISSECT-CF which allows defining events that can happen repeatedly). Table 5.1 shows the basic details of our simple example scenarios. The first three rows show the event objects and their behaviour. The fourth row shows the time instances in our simple simulation. In the table, we can see for every time instance when the events will be processed. E.g., the second event

Table 5.1: Three events with different frequencies

Events	Freq	Next events of e1, e2 and e3 based on their frequencies(Freq)														
e1	1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
e2	3	3	3	3	6	6	6	9	9	9	12	12	12	15	15	15
e3	5	5	5	5	5	5	10	10	10	10	10	15	15	15	15	15
Time		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Degree(%)		33	33	66	33	66	66	33	33	66	66	33	66	33	33	100

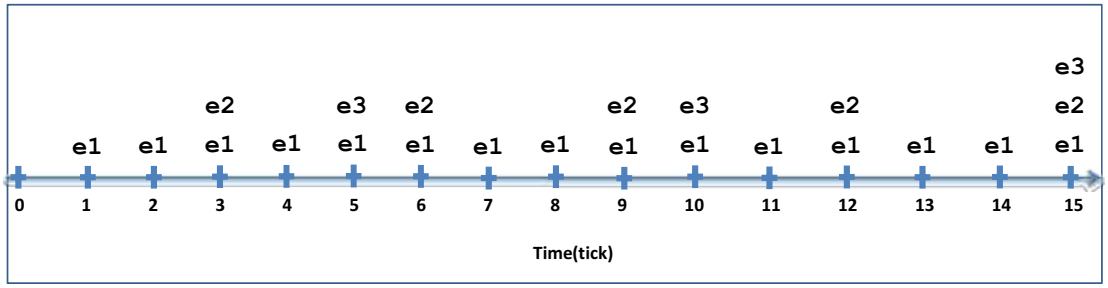


Figure 5.2: Representing multiple events in Table 5.1 occur at a specific time

($e2$) is processed in time instances 3,6 etc. Figure 5.2 shows how the event queue will look like at any particular time instance in case we execute the events defined in the previous table.

The degree of parallelism denotes the number of events (overall percentage in the last row) occurring at a specific time instance (tick). Which mainly depends on the frequencies of subscribed objects that determine how frequently these events occur. When all subscribed events happen at a specific time, the degree of parallelism is 100%. When half of them occurs at one time, the degree is 50% and so on. Thus, the degree of parallelism varies according to the occurrence of events at each tick. Therefore, the average degree of parallelism in a single simulation run is deduced from all ticks for the whole system. In Figure 5.2, there are 15 simulated time instances, out of these 7 are having parallel events, making the example's average degree of parallelism 50.66%. If we execute simultaneously occurring events (e.g., $e1$ and $e2$ in time instance 3 in the figure) in a sequential fashion, then we pay a penalty of using a sequential simulator. This observation will guide the next the subsection where we discuss how we identify these kinds of events and how we execute them in parallel.

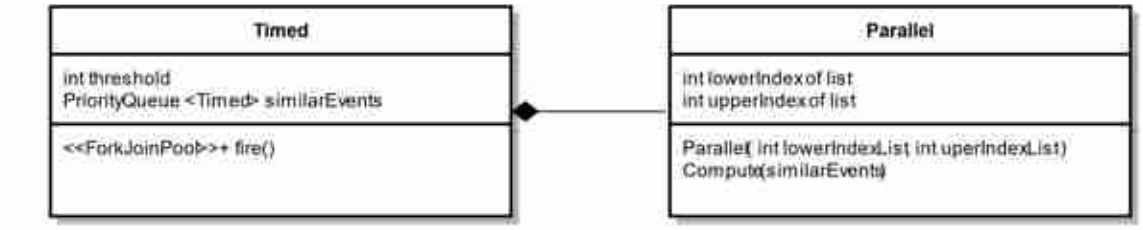


Figure 5.3: Diagram of Timed class and Parallel class

5.4 The parallelisation of simultaneous events

Figure 5.3 shows the basis of our extension. The diagram shows only the relevant parts of the original `Timed` class, and the new `Parallel` class. The new class is created as an inner class within `Timed` class, to ensure easy access to the original data structures within the event subsystem's main class. The user of the system is still expected to interface with the existing methods of the `Timed` class (thus all previous extensions to the simulator would benefit from our parallelisation approach). Note that inside the simulator, all higher level subsystems (e.g., those which simulate VMs) are users of the `Timed` class. As parallelisation is automatically executed depending on the state of the event queue, the higher level subsystems could benefit from the improvement on `Timed`.

Algorithm 2 Determining the need for parallelism

```
1: threshold = specified size
2: list = all simultaneous events
3: if list.size ⩽ threshold then
4:     while list.notEmpty do
5:         event = get single event from list
6:         Execute event
7:     end while
8: else
9:     execute Parallel(list.lowIndex, list.upperIndex)
10: end if
```

The events taking place in a particular tick are handled with the `fire()` method (see Figure 5.3). Our approach changes the behaviour of this method by introducing Algorithm 2. Here we first collect the list of simultaneously occurring events at each particular tick (see line 2) – note that this list was not needed for the sequential sub-

system as that would only work with one event at a time. As a result, the collection of this list is an overhead of the new parallel algorithm. The discussed approach below aims at minimising this overhead.

Our new `fire()` method now checks the size of the list to determine if we need to execute in sequential or parallel fashion. The old, sequential execution is shown in the loop of line 4, this is still kept and used if we have too few simultaneous events in the queue. The parallel execution utilises our new `Parallel` class to distribute the work over threads, that will be created implicitly according to the number of available cores. This is done by passing the `lowerIndex` and `upperIndex` that specify the indices of first and last elements of the list (see line 9). The `threshold` (minimal size of the list which leads to parallel execution) is configurable by the user of the simulator. To aid the user determining the threshold, an auto-tuning approach is also going to be offered for the threshold which determines its value when suitably long running simulations are executed.

Algorithm 3 Mechanism of Parallel class

```
1: Procedure Parallel( list.lowIndex, list.upperIndex )
2: lowerIndex = list.lowIndex
3: upperIndex = list.upperIndex
4: Funct compute ()
5: if upperIndex - lowerIndex ⩽ threshold then
6:     while list.notEmpty do
7:         Execute events of list
8:     end while
9: else
10:    midIndex = (lowerIndex + upperIndex / 2)
11:    execute all (Parallel(lowIndex, midIndex), Parallel(midIndex,
       upperIndex))
12: end if
```

After the decision to parallelise, the actual parallelisation is organised by the `Parallel` class according to Algorithm 3. Instances of this class are executed in their own threads. Thus, they will likely run on another CPU core compared to the original `fire()` method. When a `Parallel` instance is instructed to compute, it again uses our the previously discussed and determined `threshold` value to decide if the workload assigned to the thread is sufficiently small or not.

If the sublist of simultaneous events is short enough (see line 5), the sublist is executed in the current thread. This sublist execution is done just like the sequential

one discussed before (see Algorithm 2's line 4). But instead of going through the entire list of simultaneous events, now we have a shorter list to process which was assigned only to the thread of this `Parallel` object in the parallel invocations of Algorithms 2 and 3.

In contrast, when there are more simultaneous events than a single thread should handle, we sub-divide the list of events based on its size in equal parts and pass them on to further threads (see line 11). We recursively perform this process until the list of events is divided into sublists (sublists size become less than or equal threshold) and all threads have sufficiently short lists, then the threads are scheduled according to a fork-join model. This list division method ensures that we execute on all available processors in the current machine and also offers an initial load balance. Although each thread has an almost equal number of sublists, work-stealing approach ensures that the threads workloads are almost equal to avoid wasting resources. Note that the proposed approach also supports parallelism of irregular events as they happen somewhere during the simulation, but to which level, it depends on how intensively they happen and how the queue of events appears at particular times.

5.5 Evaluation of the parallel event system

Intel High Performance Computing (HPC) platform was used for the evaluation of our parallel DISSECT-CF. It has the following specifications: Intel (R) Core (TM) i7-8700 CPU @ 3.2GHz (6 cores + 6 hyper threaded cores), 64GB memory, 1T SSD, 1T HDD, Debian Linux Buster 10.4, OpenJDK 11.0.6. We have designed several scenarios to test the performance of the parallel version by focusing on time management while ensuring complete control over event occurrence. We also made sure the evaluation was validating the parallel version: we used the complete API of the `Timed` class to verify if the parallel version produces results matching output from the unmodified sequential code.

5.5.1 Validation of the parallel event system

To ensure that the behaviour of our evaluation is following real life simulation patterns, we have instrumented the `JobDispatchingDemo` class of the dissect-cf-examples project. This class was already validated before to produce realistic simulations e.g., comparable to CloudSim (see [57]). Our instrumentation focused on how the realistic simulation utilises the lowest abstraction layer of DISSECT-CF. We measured, the degree of parallelism, the typical event behaviour, the number of events in total and the average execution time of a single tick method call in

nanoseconds (i.e., the single event workload). To enable the comparison, we have also instrumented our parallel `Timed` class in the same way allowing us to acquire the typical workload of our synthetic tick methods.

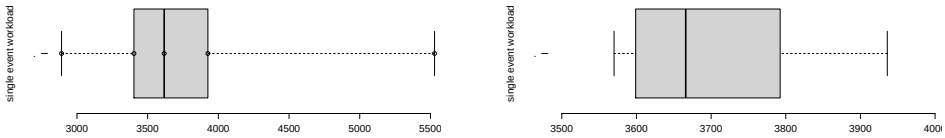
We have set up our realistic simulation with `JobDispatchingDemo` as follows: (*i*) maximum number of jobs that exist in parallel was set to 2; (*ii*) the amount of seconds the job startup times was set to 10; (*iii*) minimum execution time of a single job was set to 10s; (*iv*) maximum execution time of a single job was set to 90s; (*v*) minimum and maximum gaps between the last and the first job submission of two consecutive parallel batches were set to 200s; (*vi*) minimum number of processors for a single job was set to 1; (*vii*) maximum number of processors for a single job was set to 2; (*viii*) total number of processors usable by all parallel jobs was set to 4; (*ix*) total number of jobs was 100000; (*x*) the number of nodes was 5000.

To allow our evaluation to focus at the lowest abstraction layer, we set out to capture the event workload behaviour of the above complex simulation, but with a synthetic workload. This synthetic workload ensures that we do not run complete simulations all the time and that our parallelism evaluations not to be distracted by upper layer behaviour. Our synthetic tick method (implemented in `TimeRandomGenerator` class), does a busy waiting loop by calculating the following formula:

$$SyntheticEventWorkload(size) := \sum_{i=0}^{size} \left(2e^i\sqrt{i}\right) \mod \left|\left\lfloor \frac{i+5}{i+1} \right\rfloor\right| \tag{5.1}$$

, where *size* can control the single event workload, while the denoted operations ensure that the distribution of the single event execution time is closely matching the above mentioned more realistic simulation. Thus, the synthetic Equation 5.1 can model the captured behaviour of realistic large-scale simulations by producing the same weight of the single event workload. Note that the formula is selected to make a single event workload have the same behaviour as a realistic simulation, and any other formula can be used as long as it makes the loop busy to capture the same behaviour.

To ensure that the workload produced by this busy waiting loop is equivalent to the realistic simulation, we have executed the same number of events we have recorded in the realistic simulation and repeated the measurement 100 times. The repetition allowed us to collect several statistical properties of the single event workload in both the synthetic and the realistic simulations. We present our findings for the realistic simulation in the box plot of Figure 5.4a. Our best approximation of this realistic workload was captured by our synthetic workload parametrised with $size = 49$.



(a) The execution time(ns) of single event workload using JobDispatchingDemo class

(b) The execution time(ns) of single event workload using our experiments classes.

Figure 5.4: Boxplot diagrams for JobDispatchingDemo class and our classes

Figure 5.4b shows the behaviour of our best approximate synthetic workload. Our median duration is within 3% of the realistic. The distribution of our workload is a bit narrower and more even, but the upper and lower whiskers of our synthetic experiment are within the typical range of the realistic simulation's values. As a result, from this point onwards, we will refer to synthetic workloads set up with this particular parameter as the *original* single event workload.

Note, that later we have evaluated the system with other single workloads. For example, changing the *size* to 147 (49 x 3), leads to a threefold increase in single event workload compared to the realistic setting. In contrast, changing it to 16 (49/3), leads to a three fold reduction in single event workload again compared to the realistic setting. These two values will be the extremes used in Figure 5.5.

5.5.2 Performance of the parallel event system

Our evaluation scenarios create 35,000 recurrent event objects. The object count was set so the minimum execution time of the sequential version is at least 5 minutes, allowing sufficient time for the parallelisation to take effect. The recurring events subscribe with different frequencies so we have control over the degree of parallelism. We provided controls to these scenarios, so we can easily adjust the degree of parallelism (through event subscription changes) and the single event workload (through changing the *size* in Equation 5.1). The evaluation scenarios are publicly available in the ParallelTimed package released in the dissect-cf-examples project on GitHub[1].

The invocation of `Parallel` class depends on the threshold value (see Algorithm 2 for details) to determine the maximum length of the event list processed by a single thread. To determine the ideal setting for the threshold, we evaluated our solution with four different values: 8, 16, 32 and 64. We have also generated recurring events with four different degrees of parallelism as shown in Table 5.2. Based on our analysis

[1] https://github.com/dilshadsallo/dissect-cf-examples

Table 5.2: The execution time(s) of parallel version using four different sizes of list

Degree of Parallelism				
Threshold	**25%**	**50%**	**75%**	**100%**
8	235	412	549	657
16	234	411	548	657
32	231	408	545	654
64	229	406	541	652

of the execution times in the table. Even though the differences are not big, it is recommended to use a threshold equal or exceed 32 to enhance the performance.

With the respect to the number of cores, there are two factors that influence the performance of the parallel version. First, the degree of parallelism plays a significant role and it is shown in Table 5.3 that the parallel version can significantly improve performance. We evaluated both the parallel and sequential versions of the simulator with four different degrees of parallelism (25%, 50%, 75%, 100%). Even though the evaluation of this scenario has been done with the same number of aforementioned objects, the number of events that occur, and the number of events that occur at the same time significantly increase. This is because we simulated for the same amount of simulation time, but with increasing subscription frequency each object receives more event notifications. E.g., to increase the degree of parallelism on the scenario in Table 5.1, we can change the subscription frequency of event two to 1. In this example, the degree of parallelism increases to 73%, but we see more event notifications delivered as we will have 15 notifications for event two as well.

With regards to Table 5.3, in 25% of parallelism, the parallel version runs 1.72 faster than the sequential. When the degree reaches 50%, the ratio increased to 1.74. The parallel version executes simulations 1.84 faster than the sequential version when 75% of all subscribed events occur recurrently during a simulation time. Finally, the parallel version reaches 2.01 times faster than the sequential version when the degree of parallelism is 100%. Even with a high degree of parallelism and using multi-core, we cannot use all cores because there is still a performance cost such as coordinating threads that introduced by multi-thread compared to a single-threaded approach.

Now let's analyse the effect of the size of the single event workload (as per Equation 5.1). We tested both of the parallel and sequential versions with various single event workload sizes, commenced with threefold lower than the *original* one to show the behaviour of simulating very low single event workload. Then reaching to threefold higher than the *original* single event workload to demonstrate the advantage of parallel version as shown in Figure 5.5. When the single event workload is threefold

Table 5.3: The execution time(s) of parallel and sequential versions in four different degrees of parallelism

Degree of Parallelism				
Version	**25%**	**50%**	**75%**	**100%**
Sequential	379	717	989	1312
Parallel	220	410	543	651

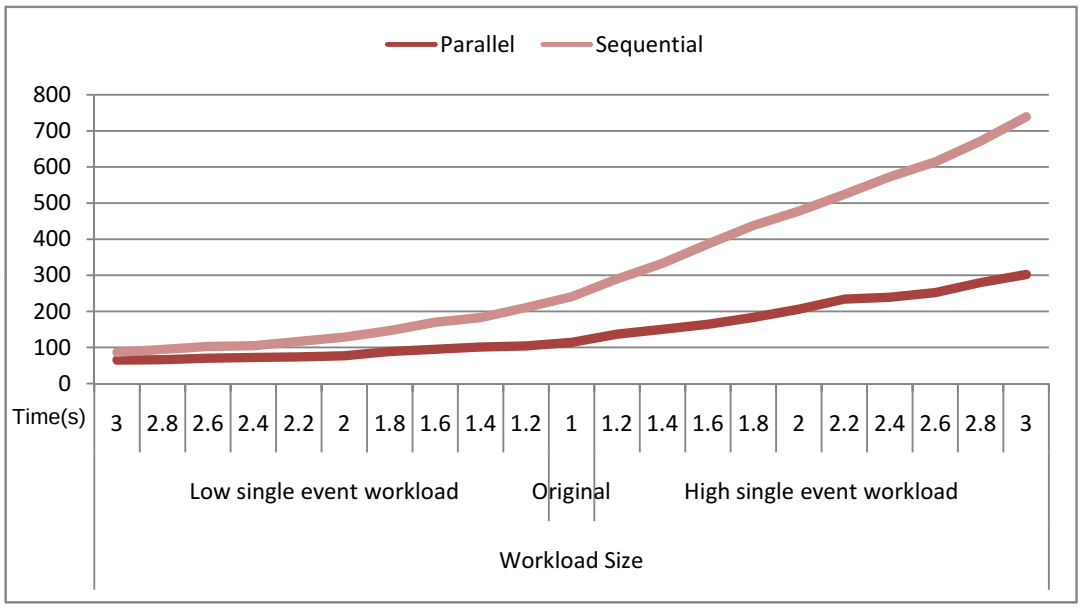


Figure 5.5: The execution time in seconds of different workload sizes simulated by parallel and sequential versions

lower than *original* one, the parallel version runs 1.33 faster than the sequential version. This ratio increases to 1.67 when the single event workload is two times lower than the *original* single event workload. The parallel version even runs 2.11 faster than the sequential version using *original* single event workload. When the single event workload size doubled, the parallel version executes the simulation 2.32 times faster than the sequential version. The ratio increases to 2.44 when the single event workload size becomes threefold higher than the *original* one.

Thus, the parallel version speeds up the performance of simulation by using the additional cores of the host. The biggest advantages of the parallel version can be exploited when there are larger simultaneously occurring event queues and when the

single event workload is larger as well.

5.6 Evaluation of our serverless environment using parallel event system

One of the main concept that attracted users towards serverless computing is paying only for actual use, no need to pay for idle VMs and other resources such as traditional cloud computing. Thus, the serverless provider accommodates many users simultaneously and its resources could be shared among them effectively.

The nature of the task in serverless computing is also central to increasing the opportunity of invoking a provider at a specific period. The tasks in serverless are usually lightweight and are not executed for a long time compared to the tasks in cloud computing. This gives a chance to execute numerous tasks in a period such as a one day. According to the traces that collected from real providers, 942 million tasks were executed in one day using a serverless provider [97]. Whereas in one month, the maximum number of tasks executed by many cloud computing providers is 13 million [7].

Such these state-of-art scenarios that involve large number of functions require parallel simulation and execution to handle the requests properly. Regarding simulations, particularly in DES environments that produce precision results rapidly, executing these scenarios sequentially often leads to losing the benefit behind existing simulators. Thus, simulating many functions or revealing their behaviour within a simulation environment requires parallel execution that exploits all the cores behind the simulator.

We exploited our parallel version to foster the aforementioned scenarios that happen in our introduced serverless environment in the chapter 4, particularly the **Cost Modeling and statistic** layer that mainly depends on the event system of DISSECT-CF. Our parallel version can be advantageous to these scenarios that require high performance, such as revealing the internal behaviour of provisioning resources and estimating the cost in our introduced environment.

To demonstrate the performance of our parallel event system, we have conducted experiments on different workloads that require faster processing. We have generated workloads with different numbers of functions' invocations, namely, light-workload (100 thousands), normal-workload (1 million), and heavy-workload (10 million) by using our generator that we introduced in chapter 3. We then imitated the AWS Lambda provider by setting all functions to the following configurations: memory size (3082 MB), region (Frankfort), and architecture (x86). After that, we simulated

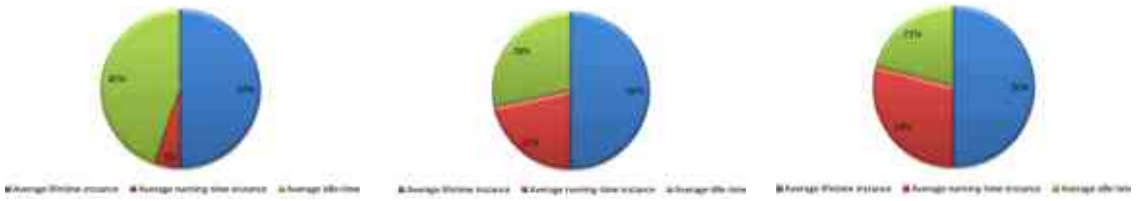
Table 5.4: Extracted performance metrics of different workloads using our serverless environment

Performance metrics	Light	Normal	Heavy
Cold-start probability (%)	2.542	0.385	0.135
Warm-start probability (%)	97.458	99.614	99.864
Arrival rate (s)	0.031	0.115	0.442
Average execution time of function (ms)	512	476	626
Number of unique functions	1256	1639	4660
Estimated cost ($)	2.503	23.264	305
Average Number of Concurrent instances	389	770	3913
Average lifetime instance (s)	1218	2199	6671
Average running-time instance (s)	126	947	3882
Average idle-time (s)	1092	1251	2789

these workloads by our serverless environment in HPC platform 12 threads (see 5.5). We then analyzed the internal behavior of each simulation session, one times using the sequential event system and other using parallel event system. In the parallel version, the threshold was set to 32, as we recommended in the Subsection 5.5.2. Finally, we have collected the extracted performance metrics of each session, including the duration of the simulation in real-time.

As both sequential and parallel versions produce the same simulation results (except execution time), Table 5.4 lists the performance metrics extracted from simulation sessions for all different workloads. Although the number of unique functions in heavy workload is almost fourfold than in the light-workload, it is obvious that the probability of cold-start is a low in heavy-workload compared to other. The reason is that the functions are dispatched intensively to be simulated in the heavy-workload (see arrival rate), and this leads to reuse of the available instances again and again. Which eventually, they are not exceeding the expiration-threshold to be terminated. As consequence, the average percentage of running-time of instance is increased in heavy-workload compared to other different workloads, as shown in Figure 5.6. This also has influence to increase the number of provisioning instances in our serverless environment during simulation. The result shows that it is reaches up to 3913 concurrent-instances for all functions in heavy-workload, whereas it is 389 in light-workload, as shown in Table 5.4.

When we come to the simulation time for these workloads, as we previously mentioned in Section 5.6, the workload size and the degree of parallelism have significant influence in demonstrating the benefit of using the parallel version. As the num-



(a) Light-workload (b) Normal-workload (c) Heavy-workload

Figure 5.6: Percentage of average life-time, running-time and idle-time of instance for different workloads

Table 5.5: The execution time in seconds of our serverless environment using sequential and parallel versions

Version	Light-workload	Normal-workload	Heavy-workload
Sequential	56	850	23204
Parallel	45	378	8703

ber of functions' invocations in heavy-workload is huge compared to light-workload, the probability of invocations to invoke simultaneously increases and leads to a high degree of parallelism. This also has an effect on the single-event workload.

Table 5.5 shows the execution time of simulated light, normal, and heavy workloads (the results listed in Table 5.4) in our serverless environment using sequential and parallel versions. The results show that the parallel version runs 1.24 faster than the sequential version during analyzing and simulating light-workload. This ratio increased to 2.24 when the workload was normal. Finally, the performance of the parallel version reaches 2.66 in heavy-workload compared to the sequential version. The results show that the parallel version demonstrates better performance for scenarios that engage heavy workload.

5.7 Summary

In this chapter, we have introduced a parallel event system to DISSECT-CF simulator for fostering the execution of scenarios that require high throughput. We have validated our parallel version by designing experiments that had independent control on the following four properties: (*i*) event independence (no influence on future events); (*ii*) pattern of events throughout a simulation (i.e., how many events do we have in total and when should they happen); (*iii*) number of simultaneous events

(degree of parallelism) happening at an average time-simulated instance; (*iv*) the single event workload (i.e., how compute heavy is a particular event). We instrumented and measured the behaviour of realistic simulations in terms of these properties. Then, we implemented simple synthetic event patterns (that are only exercising the event system of DISSECT-CF) for the simulator, which we calibrated to imitate the properties of the previously measured realistic simulations.

To ensure the quality of our experiments, we collected the synthetic event pattern's properties with the same measurement approach that we applied for the realistic setting to compare and analyse them. We also evaluated with random event patterns to test the behaviour of the parallel version under unforeseen conditions. With respect to the number of cores, evaluation results show that two factors have affected the performance of the parallel version. First, if we have at least two simultaneous events for more than 50% of the simulated time instances, then the parallel version already runs two times faster than the sequential. Second, increasing the single event workload leads to 2.44 times faster simulation execution than sequential.

Finally, we used a parallel version with our introduced serverless environment to evaluate real serverless workloads. We have generated different workloads, namely, light, normal and heavy, and we then simulated them using sequential and parallel versions. The result shows that the parallel version can speed up the process of simulation up to 2.66 compared to sequential when the workload is heavy.

Chapter 6

Conclusion

6.1 Summary

Simulators play a crucial role in the computing field by providing a flexible environment that could mimic real providers in the research area. As serverless computing is in its infancy, the research community needs to explore this promising cloud paradigm in a simulation environment to evaluate FaaS scenarios, and explore potential on architectures, operations, and mechanisms that could foster this computing paradigm.

In this dissertation, we proposed the DISSECT-CF-FaaS serverless environment. This integrated environment is capable of generating realistic traces that closely matches the original dataset's characteristics in terms of execution time, memory utilisation as well as user participation percentage. The evaluation in chapter 3 showed that our generator approach provided excellent values for predicting generated trace attributes and users' invocations compared with the behaviour in the real-life dataset.

Our serverless environment is able to mimic the provisioning of resources, services, while also mimic the policy of the most well-known serverless providers. It also reveals the internal mechanisms and behaviours of the imitated providers by extracting performance metrics during simulation sessions. In chapter 4, our evaluation showed that our environment provided the expected experimental results by, first, estimating the costs for various memory configurations. Second, reflecting provisioning policy properly to reduce cold-start. Third, capturing the behaviour of the trigger successfully. Finally, extracting average concurrent instances, running-time, and idle-time of involved instances from the simulation session.

Our DISSECT-CF-FaaS offers parallel execution to foster the scenarios that re-

quire high performance in computing, such as revealing the internal behaviour of the simulation session by extracting performance metrics. The evaluation of the parallel version in chapter 5 showed that the execution of the simulation session can be sped up by 2.66 times compared to the sequential version. Thus, DISSECT-CF-FaaS is able to meet the expectations of the research community towards experiment various FaaS workloads and scenarios in a versatile environment.

6.2 Contributions

The new scientific results that are achieved during the completion of the project summarized in the following three theses.

Thesis 1 **Related Publications:** [92, 93]

I proposed a novel approach for generating realistic serverless traces to enrich cloud computing simulators with varying characteristic workload types. My approach applies a genetic algorithm to produce and select the best generated functions' attributes that resemble the behaviour in a real-life dataset. It also enables scaling-workload to fit desired scenarios while maintaining the users' behaviour disclosed in the real-life dataset. Finally, it supports the reusability of the generated traces in other computing simulators by adapting the traces to popular formats.

Thesis 2 **Related Publications:** [89, 90, 92, 94]

I proposed a comprehensive serverless extension (DISSECT-CF-FaaS) to the research community for evaluating a wide range of real-case FaaS scenarios in an environment that imitates commercial providers' behaviour. This environment is capable of capturing real behaviour services to enable establishing a cost model, offering scaling up-down function instances, introducing a trigger mechanism comparable to real usage behaviour, and applying constrained on provisioning resources. It also extracts performance metrics from the simulation session to reveal how the internal behaviour of provisioning resources responds while serverless functions are simulated.

Thesis 3 **Related Publications:** [88, 91]

I proposed a new parallel event system to foster the execution of DISSECT-CF-FaaS towards simulating large-scale scenarios. The introduced parallel version increases resource utilisation capability by allocating all available cores for backing the cost modelling and statistics of our serverless environment. The advantage of the parallel version is demonstrated when the simulated workload involves a large number of simultaneous events.

6.3 Future works

We have identified three future research directions, namely, first we have to investigate other simulators that could use different trace formats, and exploit our generator to shift the generated realistic trace to these formats. To demonstrate to which extent our introduced approach can support these formats as well as to introduce modifications to adapt with them.

Second we hope to introduce other triggers such as the http trigger that enable DISSECT-CF-FaaS to interact and communicate with real applications to support other serverless scenarios such as dependent tasks in microservices applications. Finally we aim at focusing on the simulator's second most heavily used component in DISSECT-CF: the unified resource-sharing subsystem. This subsystem has high compute complexity, and its parallelisation will enable our model to rapid estimation of resource sharing on even larger scale-distributed systems. Applying these will lead to the seamless transition of the entire DISSECT-CF-FaaS into simulating billions of service invocations and their interactions in serverless computing situations.

Chapter 7

Author's Publication and Software Availability

7.1 Author's publication

[88] Sallo, D. H., & Kecskemeti, G. (2020). Parallel Simulation for The Event System of DISSECT-CF. In the 12th Conference of PhD Students in Computer Science: Volume of short papers Szeged, Hungary, University of Szeged, pages 58-61.

[89] Sallo, D. H., & Kecskemeti, G. (2020). Towards a DISSECT-CF extension for simulating Function-as-a-Service. In the 16th MIKLÓS IVÁNYI INTERNATIONAL PHD and DLA SYMPOSIUM abstract book, Pécs, Hungary : Pollack Press, ISBN: 9789634295785.

[90] Sallo, D. H., & Kecskemeti, G. (2021). Introducing Serverless Computing Model Based on DISSECT-CF Simulator. In the XXIV. Spring Wind Conference abstract book, Association of Hungarian PHD and DLA Students. Miskolc

[91] Sallo, D. H., & Kecskemeti, G. (2021). A Parallel Event System for Large-Scale Cloud Simulations in DISSECT-CF. Acta Cybernetica, 25(2), 469-484. **Scopus Indexed [Q3]**.

[92] Sallo, D. H., & Kecskemeti, G. (2022, June). Towards Generating Realistic Trace for Simulating Functions-as-a-Service. In Euro-Par 2021: Parallel Processing Workshops: Euro-Par 2021 International Workshops, Lisbon, Portugal, August 30-

31, 2021, Revised Selected Papers (pp. 428-439). Cham: Springer International Publishing. **Scopus Indexed [Q2]**.

[93] Sallo, D. H., & Kecskemeti, G. (2023). Enriching computing simulators by generating realistic serverless traces. Journal of Cloud Computing, 12(1), 1-13. **Scopus Indexed [Q1]**.

[94] Sallo, D. H., & Kecskemeti, G. (2023). Towards a DISSECT-CF extension for simulating function-as-a-service. International Journal of Parallel, Emergent and Distributed Systems, 1-13. **Scopus Indexed [Q3]**.

7.2 Software availability

The source code of this project is open and available (under the licensing terms of the GNU Lesser General Public License 3) at the following website:

Serverless trace generator: https://github.com/dilshadsallo/DistSysJavaHelpers

Serverless environment: https://github.com/dilshadsallo/dissect-cf-examples

Bibliography

[1] Apache openwhisk. open source serverless cloud platform, 2023. https://openwhisk.apache.org/, Last accessed on October 30, 2023.

[2] Aws lambda pricing, 2023. https://aws.amazon.com/lambda/pricing/, Last accessed on April 04, 2023.

[3] Fission - open source, kubernetes-native serverless framework, 2023. https://fission.io/, Last accessed on October 30, 2023.

[4] Ironfunctions - open source serverless computing, 2023. https://open.iron.io/, Last accessed on October 30, 2023.

[5] Kubernetes Native Serverless Framework, 2023. https://github.com/vmware-archive/kubeless, Last accessed on October 30, 2023.

[6] OpenFaaS - Serverless Functions Made Simple, 2023. https://www.openfaas.com/, Last accessed on October 30, 2023.

[7] Parallel Workloads Archive, 2023. https://www.cs.huji.ac.il/labs/parallel/workload/, Last accessed on January 21, 2023.

[8] Pricing - Functions : Microsoft Azure, 2023. https://azure.microsoft.com/en-us/pricing/details/functions/, Last accessed on January 21, 2023.

[9] Paarijaat Aditya, Istemi Ekin Akkus, Andre Beck, Ruichuan Chen, Volker Hilt, Ivica Rimac, Klaus Satzke, and Manuel Stein. Will serverless computing revolutionize nfv? *Proceedings of the IEEE*, 107(4):667–678, 2019.

[10] Arif Ahmed and Abadhan Saumya Sabyasachi. Cloud computing simulators: A detailed survey and future direction. In *2014 IEEE International Advance Computing Conference (IACC)*, pages 866–872. IEEE, 2014.

[11] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[12] Mohammad S Aslanpour, Adel N Toosi, Claudio Cicconetti, Bahman Javadi, Peter Sbarski, Davide Taibi, Marcos Assuncao, Sukhpal Singh Gill, Raj Gaire, and Schahram Dustdar. Serverless edge computing: vision and challenges. In *Proceedings of the 2021 Australasian Computer Science Week Multiconference*, pages 1–10, 2021.

[13] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research advances in cloud computing*, pages 1–20. Springer, 2017.

[14] Priscilla Benedetti, Mauro Femminella, Gianluca Reali, and Kris Steenhaut. Experimental analysis of the application of serverless computing to iot platforms. *Sensors*, 21(3):928, 2021.

[15] Rajkumar Buyya and Manzur Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and computation: practice and experience*, 14(13-15):1175–1220, 2002.

[16] James Byrne, Sergej Svorobej, Konstantinos M Giannoutakis, Dimitrios Tzovaras, Peter J Byrne, Per-Olov Östberg, Anna Gourinovitch, and Theo Lynn. A review of cloud computing simulation platforms and related environments. In *International Conference on Cloud Computing and Services Science*, volume 2, pages 679–691. SCITEPRESS, 2017.

[17] Zhicheng Cai, Qianmu Li, and Xiaoping Li. Elasticsim: A toolkit for simulating workflows with cloud resource runtime auto-scaling and stochastic task execution times. *Journal of Grid Computing*, 15(2):257–272, 2017.

[18] Rodrigo N Calheiros, Marco AS Netto, César AF De Rose, and Rajkumar Buyya. Emusim: an integrated emulation and simulation environment for

modeling, evaluation, and validation of performance of cloud computing applications. *Software: Practice and Experience*, 43(5):595–612, 2013.

[19] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50, 2011.

[20] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering*, 38(4), 2015.

[21] Christopher D Carothers, David Bauer, and Shawn Pearce. Ross: A high-performance, low-memory, modular time warp system. *Journal of Parallel and Distributed Computing*, 62(11):1648–1669, 2002.

[22] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. A case for serverless machine learning. In *Workshop on Systems for ML and Open Source Software at NeurIPS*, volume 2018, pages 2–8, 2018.

[23] Henri Casanova. Simgrid: A toolkit for the simulation of application scheduling. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 430–437. IEEE, 2001.

[24] Gustavo André Setti Cassel, Vinicius Facco Rodrigues, Rodrigo da Rosa Righi, Marta Rosecler Bez, Andressa Cruz Nepomuceno, and Cristiano André da Costa. Serverless computing for internet of things: A systematic literature review. *Future Generation Computer Systems*, 128:299–316, 2022.

[25] Weiwei Chen and Ewa Deelman. Workflowsim: A toolkit for simulating scientific workflows in distributed environments. In *2012 IEEE 8th international conference on E-science*, pages 1–8. IEEE, 2012.

[26] Bin Cheng, Jonathan Fuerst, Gurkan Solmaz, and Takuya Sanada. Fog function: Serverless fog computing for data intensive iot services. In *2019 IEEE International Conference on Services Computing (SCC)*, pages 28–35. IEEE, 2019.

[27] Shilpa Choudary. Chatbot on serverless/lamba architecture. *Asian Journal of Engineering and Technology Innovation (AJETI)*, page 190, 2018.

[28] Samir Das, Richard Fujimoto, Kiran Panesar, Don Allison, and Maria Hybinette. Gtw: a time warp system for shared memory multiprocessors. In *Proceedings of Winter Simulation Conference*, pages 1332–1339. IEEE, 1994.

[29] Chavit Denninnart and Mohsen Amini Salehi. Smse: A serverless platform for multimedia cloud systems. *arXiv preprint arXiv:2201.01940*, 2022.

[30] Sheng Di and Franck Cappello. Gloudsim: Google trace based cloud simulator with virtual machines. *Software: Practice and Experience*, 45(11):1571–1590, 2015.

[31] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud computing: issues and challenges. In *2010 24th IEEE international conference on advanced information networking and applications*, pages 27–33. Ieee, 2010.

[32] Gabriele D'Angelo and Moreno Marzolla. New trends in parallel and distributed simulation: From many-cores to cloud computing. *Simulation Modelling Practice and Theory*, 49:320–335, 2014.

[33] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. A review of serverless use cases and their characteristics. *arXiv preprint arXiv:2008.11110*, 2020.

[34] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. The state of serverless applications: Collection, characterization, and community consensus. *IEEE Transactions on Software Engineering*, 48(10):4152–4166, 2021.

[35] Kalpana Ettikyala and Y Rama Devi. A study on cloud simulation tools. *International Journal of Computer Applications*, 115(14), 2015.

[36] Dror G Feitelson, Dan Tsafrir, and David Krakov. Experience with using the parallel workloads archive. *Journal of Parallel and Distributed Computing*, 74(10):2967–2982, 2014.

[37] Stephanie Forrest. Genetic algorithms. *ACM computing surveys (CSUR)*, 28(1):77–80, 1996.

[38] Richard M Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[39] Richard M Fujimoto. Research challenges in parallel and distributed simulation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 26(4):1–29, 2016.

[40] Richard M Fujimoto, Asad Waqar Malik, A Park, et al. Parallel and distributed simulation in the cloud. *SCS M&S Magazine*, 3:1–10, 2010.

[41] Phani Kishore Gadepalli, Gregor Peach, Ludmila Cherkasova, Rob Aitken, and Gabriel Parmer. Challenges and opportunities for efficient serverless computing at the edge. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 261–2615. IEEE, 2019.

[42] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K Ghosh, and Rajkumar Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017.

[43] Moin Hasan and Mohammad Anwarul Siddique. A research-oriented mathematical model for cloud simulations. In *2021 Fifth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC)*, pages 875–878. IEEE, 2021.

[44] Wilson A Higashino, Miriam AM Capretz, and Luiz F Bittencourt. Cepsim: A simulator for cloud-based complex event processing. In *2015 IEEE International Congress on Big Data*, pages 182–190. IEEE, 2015.

[45] Alexandru Iosup, Catalin Dumitrescu, Dick Epema, Hui Li, and Lex Wolters. How are real grids used? the analysis of four grid traces and its implications. In *2006 7th IEEE/ACM International Conference on Grid Computing*, pages 262–269. IEEE, 2006.

[46] Azlan Ismail. Energy-driven cloud simulation: existing surveys, simulation supports, impacts and challenges. *Cluster Computing*, 23(4):3039–3055, 2020.

[47] Heba Ismail, Nada Hussein, Rawan Elabyad, and Salma Said. A serverless academic adviser chatbot. In *The 7th Annual International Conference on Arab Women in Computing in Conjunction with the 2nd Forum of Women in Research*, pages 1–5, 2021.

[48] Vitalii Ivanov and Kari Smolander. Implementation of a devops pipeline for serverless applications. In *Product-Focused Software Process Improvement:*

19th International Conference, PROFES 2018, Wolfsburg, Germany, November 28–30, 2018, Proceedings 19, pages 48–64. Springer, 2018.

[49] Deepak Jagtap, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Optimization of parallel discrete event simulator for multi-core systems. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 520–531. IEEE, 2012.

[50] Hongseok Jeon, Chunglae Cho, Seungjae Shin, and Seunghyun Yoon. A cloudsim-extension for simulating distributed functions-as-a-service. In *2019 20th International Conference on parallel and distributed computing, applications and technologies (PDCAT)*, pages 386–391. IEEE, 2019.

[51] Devki Nandan Jha, Khaled Alwasel, Areeb Alshoshan, Xianghua Huang, Ranesh Kumar Naha, Sudheer Kumar Battula, Saurabh Garg, Deepak Puthal, Philip James, Albert Zomaya, et al. Iotsim-edge: a simulation framework for modeling the behavior of internet of things and edge computing environments. *Software: Practice and Experience*, 50(6):844–867, 2020.

[52] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[53] Soufiane Jounaid. *OpenDC Serverless: Design, Implementation and Evaluation of a FaaS Platform Simulator*. PhD thesis, Ph. D. Thesis, Vrije Universiteit Amsterdam, 2020.

[54] MA Kaleem and PM Khan. Commonly used simulation tools for cloud computing research. In *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 1104–1111. IEEE, 2015.

[55] Yogeswaranathan Kalyani and Rem Collier. A systematic survey on the role of cloud, fog, and edge computing combination in smart agriculture. *Sensors*, 21(17):5922, 2021.

[56] Pradeeban Kathiravelu and Luis Veiga. Concurrent and distributed cloudsim simulations. In *2014 IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 490–493. IEEE, 2014.

[57] Gabor Kecskemeti. Dissect-cf: a simulator to foster energy-aware scheduling in infrastructure clouds. *Simulation Modelling Practice and Theory*, 58:188–218, 2015.

[58] Gastón Keller, Michael Tighe, Hanan Lutfiyya, and Michael Bauer. Dcsim: A data centre simulation tool. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 1090–1091. IEEE, 2013.

[59] Khaled M Khalil, M Abdel-Aziz, Taymour T Nazmy, and Abdel-Badeeh M Salem. Cloud simulators–an evaluation study. *International Journal Information Models and Analyses*, 6(1), 2017.

[60] Dzmitry Kliazovich, Pascal Bouvry, and Samee Ullah Khan. Greencloud: a packet-level simulator of energy-aware cloud computing data centers. *The Journal of Supercomputing*, 62(3):1263–1283, 2012.

[61] Andreas Kohne, Marc Spohr, Lars Nagel, and Olaf Spinczyk. Federatedcloudsim: a sla-aware federated cloud simulation framework. In *Proceedings of the 2nd International Workshop on CrossCloud Systems*, pages 1–5, 2014.

[62] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 442–450. IEEE, 2018.

[63] Jyri Lehvä, Niko Mäkitalo, and Tommi Mikkonen. Case study: building a serverless messenger chatbot. In *International Conference on Web Engineering*, pages 75–86. Springer, 2017.

[64] Jason Liu, David Nicol, Brian Premore, and Anna Poplawski. Performance prediction of a parallel simulator. In *Proceedings Thirteenth Workshop on Parallel and Distributed Simulation. PADS 99.(Cat. No. PR00155)*, pages 156–164. IEEE, 1999.

[65] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE international conference on cloud engineering (IC2E)*, pages 159–169. IEEE, 2018.

[66] Márcio Moraes Lopes, Wilson A Higashino, Miriam AM Capretz, and Luiz Fernando Bittencourt. Myifogsim: A simulator for virtual machine migration in fog computing. In *Companion Proceedings of the10th International Conference on Utility and Cloud Computing*, pages 47–52, 2017.

[67] Nima Mahmoudi and Hamzeh Khazaei. Simfaas: A performance simulator for serverless computing platforms. *arXiv preprint arXiv:2102.08904*, 2021.

[68] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. Faasdom: A benchmark suite for serverless computing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*, pages 73–84, 2020.

[69] Maciej Malawski. Towards serverless execution of scientific workflows-hyperflow case study. In *Works@ Sc*, pages 25–33, 2016.

[70] Anupama Mampage, Shanika Karunasekera, and Rajkumar Buyya. A holistic view on resource management in serverless computing environments: Taxonomy and future directions. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.

[71] Zoltán Ádám Mann. Cloud simulators in the implementation and evaluation of virtual machine placement algorithms. *Software: Practice and Experience*, 48(7):1368–1389, 2018.

[72] Najme Mansouri, R Ghafari, and B Mohammad Hasani Zade. Cloud computing simulators: A comprehensive review. *Simulation Modelling Practice and Theory*, 104:102144, 2020.

[73] András Márkus. Dissect-cf-fog: A simulation environment for analysing the cloud-to-thing continuum.

[74] Andras Markus, Attila Kertesz, and Gabor Kecskemeti. Cost-aware iot extension of dissect-cf. *Future Internet*, 9(3):47, 2017.

[75] Ilias Mavridis and Helen Karatza. Orchestrated sandboxed containers, unikernels, and virtual machines for isolation-enhanced multitenant workloads and serverless computing in cloud. *Concurrency and Computation: Practice and Experience*, 35(11):e6365, 2023.

[76] Charafeddine Mechalikh, Hajer Taktak, and Faouzi Moussa. Pureedgesim: A simulation toolkit for performance evaluation of cloud, fog, and pure edge computing environments. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 700–707. IEEE, 2019.

[77] David Meisner, Junjie Wu, and Thomas F Wenisch. Bighouse: A simulation infrastructure for data center systems. In *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 35–45. IEEE, 2012.

[78] Sunil Kumar Mohanty, Gopika Premsankar, Mario Di Francesco, et al. An evaluation of open source serverless computing frameworks. *CloudCom*, 2018:115–120, 2018.

[79] Kim Long Ngo, Joydeep Mukherjee, Zhen Ming Jiang, and Marin Litoiu. Has your faas application been decommissioned yet?–a case study on the idle timeout in function as a service infrastructure. *arXiv preprint arXiv:2203.10227*, 2022.

[80] Alberto Núñez, Pablo C Cañizares, and Juan de Lara. Cloudexpert: An intelligent system for selecting cloud system simulators. *Expert Systems with Applications*, 187:115955, 2022.

[81] Alberto Núñez, Jose L Vázquez-Poletti, Agustin C Caminero, Gabriel G Castañé, Jesus Carretero, and Ignacio M Llorente. icancloud: A flexible and scalable cloud infrastructure simulator. *Journal of Grid Computing*, 10(1):185–209, 2012.

[82] Simon Ostermann, Kassian Plankensteiner, Radu Prodan, and Thomas Fahringer. Groudsim: an event-based simulation framework for computational grids and clouds. In *European Conference on Parallel Processing*, pages 305–313. Springer, 2010.

[83] Olli Paakkunainen et al. Serverless computing and faas platform as a web application backend. 2019.

[84] Annanda Rath, Bojan Spasic, Nick Boucart, and Philippe Thiran. Security pattern for cloud saas: From system and data security to privacy case study in aws and azure. *Computers*, 8(2):34, 2019.

[85] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Towards understanding heterogeneous clouds at scale: Google trace analysis. *Intel Science and Technology Center for Cloud Computing, Tech. Rep*, 84:1–12, 2012.

[86] Charles Reiss, John Wilkes, and Joseph L Hellerstein. Google cluster-usage traces: format+ schema. *Google Inc., White Paper*, 1:1–14, 2011.

[87] Paul F Roth. Discrete, continuous, and combined simulation. In *Proceedings of the 20th conference on Winter simulation*, pages 56–60, 1988.

[88] Dilshad Hassan Sallo and Gabor Kecskemeti. Parallel simulation for the event system of dissect-cf. In *The 12th Conference of PhD Students in Computer Science*, pages 58–61. University of Szeged, Institute of Informatics, 2020.

[89] Dilshad Hassan Sallo and Gabor Kecskemeti. Towards a dissect-cf extension for simulating function-as-a-service. In *Abstract book for the 16th MIKLÓS IVÁNYI INTERNATIONAL PHD and DLA SYMPOSIUM*, page 132. Pécs, Hungary : Pollack Press, 2020.

[90] Dilshad Hassan Sallo and Gabor Kecskemeti. Introducing serverless computing model based on dissect-cf simulator. In *XXIV. Spring Wind Conference: Abstract volume, Association of Hungarian PHD and DLA Students*, page 408. Miskolc University, 2021.

[91] Dilshad Hassan Sallo and Gabor Kecskemeti. A parallel event system for large-scale cloud simulations in dissect-cf. *Acta Cybernetica*, 25(2):469–484, 2021.

[92] Dilshad Hassan Sallo and Gabor Kecskemeti. Towards generating realistic trace for simulating functions-as-a-service. In *European Conference on Parallel Processing*, pages 428–439. Springer, 2022.

[93] Dilshad Hassan Sallo and Gabor Kecskemeti. Enriching computing simulators by generating realistic serverless traces. *Journal of Cloud Computing*, 12(1):1–13, 2023.

[94] Dilshad Hassan Sallo and Gabor Kecskemeti. Towards a dissect-cf extension for simulating function-as-a-service. *International Journal of Parallel, Emergent and Distributed Systems*, pages 1–13, 2023.

[95] Mathijs Jeroen Scheepers. Virtualization and containerization of application infrastructure: A comparison. In *21st twente student conference on IT*, volume 21, pages 1–7, 2014.

[96] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. Serverless computing: a survey of opportunities, challenges, and applications. *ACM Computing Surveys*, 54(11s):1–32, 2022.

[97] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 205–218, 2020.

[98] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. Numpywren: Serverless linear algebra. *arXiv preprint arXiv:1810.09679*, 2018.

[99] Mohamed Abu Sharkh, Ali Kanso, Abdallah Shami, and Peter Öhlén. Building a cloud on earth: A study of cloud computing data center simulators. *Computer Networks*, 108:78–96, 2016.

[100] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and YC Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th international middleware conference*, pages 1–13, 2016.

[101] Ahmad Siavashi and Mahmoud Momtazpour. Gpucloudsim: an extension of cloudsim for modeling and simulation of gpus in cloud data centers. *The Journal of Supercomputing*, 75(5):2535–2561, 2019.

[102] Jungmin Son, Amir Vahid Dastjerdi, Rodrigo N Calheiros, Xiaohui Ji, Young Yoon, and Rajkumar Buyya. Cloudsimsdn: Modeling and simulation of software-defined cloud data centers. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 475–484. IEEE, 2015.

[103] Cagatay Sonmez, Atay Ozgovde, and Cem Ersoy. Edgecloudsim: An environment for performance evaluation of edge computing systems. *Transactions on Emerging Telecommunications Technologies*, 29(11):e3493, 2018.

[104] Ilango Sriram. Speci, a simulation tool exploring cloud-scale data centres. In *IEEE International Conference on Cloud Computing*, pages 381–392. Springer, 2009.

[105] Kun Suo, Junggab Son, Dazhao Cheng, Wei Chen, and Sabur Baidya. Tackling cold start of serverless applications by efficient and adaptive container runtime reusing. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 433–443. IEEE, 2021.

[106] Pericherla S Suryateja. A comparative analysis of cloud simulators. *International Journal of Modern Education & Computer Science*, 8(4), 2016.

[107] Andrea Tangherloni, Simone Spolaor, Leonardo Rundo, Marco S Nobile, Paolo Cazzaniga, Giancarlo Mauri, Pietro Liò, Ivan Merelli, and Daniela Besozzi. Genhap: a novel computational method based on genetic algorithms for haplotype assembly. *BMC bioinformatics*, 20(4):1–14, 2019.

[108] Thiago Teixeira Sá, Rodrigo N Calheiros, and Danielo G Gomes. Cloudreports: An extensible simulation tool for energy-aware cloud computing environments. In *cloud computing*, pages 127–142. Springer, 2014.

[109] Wenhong Tian, Yong Zhao, Minxian Xu, Yuanliang Zhong, and Xiashuang Sun. A toolkit for modeling and simulation of real-time virtual machine allocation in a cloud data center. *IEEE Transactions on Automation Science and Engineering*, 12(1):153–161, 2013.

[110] Thurupathan Vijayakumar and Thurupathan Vijayakumar. Serverless apis. *Practical API Architecture and Development with Azure and AWS: Design and Implementation of APIs for the Cloud*, pages 133–158, 2018.

[111] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, 2018.

[112] Jinfeng Wen, Zhenpeng Chen, and Xuanzhe Liu. A literature review on serverless computing. *arXiv preprint arXiv:2206.12275*, 2022.

[113] Bhathiya Wickremasinghe, Rodrigo N Calheiros, and Rajkumar Buyya. Cloudanalyst: A cloudsim-based visual modeller for analysing cloud computing environments and applications. In *2010 24th IEEE international conference on advanced information networking and applications*, pages 446–452. IEEE, 2010.

[114] Michael Wurster, Uwe Breitenbücher, Kálmán Képes, Frank Leymann, and Vladimir Yussupov. Modeling and automated deployment of serverless applications using tosca. In *2018 IEEE 11th conference on service-oriented computing and applications (SOCA)*, pages 73–80. IEEE, 2018.

[115] Mengting Yan, Paul Castro, Perry Cheng, and Vatche Ishakian. Building a chatbot with serverless computing. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, pages 1–4, 2016.

[116] Xuezhi Zeng, Saurabh Kumar Garg, Peter Strazdins, Prem Prakash Jayaraman, Dimitrios Georgakopoulos, and Rajiv Ranjan. Iotsim: A simulator for analysing iot applications. *Journal of Systems Architecture*, 72:93–107, 2017.

[117] Miao Zhang, Yifei Zhu, Cong Zhang, and Jiangchuan Liu. Video processing with serverless computing: A measurement study. In *Proceedings of the 29th ACM workshop on network and operating systems support for digital audio and video*, pages 61–66, 2019.