DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
Yasmina BOUIZEM

Fault Tolerance in FaaS Environments

Thèse présentée et soutenue à Rennes, le 8 juin 2022
Unité de recherche : Inria Rennes - Bretagne Atlantique
Thèse N° : « si pertinent »

Rapporteurs avant soutenance :

Sébastien MONNET — Professeur, Université Savoie Mont Blanc
Thomas ROPARS — Maître de conférences, Université Grenoble Alpes

Composition du Jury :

Président :	Eric RENAULT	Professeur, ESIEE
Examinateurs :	Eric RENAULT	Professeur, ESIEE
	Sébastien MONNET	Professeur, Université Savoie Mont Blanc
	Thomas ROPARS	Maître de conférences, Université Grenoble Alpes
Dir. de thèse :	Christine MORIN	Directrice de recherche, Inria Rennes – Bretagne Atlantique
Co-enc. de thèse :	Djawida DIB	Maître de conférences, Université de Tlemcen
	Nikos PARLAVANTZAS	Maître de conférences, INSA Rennes

In memory of my mother who could not see this thesis completed

To my father for his ongoing love and support

Acknowledgement

First of all I would like to express my highly gratitude to all the jury members of the thesis for taking interest on my research work, especially, Thomas Ropars and Sebastien Monnet for accepting the task of reviewing my thesis. Thanks to Eric Renault for presiding the jury. I enjoyed every minute of the defense and this is particularly due to your organization flow of this important moment. I'm deeply grateful to my CSID committee members Guillaume Pierre and Thomas Ropars for their constructive comments, suggestions, and support throughout this work.

I would also like to express my heartfelt thanks to my supervisors Christine Morin, Djawida Dib and Nikos Parlavantzas for allowing me to do research work under their guidance, patience, support and inspiration. I am very grateful for their constant optimism and encouragement throughout this long walk with all its "ups and downs". I will never forget how they helped me, especially when i have been struggling with financial issues. I could not have made it through without their help.

Special thanks to César Viho for providing me the chance to teach some classes and believing in my abilities. Thank you so much for your support and valuable advices.

I would like to thank all the members of Myriads team at Inria for their kindness, support, and precious feedback. A word of thanks also goes out to my friends for their friendship and emotional support through all these last years.

I want also to thank all parties that granted me financial help to perform my research and live in France, especially Inria which was generous enough for me to stay longer in France. In particular, i want to thank Myriam Vinouze, a member of the HR service of Inria for her kindness and her precious help to get the funding during the pandemic.

Utmost gratitude and heartfelt appreciation for my parents for their unconditional love and never ending supply of prayers and encouragement and support, especially, my dearest darling mommy who could not see this thesis completed. Thank you, mommy, for believing in me. You inspired me to work harder, and because of that, i have achieved what seemed to be an impossible task. I also owe my special thanks to my beloved sister and brothers who always strengthened my morale by standing by me in all situations. Their belief in me has kept my spirits and motivation high during my thesis.

Table of Contents

1 Introduction **13**

1.1 Context . . . 13
1.2 Objective . . . 14
1.3 Main Contributions . . . 14
1.4 Thesis Organization . . . 15

2 Background **17**

2.1 Basic Concepts of Cloud Computing . . . 17
2.1.1 Service Types . . . 17
2.1.2 Deployment Models . . . 19
2.2 Virtualization . . . 21
2.2.1 Hardware Virtualization . . . 21
2.2.2 Operating Sysem-level Virtualization . . . 22
2.3 Containerization . . . 22
2.3.1 Container Technologies . . . 23
2.3.2 Container Orchestration . . . 24
2.4 Fault Tolerance . . . 24
2.4.1 Fault Tolerance Definition . . . 25
2.4.2 Fault Types . . . 25
2.4.3 Fault Detection . . . 26
2.4.4 Fault Tolerance Approaches . . . 26
2.5 Replication Strategies . . . 28
2.5.1 Active Replication . . . 29
2.5.2 Passive Replication . . . 29
2.6 Fault Tolerance Metrics . . . 29
2.7 Fault Tolerance in Cloud Layers . . . 30
2.8 Summary . . . 30

3 Function as a Service **33**
3.1 Function as a Service Definition . . . 33
3.1.1 Features of FaaS . . . 33
3.1.2 Execution Process of FaaS . . . 35
3.2 Benefits . . . 36
3.3 Challenges . . . 37
3.4 Use Cases . . . 41
3.5 FaaS Platforms . . . 43
3.5.1 Commercial Platforms . . . 43
3.5.2 Open Source Platforms . . . 45
3.6 Fault Tolerance in FaaS . . . 46
3.7 Summary . . . 51

4 Choosing a FaaS Framework **53**
4.1 Criteria . . . 53
4.2 Overview of Kubernetes . . . 54
4.2.1 Kubernetes Architecture . . . 54
4.2.2 Kubernetes Features Used by FaaS . . . 56
4.3 Kubernetes-native FaaS Frameworks . . . 58
4.3.1 Fission . . . 58
4.3.2 Kubeless . . . 60
4.3.3 OpenFaaS . . . 62
4.4 Performance Evaluation . . . 63
4.4.1 Environment Setup . . . 64
4.4.2 Workload Setup . . . 64
4.4.3 Metrics . . . 64
4.4.4 Results Analysis . . . 65
4.5 Overall Framework Comparison . . . 66
4.6 Summary . . . 68

5 Fault Tolerance Approaches For High Availability in FaaS **71**
5.1 Retry Mechanism in Fission . . . 71
5.2 Active-Standby in Fission . . . 72
5.2.1 Description . . . 72
5.2.2 Implementation in Fission . . . 73

5.3 Request Replication in Fission . . . 74
5.3.1 Description . . . 75
5.3.2 Implementation in Fission . . . 75
5.4 Summary . . . 76

6 Evaluation **79**
6.1 Experimental Setup . . . 79
6.1.1 Environment . . . 79
6.1.2 Applications . . . 80
6.1.3 Workload . . . 80
6.1.4 Failure Scenarios . . . 80
6.1.5 Metrics . . . 81
6.2 Experiment 1: Active-Standby with CoreDNS versus Retry . . . 82
6.2.1 Performance Results . . . 82
6.2.2 Availability Results . . . 85
6.2.3 Resource Consumption Analysis . . . 87
6.3 Experiment 2: Active-Standby with Router versus Request Replication and Retry . . . 89
6.3.1 Performance Results . . . 89
6.3.2 Availability Results . . . 93
6.3.3 Resource Consumption Analysis . . . 97
6.4 Lessons Learned . . . 99
6.5 Summary . . . 100

7 Conclusion and Perspectives **101**

Conclusion **101**
7.1 Conclusion . . . 101
7.2 Perspectives . . . 103
7.2.1 Short-Term Perspectives . . . 103
7.2.2 Mid-Term Perspectives . . . 103
7.2.3 Long-Term Perspectives . . . 104

Bibliography **105**

List of Figures

2.1 Overview of cloud services models 19
2.2 Virtualization architecture 23
2.3 Overview of fault tolerance approaches in cloud computing 28
2.4 Fault tolerance in cloud layers 31

3.1 Responsibilities of application developers and FaaS providers 34
3.2 Execution process of FaaS 36
3.3 Cold start 38
3.4 Overview of the challenges in FaaS 41

4.1 Kubernetes architecture 55
4.2 Relation between the FaaS frameworks and Kubernetes 58
4.3 Fission architecture 60
4.4 Kubeless architecture 62
4.5 OpenFaaS architecture 63
4.6 The throughput for Fission with 1, 5 and 20 function replicas. 65
4.7 The throughput for Kubeless with 1, 5 and 20 function replicas 66
4.8 The throughput for OpenFaaS with 1, 5 and 20 function replicas 66
4.9 Response time for concurrency of 200 with one function replica for three FaaS frameworks 67

5.1 Retry mechanism in Fission 72
5.2 Sequential diagram of Active-Standby mechanism using the Kubernetes *CoreDNS* 74
5.3 Overview of the Active-Standby mechanism in Fission (Implementation 2) 74
5.4 Sequential diagram of Active-Standby mechanism using a router 75
5.5 Overview of the Request Replication mechanism in Fission 76
5.6 Sequential diagram of Request Replication mechanism in Fission 76

6.1 Fibonacci application without failures 83

6.2 Guestbook application without failures 83
6.3 Fibonacci application with pod failures 84
6.4 Guestbook application with pod failures 84
6.5 Fibonacci application with node failure 85
6.6 Guestbook application with node failure 86
6.7 Recovery time in vanilla 86
6.8 Recovery time in AS 86
6.9 Resource consumption of Fibonacci in Fission vanilla and AS without and with pod and node failures 88
6.10 Resource consumption of Guestbook application in Fission vanilla and AS without and with pod and node failures 88
6.11 Response time of AS, vanilla and RR with no failure 90
6.12 Throughput of Fission vanilla, AS, and RR with pod failure 91
6.13 Response time of Fission vanilla, AS, and RR with pod failure 91
6.14 Throughput of Fission vanilla, AS, and RR with node failure 92
6.15 Response time of Fission vanilla, AS, and RR with node failure 92
6.16 Response time with 50ms of latency 93
6.17 Response time with 100ms of latency 93
6.18 Response time with 200ms of latency 94
6.19 Recovery time in AS 94
6.20 Recovery time in RR 95
6.21 CPU consumption without and with pod and node failures 98
6.22 Memory consumption without and with pod and node failures 98
6.23 CPU consumption per node in vanilla with pod failure 98
6.24 CPU consumption per node in AS with pod failure 99
6.25 CPU consumption per node in RR with pod failure 99

List of Tables

3.1 Comparison of fault-tolerant solutions in FaaS environments 50

4.1 The percentage of requests response time in milliseconds (ms) 67
4.2 Comparison of Fission, Kubeless, and OpenFaaS 68

6.1 Recovery time in Fission vanilla and AS with pod failures 87
6.2 Recovery time in Fission vanilla and AS with node failures 87
6.3 Recovery time of AS, vanilla and RR with pod failure 95
6.4 Recovery time of AS, vanilla and RR with node failure 96
6.5 Error rate in Fission vanilla, AS and RR with pod failures 96
6.6 Error rate in Fission vanilla, AS and RR with node failure 96

Chapter 1

Introduction

This chapter details the context of our study and defines the PhD thesis objective and key contributions. Finally, it gives an overview of the remainder of the document.

1.1 Context

Cloud computing is a popular paradigm that has been evolving over years. It enables developers to scale their applications that are no longer hosted on local computing resources but on shared resources obtained on demand and without provisioning a datacenter. Yet, deploying and managing cloud applications is complex and costly. Many efforts have been made to take the cloud model forward to a new model called Function as a Service (FaaS). This emerging paradigm is based on functions and has been recently gaining a lot of interest from users because of its simplicity. According to this paradigm, all the operational responsibility is transferred to the cloud providers. Thus, developers do not need to be anymore concerned about managing the underlying server infrastructure to deploy their functions and they pay only for the resources used during function execution.

FaaS is on-demand computing wherein processing is performed on remote computers; therefore, failures may occur due to communication delays or hardware failures. For example, in the event of a compute node failure, the user may experience performance instability until the failure is recovered. This means that failures should be handled in FaaS systems to guarantee high availability for functions.

In the context of FaaS, fault tolerance is managed by the FaaS provider, hence, ensuring the availability of the deployed functions in case of failures. Indeed, high availability and built-in fault tolerance are touted as main features of commercial FaaS platforms (e.g.,[10]). Most current FaaS platforms support a basic form of fault tolerance through retrying executions of idempotent functions [81], [51], [9], [73], [142]. This means that the function will return the same result when it is called multiple times with the same input data. However, while the retry mechanism allows coping with network delays, it

incurs delays in recovering from other kinds of failures such as node failures. Other fault tolerance approaches have different availability, performance, and resource consumption properties, making them appropriate for different types of faults and different scenarios. For instance, replication approaches that use function instances deployed on multiple nodes are appropriate for handling node failures. The active replication approach, in particular, favours performance and is appropriate for latency-sensitive applications. The passive replication approach favours reduced resource consumption and is appropriate for resource-constrained environments [68]. Therefore, FaaS platforms should support additional fault tolerance approaches to ensure the smooth execution of FaaS functions in the presence of different types of failures, while maintaining performance and recovering from failures without significantly impacting user experience.

1.2 Objective

Our main objective is to propose fault tolerance solutions to maintain correct execution despite the presence of failures for the deployed applications in FaaS environments. More specifically, in this thesis we assume that functions are idempotent, which is also the assumption of commercial FaaS platforms, and aim at tolerating both transient and permanent failures (such as node failures) so that the faults do not affect the execution of an application.

1.3 Main Contributions

To fulfill the aim of the study described in the previous section, this thesis makes the following key contributions:

- Comprehensive state of the art of FaaS environments emphasizing the fault tolerance techniques they offer;
- Comparison of open source FaaS frameworks in order to select one of them for the experimental evaluation of fault tolerance approaches;
- Study of the integration of two replication schemes (passive and active replication) in a FaaS environment where the first one is applied in an active-standby mode, and the second is configured in an active-active mode;

- Implementation of the replication schemes in an open-source FaaS framework, namely Fission;
- Comparative experimental evaluation according to performance, availability and resource consumption of Fission vanilla (native retry mechanism), Fission Active-Standby, and Fission Request Replication, using a stateless computational application both in normal functioning and in various failure scenarios, including function and node failures and network delays.

1.4 Thesis Organization

The remainder of this document is organized as follows:

- In ***Chapter 2*** we introduce cloud computing and its basic concepts including service models, deployment models, virtualization, and containerization. This chapter also presents fault models, fault detection techniques and different approaches to tolerate faults. In addition, various replication strategies, metrics, and fault tolerance architectures are discussed.
- In ***Chapter 3*** we present in detail the Function as a Service model, including its main features, execution process, benefits, challenges, and use cases. A brief description of FaaS platforms is also given. We also present a survey of the existing fault tolerance approaches in FaaS systems, a summary of the related works, and a comparison with our work.
- In ***Chapter 4*** we start with a discussion of a set of requirements used to select a suitable FaaS framework for our work. To select a framework that meets all the discussed requirements, we introduce and evaluate three open source FaaS frameworks that are all based on the Kubernetes container orchestration platform, which is also described. Finally, we conclude the chapter with a comparative analysis of the three frameworks.
- In ***Chapter 5*** we first present the existing fault tolerance mechanism in the Fission FaaS framework selected for our experimental evaluation of fault tolerance strategies. This framework uses a retry mechanism to provide fault tolerance. Then, we describe the two replication-based fault tolerance approaches, namely the active standby and the request replication approaches, and their implementation in Fission.

- In ***Chapter 6*** we present an experimental study aimed at evaluating the performance, availability, and resource consumption of the proposed fault tolerance approaches under different failure scenarios and compare them with the basic retry mechanism natively implemented in Fission. The interpretation of the results is also presented. We conclude this chapter with lessons learnt from the evaluation of the fault tolerance approaches.

  Finally, the last chapter concludes this thesis by summarizing our main contributions and lessons learnt from our work. We also present some future research perspectives.

Chapter 2

Background

In this chapter, we provide the necessary background to understand this manuscript. We first introduce the concept of cloud computing, its service categories, and its deployment models in Section 2.1. We also present virtualization in Section 2.2. The containerization technologies used in cloud computing environments are presented in Section 2.3. We provide an overview of different fault tolerance approaches in Section 2.4 and an overview of replication strategies in Section 2.5 since replication is the fault tolerance mechanism used in our work. Finally, fault tolerance metrics, fault tolerance in cloud layers, and the summary of this chapter are presented in Sections 2.6, 2.7, and 2.8.

2.1 Basic Concepts of Cloud Computing

According to the National Institute of Standards and Technology (NIST), cloud computing is defined as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [132]. Cloud computing is composed of different layers of services and deployment models, that we discuss in the following.

2.1.1 Service Types

The different types of cloud services offered to users can be grouped in four categories as shown in Figure 2.1: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Function as a Service (FaaS), and Software as a Service (SaaS).

1. **Infrastructure as a Service (IaaS)**

   In the IaaS model, a high level of control is given to users. It gives access to various computing infrastructure resources such as, virtual machines, storage, and networking components. IaaS users can configure and manage the systems in terms of op-

erating systems, applications and middleware, although IaaS providers still manage and control the underlying cloud infrastructure [92]. This type of service is accessible via the Internet using a pay-per-use model, where users pay only for the consumed resources based on the capacity and the time of use. Microsoft Azure, IBM cloud, and Amazon Web Services (AWS) are examples of IaaS.

2. **Platform as a Service (PaaS)**

   In the PaaS model, resources are offered by providers through a platform to the software developers. These resources include runtime environments with various programming languages (e.g., Java, PHP, Ruby) to develop applications, application frameworks to facilitate application development (e.g., Joomla, WordPress, Spring), and databases such as MongoDB, Redis to enable communication with the applications. Developers utilize these services to develop, deploy, and manage their applications without having to worry about the underlying infrastructure. It is the responsibility of the provider to provision, manage and maintain all the required hardware and software resources [92]. Just as in IaaS, PaaS is provided through a pay-per-use model. Examples of PaaS include, Google App Engine, Heroku, and CloudFoundry.

3. **Function as a Service (FaaS)**

   In the FaaS model, a platform is provided to developers to build, run, and manage serverless applications. A serverless application is decomposed into a set of independent pieces called functions that are triggered to execute a request. These functions are executed in a virtualized container and rely on other services like object storage, databases, or messaging. In contrast to other computing models, a FaaS function instance is created only when the function is called and is terminated after processing a request. Developers pay only for the resources used during the function execution, thus saving costs. FaaS adds an additional abstraction layer to the existing cloud paradigms while relieving developers from managing and configuring servers. All the server provisioning, operational responsibilities, and administration tasks are done by the cloud provider. Many big companies have adopted and offered FaaS such as Amazon's AWS Lambda, Google Cloud Functions, and Microsoft Azure Functions [103].

4. **Software as a Service (SaaS)**

   In the SaaS model, a software application is hosted and managed remotely by cloud providers, and delivered to end-users as a service. These software services are deliv-

ered on-demand through the Internet. In this type of service, the client does not need to maintain or download the software on their own machine. The software maintenance, management, and system administration are done at the provider side. SaaS provides a subscription-based pricing models for customers. This means that users need to pay monthly or annual subscription fee in order to use the software. Some examples of SaaS are: Salesforce, Slack, Zoom, and Gmail.

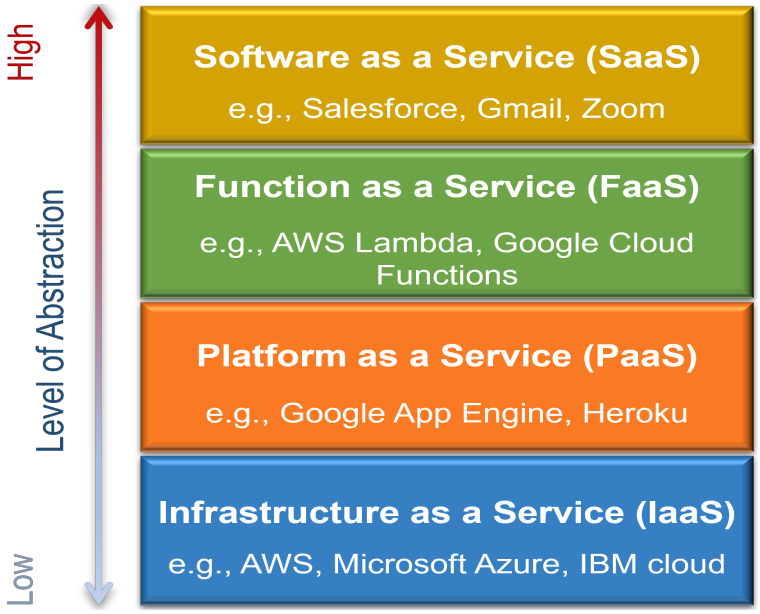


Figure 2.1 – Overview of cloud services models

2.1.2 Deployment Models

Four deployment models have been identified in cloud computing: public cloud, private cloud, hybrid cloud, and community cloud [92].

1. **Public Cloud**

   A public cloud is a computing model where all resources are available to the public remotely over the Internet. In this model, services run on servers managed by the cloud provider. The provider also has other responsibilities such as maintenance and security. In a public cloud, the pool of computing resources is shared between multiple customers with limited configuration, availability variances, and security protections. Additionally, users are typically charged based on the pay-as-you-go model. The main advantages of this model are high availability, security, and easy

scalability. The most popular public clouds include Amazon Web Services, Google AppEngine, and Microsoft Azure [92], [183].

2. **Private Cloud**

   A private cloud is a cloud service which is inaccessible to the general public. It is dedicated to a single organization (e.g., a company). In a private cloud environment, the cloud infrastructure is owned and controlled either by the organization itself, a third party, or some combination of them, and it may be implemented on or off premises. Cloud users who have access to a private cloud can utilize and store information in the private cloud from any place, similarly to public clouds. A private cloud is not delivered on a pay-as-you-go basis. The billing usually is on a subscription basis. Private clouds share many benefits of cloud computing with public clouds, such as scalability, elasticity, and flexibility. In the private cloud, the level of information security is quite high as compare to public cloud. Examples of private cloud providers include IBM, VMware, and HP [92], [183].

3. **Hybrid Cloud**

   A hybrid cloud combines a private cloud with a public cloud. It is designed to enable communication between both clouds by standardized or proprietary technology that allows data and application to work across the two deployments. For example, cloud bursting for load balancing between clouds. The hybrid setup is suitable for a business or an organization that needs to leverage the benefits of both public and private cloud environments. For example, the scalability power of the public cloud with the control and security of the private cloud [92], [183].

4. **Community Cloud**

   A community cloud infrastructure is dedicated to a specific community where only its members have access to cloud services. This model allows community members who share the same issues (e.g., mission, security requirements, policy, and compliance considerations) to solve them by integrating the services offered by different types of cloud solutions. Similarly to private clouds, a community cloud is owned and managed by one or multiple organizations in the community or by a third party provider, and can be either on or off premises. The community cloud is widely used by healthcare organizations, financial service firms, and other professional communities [92], [183].

2.2 Virtualization

The virtualization technology was first developed by IBM in the 1960s to enable concurrent access to a mainframe computer [133]. Conceptually, virtualization, as the name suggests, is a method to create a virtual version of different types of computing resources and to allow users to interact with them as if they were real resources. For example, the implementation of virtual memory allows processes to use much more memory than physically available. In the same way, virtualization can be implemented across multiple IT infrastructure layers; operating systems, storage, networking, computation and applications. Virtualization uses specialized software to simulate the hardware functionality and create virtual resources [50] [185].

Virtualization is applied within the four cloud layers. In IaaS for example, a virtual version of resources is created, such as a server or a storage device. In PaaS, the operating system environment is virtualized. In the case of SaaS, the software applications are also virtualized. In FaaS, virtualization enables developers to run function instances on top of a virtualized execution environment, which runs in an isolated manner to provide rapid scalability and better resource utilization.

There are several forms of virtualization, such as operating system virtualization, hardware virtualization, desktop virtualization, application virtualization, storage virtualization, and network virtualization.

In the next paragraphs, we present the two most popular virtualization technologies: hardware virtualization (virtual machines) and operating system-level virtualization (containers), (see Figure 2.2).

2.2.1 Hardware Virtualization

Hardware virtualization is a method which allows to run several operating systems on the same physical computer by creating virtual machines (VMs). This means, when using hardware virtualization, an abstraction layer between the software and the underlying hardware is created by a software module called a *hypervisor* or *Virtual Machine Monitor (VMM)* (see Figure 2.2a). The hypervisor sits between the host and the guest operating system to allow access to physical resources and provides resource sharing, performance isolation, and security between VMs. There are two types of hypervisors [138]:

- **Type 1 - Native or Bare-metal Hypervisor:**

Type 1 hypervisors run directly on the physical hardware of the host machine without the need for any base server operating system, meaning that the hypervisor has direct access to hardware resources (like CPU, memory, network and physical storage). Some examples of this type of hypervisor are VMware ESXi, Citrix XenServer and Microsoft Hyper-V hypervisor [93].

- **Type 2 - Hosted Hypervisor:**

  Type 2 hypervisors run as an application on a host operating system. The hypervisor abstracts guest operating systems from the host operating system. Some well-known examples of this kind of hypervisor are VMware Player and Parallels Desktop [93].

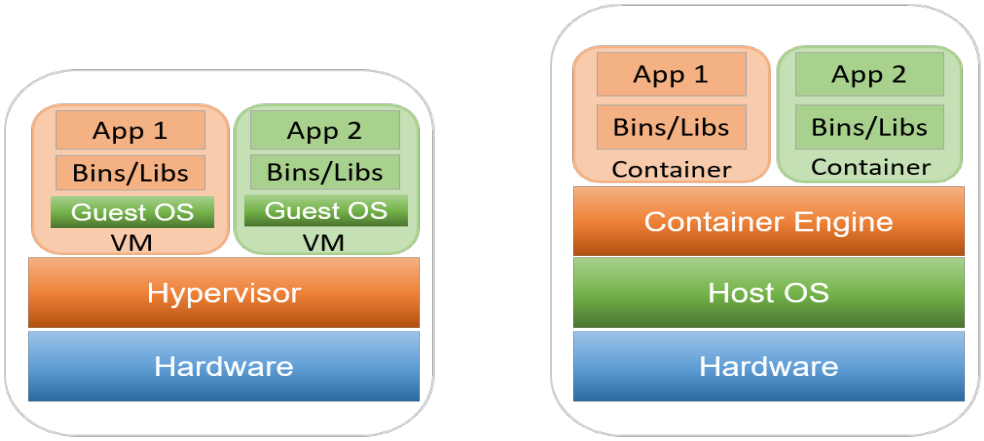
2.2.2 Operating Sysem-level Virtualization

Operating system-level virtualization, also known as container-based virtualization or containerization, is a method that allows the use of kernel features such as *cgroups*, *namespaces*, *chroot* to create distinct user space instances known as containers. The containers run on top of a shared host machine's kernel and include all the needed components to run a specific application as shown in Figure 2.2b. Unlike a virtual machine which runs a full operating system, a container does not require its own version of an operating system; instead it uses the same OS kernel as the host. This makes containers lightweight compared to virtual machines. As a result, OS-level virtualization reduces the overall CPU, memory, and networking overhead [160]. LinuX Containers (LXC) [123], Docker [59] and Rocket (Rkt) [162] are examples of OS-level virtualization.

More details about containerization are given in the next section.

2.3 Containerization

The first implementation of containerization dates back to 1979 in UNIX operating system with the development of Unix *chroot* command [106]. The *chroot* command made it possible to isolate system processes by changing the apparent root directory for a running process and its children. Then, the containerization technology evolved and other features were added to isolate file systems, users, and networking. This led to the creation of LXC [123] in 2008. In 2013, Docker [59] emerged with a full ecosystem to manage containers. These technologies gained a huge popularity and many cloud providers are using them to isolate applications. In the following subsections, we present the container



(a) Hardware Virtualization (b) OS-Level Virtualization

Figure 2.2 – Virtualization architecture

technologies and orchestrators.

2.3.1 Container Technologies

Container technologies, such as LXC and Docker, have evolved over time and are today used to pack any application as a lightweight, portable container.

1. **LXC**

   LXC is the first complete container technology providing virtualization at the operating system level and without emulating the physical hardware. It uses the features of the Linux kernel to create an isolated environment for the containers. Essentially, LXC containers are isolated by kernel *namespaces* and can access only their own set of namespaces. *Namespaces* are also used to provide the container with its own network device and virtual IP address. Another kernel mechanism called *cgroups* is implemented to limit and isolate the resources used by a group of processes.

2. **Docker**

   Docker is an open source software framework that enables packaging applications in portable containers. All the required dependencies to run the application are added in the container, such as code, runtime, system tools, and libraries. Initially, Docker was using LXC technology; starting from version 0.9, LXC was replaced with Docker's *libcontainer* library to access the Linux kernel's features [137]. Like any container technology, Docker containers run processes in isolation and control

the processes' access to CPU, memory, and disk space. Docker uses images and templates to create containers. The created container is then an instance of a specific image. Images can be created from Dockerfiles, which consist of specific instructions to build a Docker image.

2.3.2 Container Orchestration

Since the release of Docker in 2013, containers exploded in popularity and have become the choice of many developers to isolate their applications, because they are easy to deploy and thus reduce the complexity to manage applications. As the number of containerized applications has increased, managing them has become more challenging and complex. This led to the rise of container orchestration technologies. Container orchestration enables cloud and application providers to decide how to select, deploy, monitor, and dynamically control the configuration of many containerized applications in the cloud [155].

Container orchestrators make the complexity of the container life cycle easily manageable by automating different tasks such as, provisioning and deployment, scaling containers based on the load, placing containers on the desired nodes, allocating resources among containers, load balancing, and health monitoring of containers and hosts.

In the landscape of container orchestration, several platforms exist. Kubernetes [111], for example, is currently the most popular container orchestration system. It is an open source orchestrator for Docker containers, able to handle scheduling and manage workloads, depending on the demand (see Section 4.2). Besides Kubernetes, other container orchestration tools exist, such as Docker Swarm [60] and Mesosphere Marathon [130]. Docker Swarm is the native clustering system for Docker, which allows users to manage multiple containerized applications across multiple nodes. Mesosphere Marathon is also an open source container orchestration tool for Apache Mesos [17]. It provides various features to make it easier to deploy and manage containerized applications similarly to Kubernetes and Docker Swarm.

2.4 Fault Tolerance

This section defines the fault tolerance concept, presents a classification of fault types in cloud computing systems, introduces the fault detection techniques, and discusses existing fault tolerance approaches used in cloud computing.

2.4.1 Fault Tolerance Definition

Fault tolerance is the feature that enables a system to fulfill its intended function regardless of the occurrence or the presence of faults. Such a system could be a computer system, a network, or a cloud infrastructure. In other words, fault tolerance means that if a failure occurs, the system must be able to detect, identify, and recover from that failure. Fault tolerance is generally implemented by error detection and subsequent system recovery. To recover from failures, actions should be taken during normal functioning of the system. A fault-tolerant system must be capable of detecting individual hardware or software failures, power failures or other types of unexpected disasters and still fulfill its specification [63], [49].

2.4.2 Fault Types

A fault is the adjudged or hypothesized cause of the deviation of the system's state from the correct state. The deviation is called an error that may cause a system failure. A failure is recognized when the system doesn't perform according to its specification or because the specification doesn't adequately describe its function [24]. Examples of faults are: a short-circuit between two adjacent interconnects, a broken pin, or a software bug [63].

Various faults can occur in cloud computing. These faults can be classified into three main categories: transient faults, intermittent faults and permanent faults.

- **Transient Faults**

  A transient fault is a type of fault with a limited duration. It appears as a result of some temporary condition affecting the normal operation of the system. These faults include the momentary loss of network connectivity to components and services, the temporary unavailability of a service, or timeouts that arise when a service is busy [180]. However, by rolling back the system to a previous state like restarting a program or retransmitting a message, these faults are corrected [163].

- **Intermittent Faults**

  An intermittent fault is a type of fault that occurs randomly at irregular intervals then suddenly disappears, making the system switch between normal and faulty behaviors. This fault usually resembles a malfunction of a system, hardware device or component. Intermittent faults are difficult to diagnose and repair. An example

is a hard disk that has become inoperable due to temperature fluctuations, but at some point returns to normal functioning [140], [40].

- **Permanent Faults**

  A permanent fault is a type of fault that persists (i.e., the system remains in a malfunctioning state) unless fixed by some external intervention. Generally, a permanent fault occurs due to a subsystem failure, physical damage, or design errors. This type of fault can be associated with recovery events (repair/replacement), which are controllable and observable. The term failure is usually used to refer to permanent faults [140], [40].

In the remainder of this thesis, the terms fault and failure refer to the same concept, that is, the deviation from the regular behavior of the system.

2.4.3 Fault Detection

Different types of faults may occur in cloud computing, impacting the performance of the cloud system. Fault detection techniques are used to monitor and detect faults, thus helping maintain the continuity of cloud service operations. The most common detection technique used in cloud systems is heartbeat, which functions by periodically sending a signal to determine the reachability and liveness of a specific component. Normally, a heartbeat is sent periodically by a sender, which is the component requiring a health check. If a receiver, which is the component expecting and listening for heartbeats, does not receive a heartbeat after a user defined waiting period, called timeout, then the sender is considered in failure status. This method is employed for permanent hardware fault detection, where the detection is concentrated on finding the crashed nodes [118], [169]. Heartbeats are frequently used to detect the liveness of a target cloud system. For instance, Gokhroo et al. [79] applied the heartbeat mechanism to detect VM liveness. Rahman et al. [156] and Yadav et al. [197] also deployed this mechanism to detect physical host and network connection liveness. Xinming et al. [194] and Soualhia et al. [171] used the heartbeat mechanism to detect cloud service liveness.

2.4.4 Fault Tolerance Approaches

Fault tolerance in cloud computing can be achieved using a range of approaches. These approaches are mainly divided into two categories: proactive and reactive approaches [49].

These approaches have been proposed and used in cloud systems. Main fault tolerance approaches are discussed in the following and illustrated in Figure 2.3.

1. **Proactive Approaches**

   Proactive fault tolerance is the capability to predict faults and replace the suspected components by other healthy components [12]. To prevent complete outage of the system, these approaches continually monitor and predict faults so that the effects of faults are prevented before they occur [140]. Proactive fault tolerance in cloud environments can be achieved through self-healing, preemptive migration, and system rejuvenation methods.

   - **Self-Healing** is the system's ability to autonomously recover from faults by implementing specific failover procedures consisting of supervision tasks to minimize degradation effects [164]. A system can identify the faulty conditions without human intervention and can make its own decisions to restore itself to normalcy and function as it was before disruption [78]. This technique was used to automatically handle failures of application instances that run in multiple virtual machines [12].
   - **Preemptive Migration** is a method that transfers the execution of programs from one machine to another in real time. It prevents fault occurrence from impacting the performance of the system. This is achieved through monitoring and moving components away from nodes that are predicted to fail to more stable nodes [140]. This method follows the feedback-loop mechanism principle, which continuously analyzes and monitors an application [12].
   - **System Rejuvenation** is a process of performing periodic reboots for the system. After each reboot, the system is restarted with a clean state to prevent the occurrence of more severe faults, such as crashes or software malfunctioning [41].

2. **Reactive Approaches**

   Reactive fault tolerance is the capability of the system to reduce the impact of failures after they have occurred. Methods based on the reactive fault tolerance approach include checkpoint/restart, job migration, task resubmission, retry, and replication.

   - **Checkpoint/Restart** is a method that periodically records the system's state. In the event of a failure, the system is restarted from the last saved state instead

of restarting it from the beginning. This method is suitable for long-running applications [140].

- **Job Migration** is a method for migrating the job from a faulty machine to another healthy machine when a fault occurs.
- **Task Resubmission** is a technique that is mostly used in scientific workflow management systems. In this method, if a failed task is detected, it is resubmitted either to the same or to a different machine for execution [152].
- **Retry** consists of retrying a failed request multiple times with a configurable delay between attempts. The use of this method requires distributed services and applications to be idempotent. Idempotence means that calling the service once or many times with the same request parameters gives the same result. This method is effective for random and infrequent faults like temporary network glitches [44], [157].
- **Replication** is a process where some system components are duplicated on different resources, so that if a failure occurs, a replicated component called replica is available to ensure the continuous execution of tasks. This method makes the system robust, increases availability, and guarantees the execution of jobs [140].

  In our work, we focus on the replication approach that we outline in the next section.

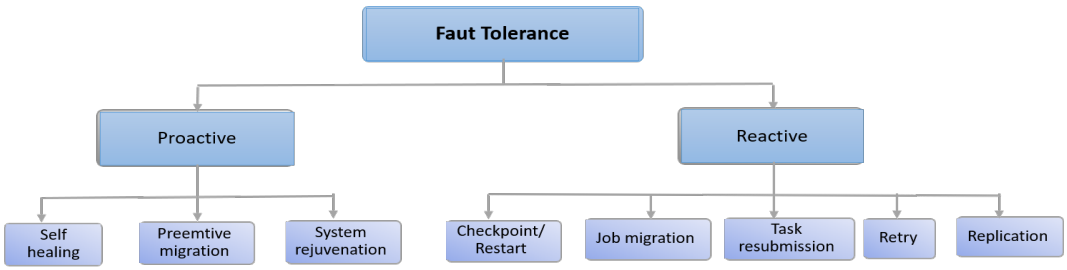


Figure 2.3 – Overview of fault tolerance approaches in cloud computing

2.5 Replication Strategies

Replication has been widely used to tolerate permanent faults. There are mainly two types of replication: active and passive, that we present in the following subsections.

2.5.1 Active Replication

In active replication [69], all replicas are concurrently invoked and each replica processes the same request at the same time. To guarantee consistency, all replicas have to execute the same requests in the same total order deterministically, so that every replica traverses the same sequence of states and produces the same output. This method makes failures fully hidden from clients, since if a replica fails, the requests are still handled by the other replicas. The major drawback of active replication is the determinism constraint, which makes this approach not suitable for a wide range of applications. Additionally, the redundancy of processing requires high resource usage.

2.5.2 Passive Replication

In passive replication [69], only one replica (the primary replica) executes the client's requests and sends update messages to the backups. If the primary replica fails, one of the backup replicas takes over the execution process. This method consumes less resources than active replication and tolerates to a larger extent non-determinism, and therefore it is more flexible. However, the passive replication incurs high reconfiguration cost in the case of primary failure (i.e., the cost of electing a new primary).

2.6 Fault Tolerance Metrics

Fault tolerance techniques in cloud computing are evaluated using various metrics. These metrics are used to identify how the system performs in normal functioning and the event of failures. The main used metrics are: throughput, response time, availability, recovery time, and associated overheads [125] [148].

- **Throughput:** this metric is used to calculate the total number of executed tasks per unit of time.
- **Response Time:** this metric measures the amount of time required by a system, an algorithm, or a component to respond to a request.
- **Availability:** it is a measure of the delivery of correct service over a specific period of time with respect to the alternation of correct and incorrect service [24].
- **Recovery Time:** is the time that it takes the system to recover from a failure.

- **Overhead:** is the amount of extra resources (CPU, memory, etc.) involved while executing a fault tolerance algorithm. Overhead is usually due to task movements, inter-process or inter-processor communication. This should be minimized so that a fault tolerance technique can be utilized efficiently.

2.7 Fault Tolerance in Cloud Layers

Cloud computing services are organized into different layers (as see in Section 2.1.1) which can be affected by various types of faults. Faults in a particular layer usually affect the services provided by the layers above it. For instance, if the operating system in the PaaS crashes, the applications running on top of the operating system at the SaaS layer will stop working. Similarly, faults in the physical hardware of the IaaS layer will conduct to a fault in the functions of the FaaS layer, which in turn will impact SaaS services. Therefore, fault tolerance in cloud computing is applied according to the layered architecture where faults in IaaS may impact its upper layers (i.e; PaaS, FaaS and SaaS) as shown in Figure 2.4. Fault tolerant techniques may be implemented in the different layers in order to guarantee the availability of the delivered services. In IaaS, for example, many proactive and reactive fault tolerance approaches are used to maintain the availability of the components. For example, some of them are preemptive migration approach handles the VM failures by replacing the faulty virtual machine with a non-faulty virtual machine, VM checkpointing periodically saves the states of a virtual machine and restarting the VM execution from the last checkpoint in case of failures. Various researchers have proposed fault tolerance solutions for cloud systems. For instance, J.Cao et al. [43] have proposed a checkpoint as a service in a VM for automatic checkpointing of long running applications.

In this thesis, we focus on the fault tolerance approaches that can be integrated in the FaaS layer. In the next chapter, we present a state-of-the-art of fault tolerance techniques applied in the FaaS layer.

2.8 Summary

Cloud computing is the delivery of multiple services over the Internet. These services are classified into four categories: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Function as a Service (FaaS), and Software as a Service (SaaS), which provide infrastructure resources, application platform, and software as services to the users. A cloud

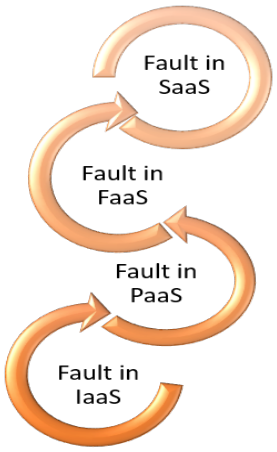


Figure 2.4 – Fault tolerance in cloud layers

can be public, private, hybrid, or a community cloud and relies on virtualization and orchestration technologies. Virtualization creates a virtual version of computing resources using software technologies such as virtual machines (VMs) and containers. Container orchestrators automate and manage the deployment of a cluster of containers. For example, Kubernetes is the widely used container orchestration tool in cloud environments.

In cloud computing, faults may occur and they can be transient, intermittent or permanent faults. Depending on the type of faults, several fault tolerance approaches can be used to deal with them such as proactive and reactive approaches. The proactive approaches avoid recovery from faults by predicting and replacing the suspected components. These approaches include self-healing, preemptive migration, and system rejuvenation. The reactive approaches reduce the effect of faults when they occur in the system. Reactive fault tolerance can be implemented using checkpoint/restart, job migration, task resubmission, retry, and replication (i.e., active and passive replication).

The next chapter presents the Function as a Service cloud service model, providing an overview of its core features, benefits, challenges and its existing fault tolerance approaches.

Chapter 3

Function as a Service

Function as a Service (FaaS) is a service model of cloud computing that enables developers to execute functions without server management. This chapter is organized as follows. In Section 3.1, we first define FaaS and its main features. Then, in Section 3.2 we present the benefits of FaaS. In Section 3.3, we discuss the challenges in FaaS and the proposed solutions to overcome each challenge. In Section 3.4, we present various use cases of FaaS. In Section 3.5, we introduce the existing FaaS platforms. In Section 3.6, we provide representative works about fault tolerance in FaaS and compare them with the work of this thesis. Finally, in Section 3.7 we summarize the chapter.

3.1 Function as a Service Definition

Function as a Service (FaaS) is a new cloud service model that abstracts away servers and infrastructure management from developers. The developers focus on writing the code of the application while the responsibility of managing and configuring the underlying resources is left to the FaaS provider. In this model, applications are in the form of functions. Functions are pieces of code that execute a specific task. In FaaS, functions are short-lived and run in a stateless container that is event-triggered. Also, these functions can automatically scale and are billed only for the resources used during their execution. This definition captures the main features of FaaS that are explained next.

Figure 3.1 shows the different responsibilities of FaaS providers and application developers.

3.1.1 Features of FaaS

The main features of the FaaS model are:

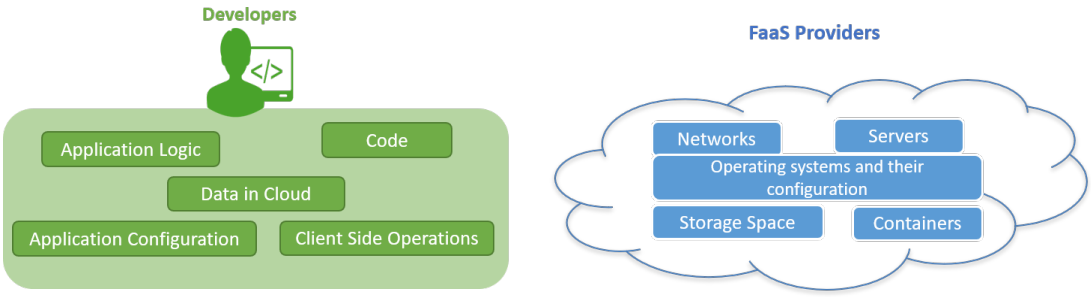


Figure 3.1 – Responsibilities of application developers and FaaS providers

Stateless

In the context of FaaS, stateless means that functions do not retain state across invocations and invocations are completely independent from each other. Stateless function are usable in different scenarios, such as isolated data processing. However, if shared state is required, then functions can make use of external services such as object stores (e.g. Amazon S3 [11]) to store shared state across requests. We call such FaaS functions that persist state in external storage services stateful functions (see [200]).

Event-trigger

Functions in FaaS are event-driven. A function listens for particular events which trigger its execution, i.e. when an event is received, a function instance is spun up and the request is processed. Many events can be used, such as HTTP requests, database events, queue services, monitoring alerts, file uploads, scheduled events.

Short-lived

Unlike typical server-based applications that are always up and running, waiting for requests, FaaS applications are designed to be short-lived processes and executed on demand. Therefore, functions can only run for a few milliseconds to a few minutes when called and then shut down until the next invocation happens. Some FaaS providers set a maximum lifetime for the function before they kill it. For example, Amazon Lambda functions can run for up to 15 minutes [115], Microsoft Azure Functions for up to 10 minutes [31], and Google Cloud Functions have a timeout of 9 minutes [77].

Pay as you go

As opposed to Infrastructure as a Service deployments, where developers pay for the reserved resources even when the application is not running, FaaS uses a different pricing approach based on pay-per-use billing model. The users only pay for resources that had been used during the execution of their function and not for idle time. As the execution time of functions is short, users are charged in fine-grained time units (such as hundreds of milliseconds).

3.1.2 Execution Process of FaaS

When using the FaaS model, the developer has to focus on writing code and deploying it in a managed environment. A typical execution process of the FaaS function is depicted in Figure 3.2 and described as follows:

- **Preparing function code:** the developer writes the application code as a set of functions and gives it to the FaaS provider. One function only performs one specific task. Functions are written in any programming language, such as Java, Python, and Go.
- **Setting up an event:** the developer configures an event to trigger the execution of her function. There are a wide variety of events. The HTTP event is the commonly used one.
- **Triggering the execution:** the user sends a request and waits for the response.
- **Executing the function:** the function execution is the responsibility of the FaaS provider (it creates a container, loads user code, prepares a runtime). When the event is received, the provider either uses an existing function instance, or, if there is no instance currently running, it starts a new one to execute the function. After the execution is completed, the function instance is killed and the resources are released.
- **Sending the response to the user:** the user receives the result of the executed function.

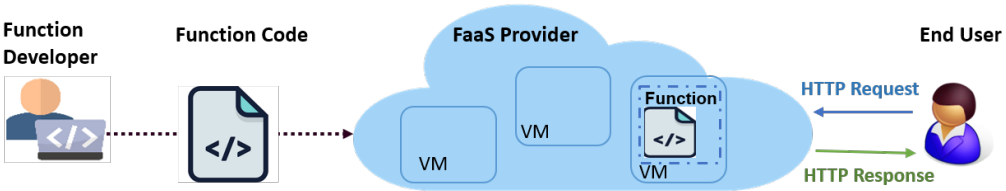


Figure 3.2 – Execution process of FaaS

3.2 Benefits

The FaaS service model offers numerous benefits to its users. These benefits include simplified development, ease of deployment and updates, cost effectiveness, and scalability.

Simplified development

FaaS offers a high level of abstraction to the application developers, which means the developers do not need to concern themselves with the low-level details related to infrastructure management including network, servers, operating systems, or storage. The FaaS provider takes care of most operational concerns of managing servers and other resources, such as provisioning, monitoring, maintenance, scalability, and fault tolerance. This makes developers focus more on the development of the functions defining the application, thus increasing their productivity and reducing development time.

Ease of deployment and updates

FaaS facilitates application deployment as the provider manages all deployment-related dependencies. Additionally, the code can be quickly deployed, since the application is composed of independent functions. If any update is required or a new feature must be added to the application, it is not necessary to make changes to the whole application, only the function implementing this feature has to be updated, which saves time.

Cost effectiveness

FaaS employs a flexible pricing model where customers are charged only for the resources consumed during the execution of their functions. There is no cost while their functions are in an idle state. The cloud providers reduce the operational costs required to execute functions. This saves user operating costs for tasks such as installation, maintenance, patching, and support.

Auto-scaling

Developers benefit from the FaaS model by not worrying about the application scalability, because the cloud provider takes over this responsibility and handles all of the scaling on demand. Scaling in FaaS happens at the function level. If there are several simultaneous requests to a function, then the provider will start up new function instances to handle the load. The function automatically scales down, when demand drops. This dynamic nature of auto-scaling allows more workloads to run and leads to an efficient use of resources.

3.3 Challenges

Besides all the benefits of abstracting away from the underlying "server" infrastructure that the FaaS offering brings, there are many hidden challenges. Let's take a look at the following challenges with FaaS and how they can be overcome.

Vendor Lock-in

One major concern that has appeared with FaaS is vendor lock-in. Lock-in tends to involve mainly the vendor-specific features and services, like, databases and storage. If a user decides to switch vendors, she will need to change the code and update her deployment scripts to use the Command Line Interface (CLI) of the new provider. This is because each provider has his own CLI to package and publish functions. Additionally, the technical differences and their associated pricing that vary from vendor to vendor make switching difficult and costly. For instance, AWS Lambda functions cannot run on Google Cloud Function because Amazon's S3 and Google's Cloud Storage do not have the same back-end mechanisms and APIs.

A few research studies have tackled this issue. In [199] the authors focused on analyzing and classifying the different lock-in types encountered during the migration of FaaS-based applications across commercial providers. A method for assessing the portability of FaaS-based applications is presented in [198]. It checks the portability for a chosen cloud provider and analyses the included source code artifacts to identify possible migration issues.

Cold Start Delay

FaaS functions face a significantly increased start-up latency, also known as cold start. A FaaS function is encapsulated in containers. When a user invokes a function that has not been executed for a long time, this requires the creation of a new container, and the initialization of the execution environment. As a result, the request will take a long time to be executed. This phenomenon also happens when scaling up function instances to handle traffic. Each function instance creation incurs a delay until the instance can process requests. This can lead to performance issues and have a significant impact on the user's response time.

The concept of cold start is depicted in Figure 3.3.

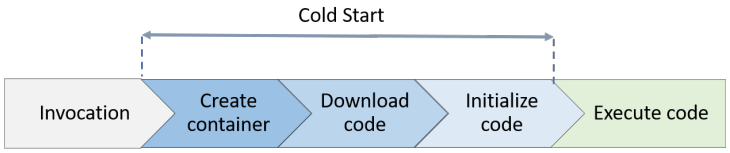


Figure 3.3 – Cold start

In the last few years, cold-start delays have became a key issue and one of the most measured metric to evaluate FaaS providers. A series of research works have investigated the cold start latencies of major FaaS platforms (i.e., AWS, Azure and Google platforms). For example, McGrath et al. [131] have studied the cold start times for popular FaaS vendors while developing their serverless computing platform model. Mikhail [54] compared the three cloud providers in terms of cold start. Other studies have analyzed the factors that contribute to cold start delays. Wang et al. [189], Jackson et al. [98] and Manner et al. [128] found that the cold-start execution times can be impacted by many factors such as, language runtime, the size of the function, and the sandbox technology used for function instances.

Numerous research studies have proposed solutions to optimize the cold start problem. Some works [62], [184], [177], [42], [72] use snapshotting techniques to reduce the start-up latencies of functions. This consists of storing the image of a fully-booted function on disk in order to enable a faster invocation instead of booting a function from scratch. Another approach has been proposed, based on using the container pool strategy [135], [196], [122] that keeps a pool of containers that are already initialized with the software environment. If a request comes in, the function code is loaded directly into these containers, without having to deploy a new container.

Limited Execution Time

FaaS functions are designed to be short-lived processes and are limited in their execution time. This limits the kinds of applications that can run in FaaS environments because there are some scenarios where long-running logic is necessary, such as machine learning algorithms.

To extend the use of FaaS for long-running applications, it is possible to decompose long-running applications into smaller functions that individually take less time to run. For this, FaaS providers come with orchestration frameworks to organize chained workflows of functions and to facilitate communication between them when they have data dependencies. AWS Step Functions [27], Azure Durable Functions [32] and IBM Composer [95] are examples of these orchestration tools. Baldini et al. [35] presented also a solution for sequential compositions on top of Apache OpenWhisk.

Debugging, Logging and Monitoring

FaaS has changed the way applications are designed and built, and that changed the debugging process as well. Debugging a FaaS application becomes much more challenging and complex compared to monolithic front-ends, because it involves many functions. When multiple functions or services are integrated, debugging becomes difficult in a local environment since most environmental dependencies are only available at runtime [119]. Additionally, remote debugging is not an option, as FaaS platforms do not allow access to the server and operating system levels. In this context, the most used method to debug errors is logging, but since FaaS functions are short-lived, the state of the function container cannot be preserved and subsequent calls to the same function will not be able to access the data from the previous runs. In this case it is hard to collect the required information for debugging.

This problem led commercial vendors to build tools that address the challenges of FaaS development. For example, Epsagon [65] and Thundra [179] are used to monitor, debug, and help troubleshoot problems. Manner et al. [129] also addressed this issue and proposed an approach that combines monitoring and logging. It is based on alerts and appropriate log messages to provide semi-automatic troubleshooting for cloud functions.

Security Concerns

FaaS applications raise certain security concerns. As applications are integrated with database services, back-end cloud services, and connected through networks and events, they are susceptible to a variety of security vulnerabilities. For example, event-data injection [66] happens when an untrusted and unauthorized data entry is passed to an application and executed without validation. This type of injection can attack the source code and other secrets of functions stored in the container. Insecure deployment configuration and broken access control allow an attacker to launch Denial of Service or Denial-of-Wallet [61] attacks. These attacks exploit functions with long timeouts to cause increased cost or get unauthorized access to function resources. Another regular vulnerability in libraries and platform code, referred to as poisoning the well attacks [190], is where a malicious code is inserted into a library that many applications depend on.

Several commercial security solutions exist to solve some of these problems [170, 21, 176, 175]. For example, Snyk [170] is one of the popular solutions used to secure the FaaS applications by finding, fixing, and tracking potential vulnerabilities in open-source dependencies. Aqua [21] is a tool that continually scans container images and function's code to ensure that developers don't introduce vulnerabilities in a library, embedded secrets (keys and tokens), or permissions. Sysdig [176] is a platform that provides runtime threats detection and response to secure function containers. SteamAlert [175] is a real-time data analysis framework built upon Amazon Lambda and used to scan log data for incident detection and response.

Researchers has also tackled the security issues and proposed various solutions [165, 100, 80, 58]. In [165], the authors developed a workflow-sensitive authorization model for FaaS applications. It proactively checks external requests' permissions for all functions of a workflow. This allows the application to reject illegal requests as soon as possible, thereby reducing the attack surface of the application. Jegan et al. [100] present SecLambda, an extensible security framework for performing sophisticated security tasks to protect a FaaS application and achieve control flow integrity, credential protection, and DoS rate limiting. In [80], the authors propose a security and privacy-based lightweight framework called iFaaSBus, that processes the data coming from IoT devices. Datta et al. [58] propose valve, a transparent solution that enables fine-grained control of information flows of functions and provides security guarantees.

Figure 3.4 summarizes the mentioned FaaS issues, such as cold start and vendor lock-in. We discuss the challenge of fault tolerance and potential ways to address it in Section 3.6.

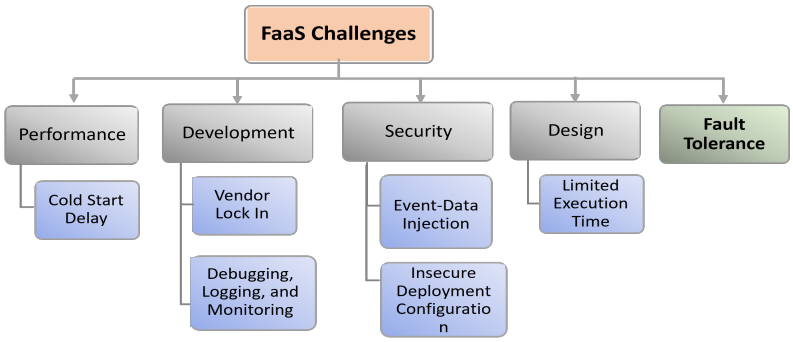


Figure 3.4 – Overview of the challenges in FaaS

3.4 Use Cases

The FaaS model is typically used in various use cases. These use cases include different classes of applications. This section attempts to cover some of these applications, namely, video processing, scientific computing, internet of things, and machine learning.

Video Processing

Just after the rise of FaaS, experts and researchers started giving a lot of interest in how to use FaaS for video processing. For instance, Ao et al. [13] proposed Sprocket, a framework that allows developers to orchestrate functions in video processing and achieve high parallelism, low latency, and low cost. A measurement study is presented in [201] to analyze the impact of function resource configurations and function implementation schemes on the performance of video processing applications.

Scientific Computing

A lot of efforts have been done to identify the possible ways to deploy FaaS for scientific applications. Existing studies, such as [47] demonstrated the feasibility of using the FaaS model for scientific and high-performance computing by presenting various prototypes and their respective measurements. In [172], the authors proposed a high-performance FaaS platform that enables the execution of scientific applications. Shankar et al. [167] designed a system for executing linear algebra programs on FaaS architectures. A prototype for

executing the scientific workflows in FaaS environments has been developed and evaluated by Malawski et al. [127].

Internet of Things (IoT) and Edge Applications

The FaaS model has been exploited in many IoT applications. Herrera-Quintero et al. [88] used microservices and FaaS functions in their smart IoT approach to implement an intelligent transportation system. A fog function programming model for data-intensive IoT services is proposed in [48] to improve efficiency and flexibility of existing frameworks. In another work, Persson et al. [150] proposed Kappa, a framework that uses the FaaS model for IoT to push computation to the very edge of the network.

Many FaaS platforms have been developed to deploy functions at the edge [151, 158, 187]. For example, Pfandzelter et al. [151] proposed a lightweight FaaS platform adapted to edge computing for faster execution. Rausch et al. [158] proposed and implemented a platform to develop and operate edge AI functions in edge cloud systems. Wang et al. [187] developed a platform that applies model-driven resource management algorithms for running latency-sensitive applications on edge resources. Amazon has also joined this field and allowed application developers to place Lambda functions in edge nodes with Lambda@Edge [116].

Machine Learning (ML)

The benefits of FaaS have triggered a growing interest on how to use it in machine learning. Recently, research from both academia and industrial communities have focused their attentions toward the FaaS model for those applications. For instance, Xu et al. [195] found that deep neural networks could benefit from the FaaS paradigm, since users are allowed to decompose complex model training into multiple functions without managing the server. A novel FaaS architecture for the deployment of neural networks is discussed in [182]. Furthermore, various frameworks have been proposed to deploy machine learning in FaaS environments. For example, SIREN [188] is an asynchronous distributed machine learning framework based on FaaS. AWS [7] also provided one example of ML training in AWS Lambda using SageMaker [6] and AutoGluon [23]. SageMaker [6], is a fully managed service that provides the necessary tools to create, train, and deploy ML models. AutoGluon [23], is an open-source library that automates the ML tasks.

3.5 FaaS Platforms

FaaS platforms are systems that implement the FaaS service model. At the beginning of 2014, only the major cloud providers like Amazon offered such a service, but since 2016, many open-source frameworks have been created. In this section, we present an overview of the most well-known platforms.

3.5.1 Commercial Platforms

Many cloud providers have adopted the FaaS service model and offered their own solutions, namely, AWS Lambda [10], Google Cloud Functions [52], Microsoft Azure Functions [33], IBM Cloud Functions [94], Oracle Functions [146], and Alibaba Functions [3].

Below is the description of three widely used platforms provided by Amazon, Microsoft, and Google.

1. **AWS Lambda**

   AWS Lambda [10] was the first public FaaS platform, released in 2014 by Amazon. Lambda functions are deployed via isolated Linux containers and executed on a multi-tenant cluster of machines managed by AWS. The platform provides support for many languages and runtimes, including Java, C#, Python and Node.js. Each lambda function can be deployed in a particular region and invoked using variety of event sources (e.g,. DynamoDB [4], S3 object storage [5]). When the number of events increases, Lambda scales automatically to process each trigger individually. Functions are charged based on the amount of memory allocated and for every 100 milliseconds of usage. The usage is measured in GB-second (Gigabyte-second), which is the number of seconds multiplied by the number of GB of memory consumed.

2. **Microsoft Azure Functions**

   Azure Functions [33] is a public FaaS platform, launched by Microsoft in 2016. It was designed to extend the existing Azure application platform with capabilities to load user function code in containers. The loaded functions can be written in C#, F#, Node.js, Python, PHP, batch. The NuGet open source package manager and the Node Package Manager for JavaScript are also supported, allowing developers to use popular libraries. Similarly to AWS, Azure Functions can be triggered by Azure services, such as Azure Blob storage [28], the Azure SQL database [30], and the Azure Queue storage [29]. Azure Functions offer automatic scaling. When the

traffic increases, the function is automatically scaled out and when it is reduced, the number of function instances is scaled down. Azure has identical free monthly quotas and charges customers in the same way as AWS. Besides, Microsoft does not charge for allocated memory but for used memory.

3. **Google Cloud Functions**

   Google released its own FaaS platform in 2016, called Google Cloud Functions [52]. It offers a very similar set of features compared to AWS and Azure. Each Function runs in gVisor [87] container sandboxes. GVisor is a sandboxed container runtime that provides secure isolation for containers. Google Cloud Functions supports functions developed in JavaScript, Python 3, Go, and Java runtimes. Functions can be triggered by several events, such as HTTP, Cloud Pub/Sub, and other sources like Firebase [71], and in response to Google Cloud Logging events. Google Cloud Functions implement automatic scaling, which allows to scale up as many instances as needed based on load, and scale down to zero when there is no traffic. Google Cloud Functions offers 1 million extra free requests per month compared to AWS and Azure, but it is considered more expensive for high-volume.

There has been an increasing interest of the academic community in comparing different FaaS platform solutions. Lee et al. [117] investigated the performance of Amazon Lambda, Microsoft Azure Functions, Google Functions, and IBM OpenWhisk regarding the CPU performance, network bandwidth, and file I/O throughput. They also presented a comparison of their features. They found that AWS Lambda outperforms other public cloud solutions. Lynn et al. [124] presented a detailed high-level technical comparison of seven enterprise computing platforms, including AWS Lambda, Microsoft Azure Functions, and Google Cloud Functions. In another study by Figiela et al. [70], a framework for performance evaluation of cloud functions has been developed and applied to AWS Lambda, Azure Functions, and Google Cloud Functions. Similiary, Timon et al. [34] developed and used a microbenchmark to evaluate the performance and cost model of popular public FaaS solutions.

The increasing interest to understand the performance behavior of the different commercial FaaS platforms has resulted in developing benchmark suites and tools. For example, a FaaS benchmarking framework was developed by Pellegrini et al. [149] to evaluate the performance of FaaS platforms. It allows to collect different metrics including execution time, routing time, throughput, and response time. It offers micro-benchmarks and application-based functions. Martin et al. [84] also designed BeFaaS, an extensible

benchmark framework for FaaS environments. It allows to evaluate arbitrarily complex deployments in which functions are deployed on multiple FaaS platforms and distributed across cloud, edge, and fog nodes.

Kim et al. [107] proposed FunctionBench, a benchmark suite that allows users to evaluate various cloud FaaS functions. In addition to micro-benchmarks, FunctionBench offers realistic application-based functions. The source code of functions is customized for AWS Lambda, Google Cloud Functions, and Azure Functions. Another benchmark suites for FaaS called FaaSdom [126] and SeBS [55] have been proposed to assess the performance of AWS lambda, Google Cloud Functions, and Azure Functions.

3.5.2 Open Source Platforms

Unlike existing commercial FaaS platforms, open-source FaaS frameworks allow developers to deploy and manage functions on self-hosted clouds with no vendor lock-in by using technologies, such as Docker Swarm and Kubernetes. For example, the most well-known solutions are Kubeless [110], Fission [73], OpenFaaS [143], Knative [108], and OpenWhisk [19]. In this section we describe Knative [108], and OpenWhisk [19]. We discuss Kubeless, Fission, and OpenFaaS in more detail in Chapter 4.

1. **Knative**

   Knative [108] is an open-source FaaS framework, developed by Google in 2018. It runs on top of Kubernetes, which is an orchestration tool for container management. The platform consists of three main components: Build, Eventing, and Serving. Build is a deployment tool that helps in building container images from source code, and deploying them via Kubernetes. Eventing is a component that provides an event driven framework to manage and deliver events between containers. Serving consists of a request-driven model for serving workloads. This component enables automatic scaling of containers based on the received requests. Knative also uses the Istio [97] service mesh networking framework for efficient and flexible traffic routing. Knative functions can be written in many programming languages, such as, C#, Go, Java, Node.js, PHP, Python, Ruby, and Rust. Multiple eventing sources can be configured to invoke functions, such as GitHub, Apache CouchDB, Cron, Kafka, Camel, and SQS.

2. **Apache OpenWhisk**

   Apache OpenWhisk [19] is an open-source FaaS platform, that is a part of Apache

Software Foundation [178]. It was developed by a research group at IBM and was released in 2016. It is built on top of Kubernetes. In the context of OpenWhisk, a function is named action, and a trigger represents an event that can occur from different sources. A rule associates a trigger to an action. OpenWhisk includes many components: a controller component, an invoker component, an Nginx webserver, an Apache Kafka component, and a CouchDB database for storing the configuration data. The controller component manages the different entities, handles triggers and routes requests to the invoker. The invoker launches isolated containers to execute the actions. The Nginx webserver acts as the reverse proxy for the system. The Kafka component controls the connection between the controller and invokers. OpenWhisk supports the execution of actions written in Node.js, Java, Python, Go, Swift, PHP and Ruby. Actions can be triggered by different event sources, including databases, timers, message queues, or websites like Slack or GitHub.

After the FaaS frameworks were released, they received attention from the scientific community, resulting in several comparative studies. For example, Mohanty et al. [136] presented a feature comparison and performance evaluation of Kubeless, OpenFaaS, Fission, and Apache OpenWhisk. They also provided an insight into the design choices of each framework. In [121] the authors focused on the architectural blocks that impact the performance of the open-source FaaS frameworks. Li et al. [120] provided a comprehensive analysis of the performance and the impact of the supported scaling algorithms of open-source FaaS frameworks.

3.6 Fault Tolerance in FaaS

We identified in Section 3.3 the key challenges of FaaS and the various solutions proposed to overcome each challenge. In this section, we consider the fault tolerance challenge in the context of FaaS and the used mechanisms to recover from failures. In terms of related work, and considering how recently FaaS was introduced, existing literature on the subject of fault tolerance is relatively limited and most FaaS platforms do not pay enough attention to fault tolerance, which is crucial especially for real time applications like safety-critical IoT applications [147].

The commonly used mechanism to handle function failures is the retry mechanism. All major commercial platforms, such as AWS Lambda [9, 25], Google Cloud Functions [81] and Microsoft Azure Functions [51] provide automatic retry functionality to handle fail-

ures and timeouts. For instance, AWS Lambda retries asynchronous invocations up to two times with a delay between the retries. Some open-source FaaS platforms also support the retry mechanism, including Fission and OpenFaaS, which retries asynchronous invocations with an exponential back-off [75, 142].

Fault tolerance in FaaS systems can also be supported through using further services provided by cloud platforms. For instance, using Azure load-balancing and event ingestion services, developers can deploy functions in different regions in an active-active or active-passive pattern, which provides protection against disaster scenarios [26]. Using AWS services, developers can mitigate against a regional disaster by replicating applications across regions in an active-active and active-passive setup [1, 2].

Using FaaS orchestration services (such as Google Workflows [82], AWS Step Functions [27], or Azure Durable Functions [32]), developers can define workflows that coordinate functions, automatically retry failed or timed-out invocations, and run custom code to handle different types of errors. For instance, using AWS Step Functions, developers can resume failed workflows from the state at which they failed [161]. Azure Durable Functions also handles failures by periodically checkpointing execution history into storage tables. The used storage tables use eventual consistency patterns, which guarantee that no data is lost in the event of a crash or loss of connection during a checkpoint [91].

Fault tolerance is also provided by open-source orchestration frameworks, such as Apache OpenWhisk Composer [20] or Faas-flow for OpenFaaS [67] that create a workflow which can re-execute a function if a failure occurs.

Recent research works are investigating fault tolerance for stateful FaaS applications, composed of multiple functions and interacting with storage services. Sreekanti et al. [173] propose inserting a layer between commodity FaaS platforms and key-value stores to ensure atomic visibility of storage updates. The proposed system assures fault tolerance by enforcing the read atomic consistency guarantee when retrying function executions. Zhang et al. [200] describes a library and runtime for building transactional, fault-tolerant workflows on existing FaaS platforms. The system supports transactions within and across functions through applying a log-based fault tolerance approach. Jia et al. [101] provide also Boki, a FaaS runtime that offers an API for stateful applications. The API enables the applications to manage their state and uses a log-based mechanism to achieve fault tolerance.

Wu et al. [192] present HydroCache, a distributed cache layer for FaaS systems, which provides transactional causal consistency for stateful functions. The system relies on Anna

storage [191], a key-value state backend that supports fault tolerance. To ensure fault tolerance, transactions are retried with the same key version in case of storage node failure or network delay. Additionally, the node failure is detected by the heartbeat mechanism and unfinished functions of the failed node are re-scheduled on another node.

Another recent work presented in [89], Heus et al. introduce and implement a programming model for transactions across stateful FaaS functions. The solution is built on top of Apache Flink StateFun [15], an open-source platform for stateful FaaS functions to manage state and provide fault tolerance. The system deals with failures via checkpointing/snapshots to achieve exactly-once-processing guarantees.

Zhang et al. [202] propose Kappa, a programming framework that facilitates FaaS functions development and periodically checkpoints function results in order to enable failure recovery. In [45], Carver et al. present a framework called Wukong for building parallel FaaS applications atop AWS Lambda. In case of failure, the automatic retry mechanism of AWS Lambda is used to re-execute the failed function. Karhula et al. [105] use Docker and CRIU (Checkpoint/Restore In Userspace) for checkpointing and resuming long-running functions that run on IoT devices as well as for migrating these functions to different IoT devices. These mechanisms could be used as building blocks for a fault-tolerant FaaS system, but this work does not provide a complete, practical implementation of such a system.

In Table 3.1, we provide a summary of the aforementioned related work and our solution, which will be described in detail in Chapter 5. We compare them based on a set of features, namely, the environment type, the targeted applications, the fault type, the fault tolerance mechanism, and the additional services requirement. The proposed solutions rely on reactive-based fault tolerance approaches including retry, checkpointing, and migration to tolerate transient and permanent faults. Among the discussed studies, some of them focused on state consistency of stateful applications in the face of transient and permanent faults [173], [192]. Their solutions mitigate the problem of duplicated data if a function is retried. Other solutions [89], [202] apply the checkpoint mechanism to deal with transient and permanent failures for stateful applications. All the solutions explicitly targeting stateful applications rely on programming models and that are not included in the core FaaS model, which only supports dispatching functions in response to events. In the table, a solution is considered to require additional services when it relies on services beyond those provided by the core FaaS model; for example, when it relies on load balancing, event injection, and workflow orchestration services [27, 32, 82], or

specialized, state-aware programming models [173, 200, 101, 192, 89, 202, 45].

The solution proposed in this thesis relies on having the replicas at different nodes in active-active and active-passive setup to achieve high availability for FaaS functions in case of transient and permanent faults. Our solution differs from the other solutions in one major way; that is, their design involves additional services working in conjunction, while our solution does not use any additional services. Such solutions impose complexity on FaaS developers.

In short, the table shows that there is no existing solution that handles both transient and permanent faults and supports fault tolerance mechanisms beyond retry without imposing the use of services outside those of the core FaaS model. Our solution is the only one that has these characteristics, integrating two fault tolerance mechanisms (active and passive replication) within the FaaS runtime without imposing additional complexity to developers.

Work	Environnent Type	Target Explicitly Stateful Applications	Fault Type	Mechanism	Additional Services Required
• AWS Lambda [9] [25] [1, 2] • Google Cloud Functions [81] • Azure Functions [51] [26]	Commercial FaaS Platform	No	Transient and Permanent	Retry, Replication (Active and Passive)	Yes
• Fission [75] • OpenFaaS [142]	Open source Framework	No	Transient	Retry	No
• AWS Step Functions [27] • Google Workflows [82] • Azure Durable Functions [32]	Commercial Function Orchestrator	Yes	Transient	Retry, Checkpointing	Yes
• Apache OpenWhisk Composer [20] • Faas Flow [67]	Open source Function Orchestrator	Yes	Transient	Retry	Yes
• Zhang et al. [200] • Jia et al. [101]	FaaS Runtime	Yes	Transient	Log-based	Yes
• Sreekanti et al. [173] • Wu et al. [192]	FaaS Runtime	Yes	Transient and Permanent	Retry	Yes
• Heus et al. [89] • Zhang et al. [202]	Programming Framework	Yes	Transient and Permanent	Retry, Checkpoining	Yes
• Carver et al. [45]	Programming Framework	Yes	Transient	Retry	Yes
• Karhula et al. [105]	Mechanism for checkpointing and migrating IoT edge functions	Yes	Permanent	Checkpointing and Migration	No
Our Solution	Open source Framework (Fission)	No	Transient and Permanent	Active and Passive Replication	No

Table 3.1 – Comparison of fault-tolerant solutions in FaaS environments

3.7 Summary

FaaS is a new paradigm of cloud computing that enables developers to execute their functions in response to events. With FaaS, the developers do not have to worry about the management of infrastructure and their functions are scaled automatically as needed. FaaS also offers a pay-per-use billing model where customers pay only for the resources used during the execution of the function. Despite all these benefits, FaaS is still immature and suffers from multiple limitations. For example, FaaS providers have a vendor-lock in problem. There is also a lack of tools that help debugging and managing FaaS applications. Another major issue in FaaS environments is the cold start of functions. FaaS offers support for a wide variety of use cases, including video processing, scientific computing and IoT applications. Additionally, FaaS is offered by many big companies like Amazon, Google and Microsoft. Some open source frameworks also have been released to use the FaaS model. Most of the existing FaaS platforms have a retry mechanism to tolerate faults. We analysed existing studies that discuss fault tolerance in FaaS environment and identified their limitations. We found that there is no system that sufficiently supports fault tolerance and does not require complex configurations for FaaS users. In the next chapter, we evaluate three Kubernetes-native FaaS frameworks and we select one of them for our work.

Chapter 4

Choosing a FaaS Framework

This chapter presents our method to select a suitable FaaS framework in order to integrate FaaS fault tolerance mechanisms. We first provide a list of criteria in Section 4.1 that we have used to make a first selection among available open source FaaS frameworks. The selected frameworks offer the feature of no vendor lock-in and are all based on Kubernetes [111]. At the time of doing this work in 2018, the selected frameworks were gaining large-scale adoption within the community. We evaluated their performance and decided which framework is best suited for our work.

This chapter is organized as follows. Section 4.1 provides the list of selection criteria. Section 4.2 provides an overview of Kubernetes. Section 4.3 presents a structured overview of the three frameworks. Section 4.4 presents their evaluation. Section 4.5 elaborates on selecting the most appropriate framework for our use case. Finally, Section 4.6 concludes this chapter.

4.1 Criteria

In order to choose the framework to use in our work, we started by making a first selection of frameworks among available ones. Afterwards, we evaluated them. The first selection was based on the following criteria.

- **Open source**: We only considered open-source frameworks that do not incur vendor lock-in. This allows us to make modifications to source code and add new features. In 2018, several open source FaaS frameworks have emerged based on open-source containerization technologies, such as Docker [59] and Kubernetes.
- **Popularity and Community Size**: We considered the frameworks that have an active development community that contributes to testing, developing, and releasing new versions. As all open-source FaaS frameworks are hosted on GitHub, we can easily check the number of GitHub stars of each framework to get an idea of its

popularity.

- **Documentation and Support**: We considered frameworks having an official technical documentation to guide in the use of the framework. If a technical documentation exists, it has to be updated and easily accessible to provide detailed information on the framework's architecture. The framework should also have community support on forums or chat channels, such as Slack channels, to provide some help or allows chatting with contributors.
- **Extensibility**: As we want to integrate a fault tolerance strategy, we needed an extensible framework with code available in a public repository to allow making the required changes for our work.
- **Ease of Use**: We considered frameworks that offer ease of use and simplicity and provide a CLI interface to manage functions.

Taking into account the defined criteria, we selected three FaaS frameworks: Fission [73], Kubeless [110], and OpenFaaS [143]. At the time of choosing, the three frameworks were the first FaaS open source offerings and they have gained large-scale adoption within the community. Therefore, from the popularity/community point of view, the three frameworks have an active community. For example, at the time of writing, the number of GitHub stars is 6545 for Fission, 6813 for Kubeless and 20547 for OpenFaaS. Each framework has also many programmers contributing to its development. The number of Github contributors is 135 for Fission, 116 for Kubeless, and 174 for OpenFaaS. The code source of each framework is publicly available on GitHub (i.e., Fission [74], Kubeless [109] and OpenFaaS [145]).

4.2 Overview of Kubernetes

In this section, we present Kubernetes, a container orchestration platform. The three selected FaaS frameworks, Fission, Kubeless, and OpenFaaS are specifically designed to operate on Kubernetes.

4.2.1 Kubernetes Architecture

Kubernetes [111] is an extensible open source platform used to orchestrate and manage containerized applications. Kubernetes uses pods, deployments, and services to run applications on a cluster.

1. **Pods:** are the smallest deployable units of an application in Kubernetes. They encapsulate either a single container or a group of containers that are running in the same execution environment and sharing an IP address.
2. **Deployments:** are one of the Kubernetes objects used to describe how to run an application container as a pod and control the number of replicas.
3. **Services:** are abstractions that maintain a set of pods in the cluster and define a policy for accessing them. Every service defined in the cluster can be referenced by name that corresponds to one or more pods. The service names are resolved by the Kubernetes DNS server, called *CoreDNS* [56]. The CoreDNS is a service discovery solution that resolves any DNS request to an IP address. It listens for service events and updates its DNS records as needed. These events are triggered when a user creates, updates or deletes the services and their associated pods.

As shown in Figure 4.1, we can distinguish two types of nodes in a Kubernetes cluster: master and worker nodes. We describe in the following the main components of each kind of node.

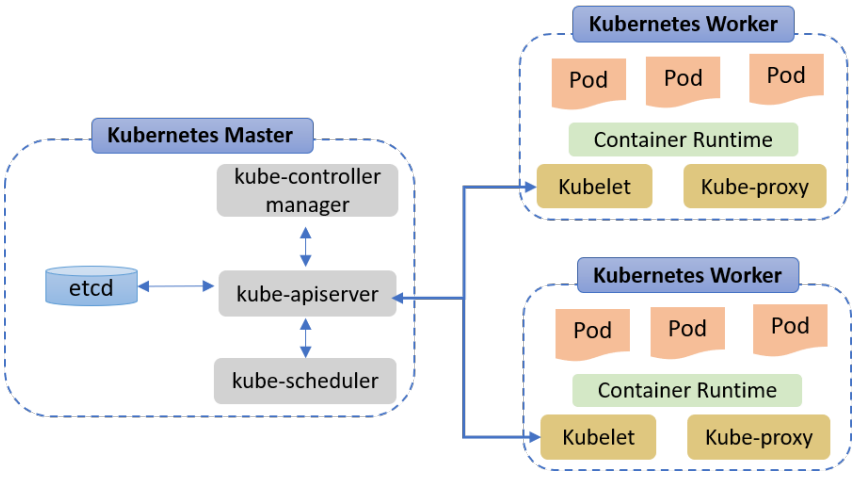


Figure 4.1 – Kubernetes architecture

A) **Master Node Components**

The master node is responsible for managing the cluster. Its main components are: kube-apiserver, etcd, kube-controller-manager, and kube-scheduler.

- **Kube-apiserver:** it receives all REST requests, validates them, and modifies data for API objects, including pods and services.

- **Etcd:** it is a key-value store database used to store and replicate the Kubernetes cluster state.
- **Kube-controller-manager:** it is a daemon that manages a collection of controllers in Kubernetes. A controller is a control loop that continually watches the state of the cluster via the API server. Examples of controllers are the replication controller that controls the number of pods replicas, the node controller that identifies changes that happen in nodes, and the endpoint controller that watches pod lifecycle events and updates endpoints.
- **kube-scheduler:** it determines whether new pods should be deployed and where they should be placed.

B) **Worker Node components**

The worker nodes are used to host and execute the pods. Below are the main components of a worker node:

- **Kubelet:** it is an agent process that runs on each worker node and ensures that pods and their containers are healthy.
- **Kube-proxy:** it is a proxy service on each worker node that maintains the network rules and exposes services to the external world. It performs load balancing for services running on a node.
- **Container runtime:** it is a software used to run user-defined containers (e.g., Docker [59], rkt [162]).

4.2.2 Kubernetes Features Used by FaaS

Kubernetes offers a series of features that are particularly useful for the execution of FaaS applications: auto-scaling, scheduling, load-balancing, health checking, and self-healing of containers.

- **Auto-scaling:** it is one of the major automation capabilities of Kubernetes, useful for responding quickly to peaks in demand. One of the well-known scaling mechanisms is the Horizontal Pod Autoscaler (HPA). The HPA is used to automatically scale up and down the number of pods belonging to the same application based on the current usage of resources, such as CPU or memory utilization.
- **Scheduling:** it is a process to determine the most suitable node for pod placement performed by the kube-scheduler. When the kube-scheduler has a pod to deploy,

it makes sure that the assigned node meets all specific requirements of the pod such as the CPU and memory resource requirements. To achieve that, it first starts by filtering the appropriate nodes using a set of filters. For example, it uses some affinity and anti-affinity rules, which consist of labels and annotations that define constraints on pod placement. Second, the kube-scheduler ranks each node and accords a higher score to nodes with higher affinity and a lower score to nodes with higher anti-affinity. Lastly, the pod is assigned to the node with the highest score.

- **Load-balancing:** it is the process of efficiently dispatching the traffic across multiple pods of a specific service. The traffic sent to a Kubernetes service is routed by the kube-proxy component. The kube-proxy implements a virtual IP for a service via iptables rules and uses by default the random selection mode, which means that when a request is received, it goes to a randomly chosen pod within a service. There are many ways used to expose services externally; the most flexible one is Ingress, which runs as a controller in a specialized pod and provides routing rules to manage the access to the Kubernetes services.

- **Health checking:** is a simple method that Kubelet constantly performs on pods to determine information about their current state. The health check can be based on the readiness probes [112]. A readinesss probe determines if the pods are healthy and ready to start receiving traffic. A pod is considered ready when all the containers inside the pod are ready. When a pod is not ready, it is deleted from service load balancers. There are three ways of implementing the readiness probe: an HTTP request; a TCP socket, where a TCP check is performed against the container's IP/port; and a user-defined command.

- **Self-healing:** it is an automated recovery method that Kubernetes uses to ensure that the actual state of the cluster is healthy. It consists of auto-placement, auto-restart, and auto-replication. For example if a pod crashes, Kubernetes restarts a new one, and if a node goes down, Kubernetes reschedules all the pods from the broken node onto other healthy nodes as soon as it realizes that the node is no longer available (which may take up to 5 minutes).

To take advantage of the rich Kubernetes system, many open source FaaS frameworks move the responsibility of container orchestration functionality to Kubernetes and focus only on FaaS features, as shown in Figure 4.2.

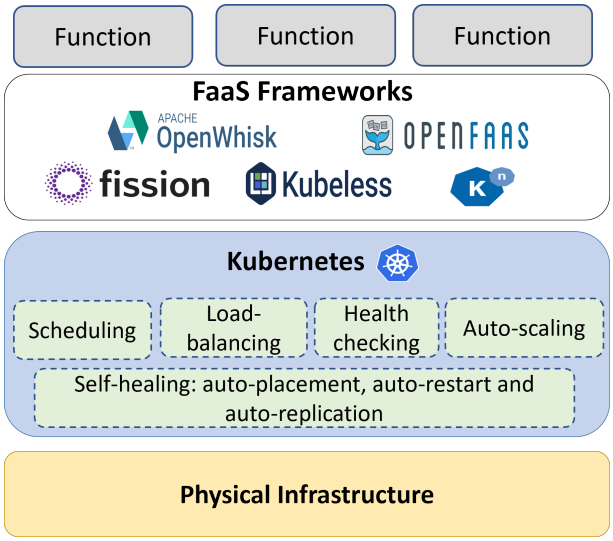


Figure 4.2 – Relation between the FaaS frameworks and Kubernetes

4.3 Kubernetes-native FaaS Frameworks

This section provides a description of the three popular open source Kubernetes-based FaaS frameworks, Fission [73], Kubeless [110], and OpenFaas [143], that we selected for our evaluation.

4.3.1 Fission

Fission [73] is an open-source FaaS framework, released in 2017 and built on top of Kubernetes.

A) **Architecture**

The core Fission components are: Function Pods, Router, and Executor (see Figure 4.3).

- **Function Pods:** they contain function-specific containers to serve requests coming from users.
- **Router:** it routes a function call to the corresponding function pod and retries the call in case of failures.
- **Executor:** it creates and controls the lifecycle of function pods. It provides the requested function pods in two different ways according to the used Executor type: PoolManager or NewDeploy.

(a) **PoolManager:** it maintains pools of warm generic containers and warm function containers in order to provide low cold start latencies [168] and start functions quickly (see Message 3.a in Figure 4.3). However, PoolManager does not allow selecting multiple pods per function, which limits its usefulness during high traffic.

(b) **NewDeploy:** it creates a Kubernetes service to loadbalance the requests between function pods. It uses a Horizontal Pod Autoscaler (HPA) to execute a function and adjust the number of pods to the traffic (see Message 3.b in Figure 4.3).

B) **Features**

- **Language Runtimes:** Fission supports most of the popular languages such as Go, Node.js, Python, Java, Ruby, Binary, C#, .Net, Perl, and PHP. It can be extended to support other languages.
- **Function Triggers:** Functions are invoked using numerous types of triggers like HTTP triggers, timer triggers (cron), message queue triggers (Kafka, NATS, and Azure queues), and Kubernetes watch triggers.
- **Function Monitoring:** Fission automatically tracks various function metrics via Prometheus [154], a monitoring and alerting toolkit that collects and stores metrics. The collected metrics are the number of requests, function execution time, success/failure rate metrics, and more. The Grafana tool [83] can be used with Prometheus to visualize all the metrics.
- **Function Auto-scaling:** Fission scales functions using the Kubernetes HPA. The user needs to set the initial and maximum CPU utilization for a function and target CPU utilization at which autoscaling will be triggered. When the CPU utilization thresholds are reached, the HPA triggers the creation of additional pods to handle the load.
- **Routing:** function invocations in Fission are routed by the Router component. When the Router receives a function call (see Message 1 in Figure 4.3), it first checks if this request already has a running pod and forwards it to the corresponding function pod, if it exists (see Message 2.a in Figure 4.3). Otherwise, the Router requests a function pod from the Executor (see Message 2.b in Figure 4.3) and then forwards the function call to the address (see Message 4 in Figure 4.3, the address is the service address in case of using NewDeploy

or the IP of the pod in case of using PoolManager) returned by the Executor (see Message 5 in Figure 4.3). If a service address is returned by NewDeploy to the router, then the requests are balanced across all function pods using the service.

- **Fault Tolerance:** Fission uses the retry policy to tolerate faults. It is implemented in the Router and triggered if a request fails.
- **Pooling Technique:** Fission runs a pool of warm containers to host functions. The pool size could be configured by the user and is managed by the PoolManager executor. This method is used to reduce function startup latency.

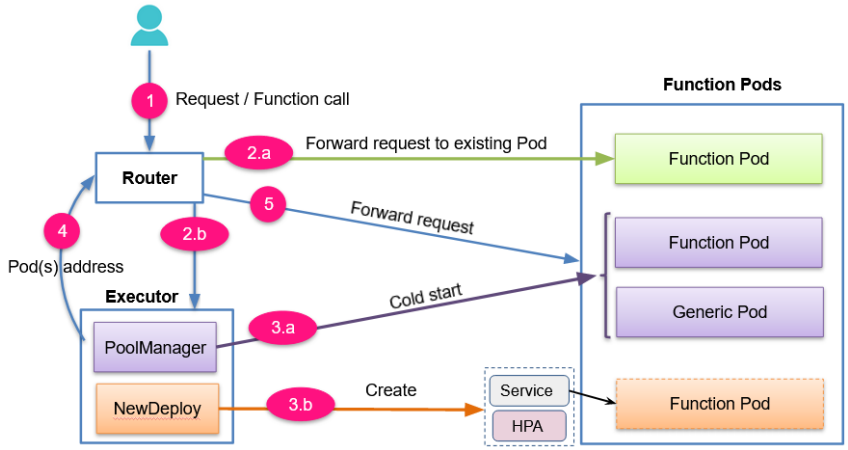


Figure 4.3 – Fission architecture

4.3.2 Kubeless

Kubeless [110] is an open source FaaS framework, released in 2018 and built on top of Kubernetes.

A) **Architecture**

Kubeless relies on Kubernetes abstractions, such as service, pods and controllers. It uses Kubernetes Custom Resource Definition (CRDs) which are extensions of the Kubernetes API to create functions. The functions are managed by the CRD controller and exposed as a Kubernetes service. To make the functions publicly accessible, Ingress resources are used such as Nginx [141], which is a web server acting as a reverse proxy for a controller. Figure 4.4 describes the key components in Kubeless.

B) **Features**

- **Language Runtimes:** Kubeless supports different languages: Node.js, Ruby, Python, Golang, PHP, .NET, Ballerina, and custom runtimes.
- **Function Triggers:** there are three possible types of mechanisms to trigger a function in Kubeless: HTTP triggers, Publish-Subscribe triggers, and Schedule triggers. HTTP triggers use the Ingress controller to make the functions available to the public. Publish-Subscribe triggers use the messages published to Pub/Sub topics to trigger the function. Kubeless supports events from Kafka and NATS messaging systems. Schedule triggers use Kubernetes CronJob [57] to trigger the function, following a given schedule.
- **Function Monitoring:** Kubeless uses the Prometheus toolkit to monitor function metrics such us function call rate, function failure rate and execution duration. These metrics are collected automatically by a runtime and visualized on the Grafana dashboard.
- **Function Auto-scaling:** as each Kubeless function is deployed into a Kubernetes deployment, it relies on the HPA to automatically scale up and down based on the defined workload metrics. Currently, the auto-scaling supported by Kubeless is based on two indicators: the CPU usage and the queries per second (QPS). For example, if a developer needs to autoscale her function based on CPU usage, she has to deploy the corresponding function with the CPU request limit set using the cpu parameter, so that the HPA can control the number of pods that run this function, and maintain the average CPU usage.
- **Routing:** in order to create routes for functions, Kubeless leverages Kubernetes Ingress, which is responsible for routing the HTTP events to functions. For example when using the Nginx Ingress Controller, the function is available to the public and every request to the Kubeless infrastructure is first processed by Nginx, and then transferred to the corresponding service of the function based on the routing rules defined in the Ingress sources. At last, the request is transmitted to the back-end pods according to the load balancing rule (e.g., round-robin) for processing.
- **Fault Tolerance:** to the best of our knowledge, Kubeless does not provide any fault tolerance service.
- **Pooling Technique:** Kubeless does not provide a pool of warm containers.

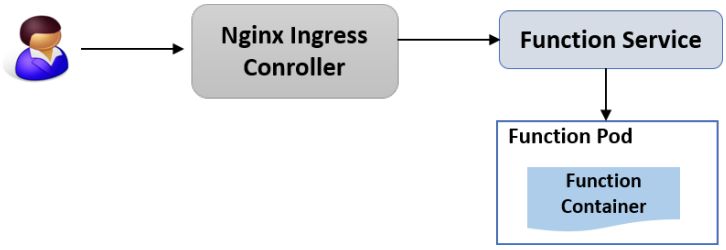


Figure 4.4 – Kubeless architecture

4.3.3 OpenFaaS

OpenFaaS [143] is an open source framework, released for the first time in 2017.

A) **Architecture**

OpenFaaS supports Docker Swarm and Kubernetes orchestration engines. As shown in Figure 4.5, it consists of the following components: API Gateway, Function Watchdog, AlertManager, and Prometheus.

- **API Gateway:** it provides an external route for all functions and it is connected with a respective plugin for the chosen orchestrator. For instance, faasnetes [144] for Kubernetes.
- **Function Watchdog:** it is packaged together with the function and is responsible for forwarding the requests to the functions and starting a new process for each incoming request.
- **AlertManager**: it manages the alerts received from Prometheus and triggers auto-scaling.
- **Prometheus:** it is used for collecting various function metrics which are available via the Gateway's API and enables auto-scaling through AlertManager.

B) **Features**

- **Language Runtimes:** OpenFaaS allows developers to build functions from a set of supported language templates including Java, C#, Node.js, Python, Go, and PHP.
- **Function Triggers:** OpenFaaS functions can be triggered using many event-trigger types. These include the HTTP triggers, the publish-subscribe triggers and the schedule triggers. Different connectors can be used by OpenFaaS such as Apache Kafka [16], Minio [134], CloudEvents [53], AWS SNS [8], IFTTT [96], VMware vCenter [186], and Redis [159].

- **Function Monitoring:** function metrics are available via the Gateway's API and monitored by Prometheus. Prometheus allows to monitor Rate, Duration and Error metrics for each function. All the metrics can be visualized with the Grafana dashboard.
- **Function Auto-scaling:** OpenFaaS allows to auto-scale function replicas based on CPU and/or memory utilization by using the HPA. It also supports auto-scaling based on the number of RPS (requests per second) using the AlertManager.
- **Routing:** each request to the function is routed by the OpenFaas API Gateway which provides an external route into the functions. The requests are distributed between function instances using the load-balancing implemented by service instances.
- **Fault Tolerance:** OpenFaaS provides a retry mechanism only for asynchronous functions [142].
- **Pooling Technique:** OpenFaaS does not have a pool of warm containers.

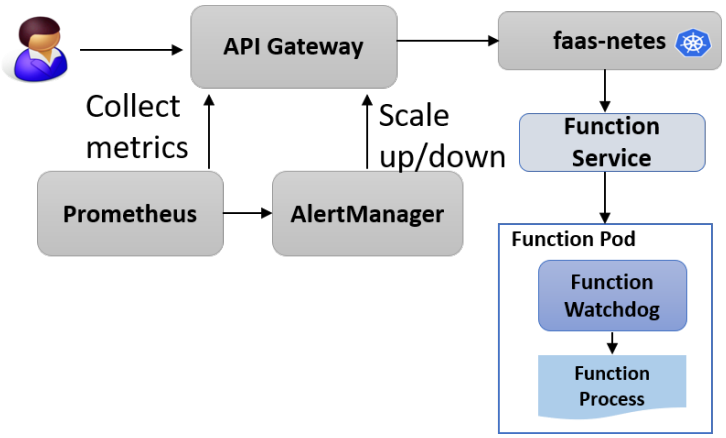


Figure 4.5 – OpenFaaS architecture

4.4 Performance Evaluation

We evaluated the previously-described Kubernetes-native FaaS frameworks, Fission, Kubeless and OpenFaaS, in terms of performance. This allowed us to understand how the FaaS frameworks can be utilized and how they take advantage of the different services

provided by Kubernetes. The main focus of this evaluation is to compare them and investigate the factors, such as concurrency, that influence their performance. The purpose of conducting the comparison was to select the most appropriate framework for hosting our fault tolerance approaches (see Chapter 5). More specifically, we wanted a framework that provides a fault tolerance mechanism and can be easily extended to add new fault tolerance mechanisms; it should also provide good performance (i.e., throughput and response time).

4.4.1 Environment Setup

We ran our experiments on an environment composed of 6 nodes from the Lyon site (nova cluster) of the Grid'5000 [85] testbed, an experimental platform that supports research on all areas of computer science, including Cloud. This testbed provides a large amount of resources which makes it suitable to carry out our experiments. Each node is equipped with 2 CPUs Intel Xeon E5-2620 v4 with 8 cores/CPU and 64 GB memory. We installed Kubernetes 1.11.0 with one master and four worker nodes. We also set up a node that plays the user role to invoke the function deployed on the Kubernetes cluster. The three frameworks we tested are: Kubeless version 0.5.0, Fission version 0.6.0 and OpenFaaS version 0.6.1 (the latest version available at the time of our experiments back in 2018).

4.4.2 Workload Setup

We deployed the hello world function in Python and made some modifications to conform to the programming templates of each framework. The function is triggered by HTTP requests.

The testing load is generated with Apache Bench [14], an HTTP server benchmarking tool. For the measurements, we sent 1, 50, 200, 500, and 1000 concurrent requests.

4.4.3 Metrics

We evaluate the performance of the three frameworks using the throughput and response time metrics. The throughput is the number of requests served per second, and the response time is the time between a user request and the system response.

4.4.4 Results Analysis

Figures 4.6, 4.7, and 4.8 show the throughput under different levels of concurrent requests for Fission, Kubeless and OpenFaas with 1, 5 and 20 function replicas. We observe that Fission has the highest throughput compared to Kubeless and OpenFaaS in all scenarios. This can be attributed to the simplest routing method used by Fission, as we know that Fission maintains its own router. Thus, when the router receives the HTTP requests, it routes them directly to the function instances. The routing used by the API Gateway of OpenFaaS and the Nginx Ingress controller of Kubeless incurs more overhead, as every function call has to go through multiple components to get it routed to the function pod. We also notice that the throughput of OpenFaaS is significantly lower than the others.

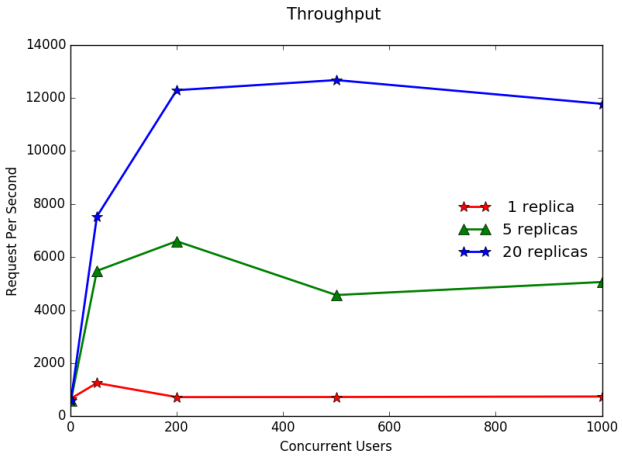


Figure 4.6 – The throughput for Fission with 1, 5 and 20 function replicas.

Figure 4.9 and Table 4.1 show the response time for a concurrency level of 200 with one function replica for the considered frameworks.

Regarding the time spent processing the requests, the average Fission response time is 161 ms and all responses are received within 13091 ms (13 sec). We notice that response times jump up, peaking around 98531 ms at 99% and 60001 ms at 90% in Kubeless and OpenFaaS, respectively. Kubeless and OpenFaaS suffer from higher average response times compared to Fission, with a value of 1984 ms for Kubeless and 11549 ms for OpenFaaS. This is due to queuing requests at both the gateway and watchdog components in OpenFaaS and at the Ingress controller component in Kubeless. Overall, Fission has the best average responsiveness.

As a conclusion, Fission exhibits the best performance among the three frameworks,

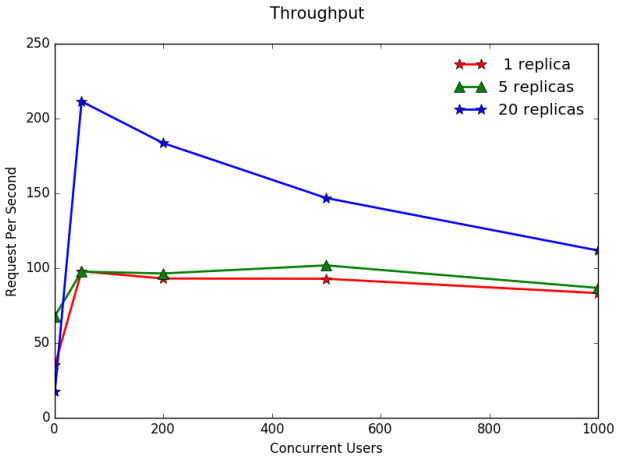


Figure 4.7 – The throughput for Kubeless with 1, 5 and 20 function replicas

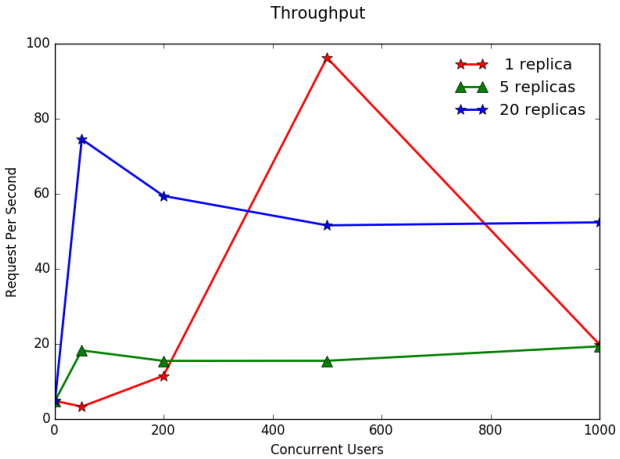


Figure 4.8 – The throughput for OpenFaaS with 1, 5 and 20 function replicas

both in terms of throughput and average response time.

4.5 Overall Framework Comparison

This section provides a comparative analysis of the discussed FaaS frameworks. The objective of this analysis is to choose a suitable framework for our study.

Table 4.2 summarizes the complete comparison of the selected frameworks. The comparison shows that the frameworks offer similar features. For instance, all the frameworks have the principal autoscaling mechanism, based on the Kubernetes HPA feature. Contrary to Kubeless and OpenFaaS, Fission has additional useful features, such as the pooling feature, where a pool of warm containers is maintained to reduce cold starts of

Percentage	Time (ms)		
	Fission	Kubeless	OpenFaaS
50%	5	11	19
66%	150	14	24
75%	219	289	37
80%	247	303	18023
90%	304	402	60001
95%	565	500	60002
98%	1601	3321	60006
99%	1659	98531	60012
100%	13091	105876	61009
Min	1	2	1
Mean	161	1984	11549
Max	13091	105876	61009

Table 4.1 – The percentage of requests response time in milliseconds (ms)

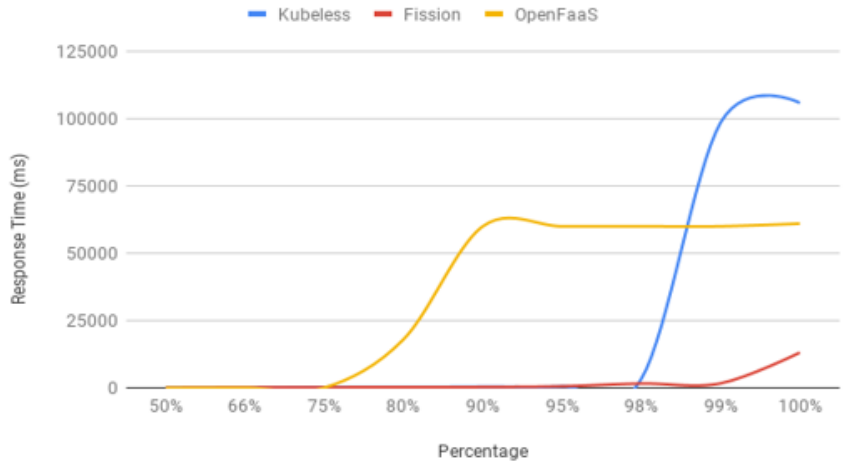


Figure 4.9 – Response time for concurrency of 200 with one function replica for three FaaS frameworks

functions (cold start is a well-known challenge in FaaS platforms as seen in Section 3.3). Fission also supports fault tolerance with applying a basic retry mechanism to re-execute a function when an error is detected. In terms of performance, based on our experiments, Fission outperforms the two other frameworks. After evaluating and analyzing the different features offered by each framework, we were convinced that the most suitable framework for our use case was Fission not only because of the good performance provided, but also because it had an integrated fault tolerance mechanism (retry) that we intended to compare with our fault tolerance approaches.

Features/ Frameworks	**Fission**	**Kubeless**	**OpenFaaS**
Licence	Open-source	Open-source	Open-source
Orchestration	Kubernetes	Kubernetes	Kubernetes and Docker Swarm
Function Runtimes	Go, Node.js, Python, Java, Ruby, Binary, C#, .Net, Perl and PHP	Node.js, Python .NET, Ruby, Ballerina, and PHP	Java, C#, Node.js, Python, Go, PHP and Dockerfile
Function Triggers	HTTP, timer, message queue and Kubernetes watch	HTTP, scheduled, Pub/Sub	HTTP, scheduled, Pub/Sub
Function Autoscaling	HPA	HPA	HPA/AlertManager
Function Monitoring	Prometheus+Grafana	Prometheus+Grafana	Prometheus+Grafana
Function Runtime Pooling	Yes	No	No
Routing	Router/Ingress Controller	Ingress Controller	API Gateway/Ingress Controller
Fault Tolerance	Yes (Retry)	No	Yes (Retry)
Performance	High	Medium	Low

Table 4.2 – Comparison of Fission, Kubeless, and OpenFaaS

4.6 Summary

In this chapter, we presented three open source FaaS frameworks based on Kubernetes, namely, Fission, Kubeless and OpenFaaS. We selected the frameworks based on a set of criteria such as public availability, popularity and community support. Then, we evaluated them with a focus on the end-user's perception of performance. Specifically, we compared their throughput and response time. Results showed that the performance of Fission increases with the number of function replicas and it has the best throughput and response time compared to Kubeless and OpenFaaS. We found that OpenFaaS has poor performance regardless of the number of function replicas.

As a conclusion, we ended up choosing Fission due to its ease of deployment, flexibility and good performance. Fission gives the possibility to create functions in two different ways by using the PoolManager and NewDeploy executors. Fission has also an integrated fault tolerance mechanism (retry) that we intend to compare with our fault tolerance

solutions.

In the next chapter, we describe the basic retry mechanism used in Fission. We also present our proposed fault tolerance approaches and their integration in Fission.

Chapter 5

Fault Tolerance Approaches For High Availability in FaaS

In FaaS environments, retrying function executions in case of failures is a common strategy for tolerating faults. The retry mechanism is well-suited for transient faults. However, it is not effective for handling other types of faults, such as node failures, a frequent type of fault in underlying infrastructures. This motivates the need for other fault tolerance approaches. In our work, we propose to integrate two fault tolerance approaches in FaaS frameworks in order to make failures transparent to the applications. The proposed approaches are based on replication (i.e., active and passive).

The remainder of the chapter is organized as follows. In Section 5.1, we describe the retry mechanism, the native fault tolerance mechanism in Fission. In Section 5.2, we present the first proposed fault-tolerance approach, namely Active-Standby and its implementation in Fission. The second approach, called Request Replication with its implementation is presented in Section 5.3. Finally, Section 5.4 provides a summary.

5.1 Retry Mechanism in Fission

We present in this section the native retry mechanism implemented in most FaaS platforms including Fission. The retry fault tolerance mechanism consists basically in restarting the entire submission process of a failed request. In Fission the retry mechanism works as shown in Figure 5.1. When a function call is received, the Router first checks whether a function service record exists in its cache. If it doesn't, it asks the Executor to get a new service for the function. Once the new record is returned, the Router forwards the request to the function. If the function execution fails, the Router retries to forward again the function call until receiving a response from the function execution or reaching the maximum number of retries set by the administrator [75]. If all the retries fail or the received response is an error, Fission assumes that the function pod doesn't exist

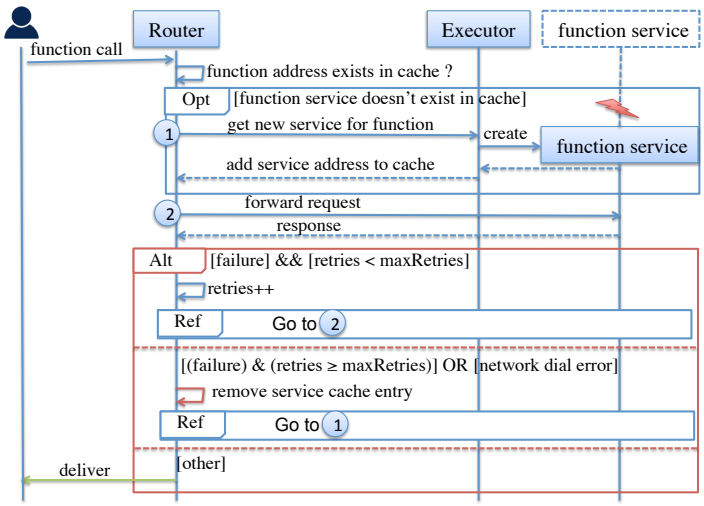


Figure 5.1 – Retry mechanism in Fission

anymore. Thus, the Router asks the Executor for a new service for the function. Then, it retries to forward the function call to the new function service and so on until the request is served.

5.2 Active-Standby in Fission

In this section, we present the Active-Standby approach, along with details of its implementation in the Fission framework.

5.2.1 Description

In the context of FaaS, the Active-Standby (AS) mechanism consists in creating two function service instances. The first one is active and serves all requests during normal usage. The second one is passive (on standby). The two instances are connected by a heartbeat mechanism that continuously checks their connectivity and status. If the heartbeat of one instance is not received within a configured amount of time, an action is triggered depending on the identity of the unreachable instance. If the passive instance is unreachable, another passive instance is created. If the active instance is unreachable, the standby instance is activated to serve incoming requests and another passive instance is created.

5.2.2 Implementation in Fission

To implement the AS mechanism in Fission, we use the NewDeploy executor type as it supports creating replicas of function pods. In this approach, two function pods are created (i.e., active and passive) and both support the Kubernetes readiness probe that indicates when the container is ready to receive requests. For instance, the active pod is marked in ready state and is therefore ready to receive and serve traffic. The passive pod is in standby and is marked in not-ready state, so no traffic is forwarded to it. Both active and standby pods exchange heartbeats for health checks. The heartbeats are performed each second (the minimum configurable value using Kubernetes readiness probes). When the active pod is healthy, the passive pod fails the readiness probe and keeps running in a not-ready state. If the active pod fails, the health checks performed by the passive pod detects the failure and the passive pod becomes active. Another pod is then created to replace the passive pod. The same action happens if the passive pod fails.

We implemented this approach in Fission in two different ways: The first implementation relies on a component from Kubernetes, and the second one is based on using a router component. The two implementations are described in the following.

- ***Implementation 1: Active-Standby with Kubernetes CoreDNS***

  In the first implementation, we use a specific component from Kubernetes, a DNS server called *CoreDNS*, to get the IP address of the active pod. The approach works as follows as shown in Figure 5.2. The Kubernetes *CoreDNS* receives the function call and returns the IP address of the active pod to the user. The user forwards her request directly to the active pod.

- ***Implementation 2: Active-Standby with a Router***

  In the second implementation, we implemented a new router, called Router Active-Standby (Router AS), and used it instead of the default Fission Router (this is what makes this implementation different from the previous one). The Router AS forwards all received function calls specifically to the active pod, as shown in Figure 5.3.

  This approach works as shown in Figure 5.4. While the request is being processed, both active and standby pods exchange heartbeats for health checks.

The two implementations of AS use a different component to route requests. In the first implementation, the architecture makes use of the *CoreDNS* of Kubernetes to get the IP address of the active pod and then route the traffic to that address. The second implementation involves a router component instead of the *CoreDNS*. The second imple-

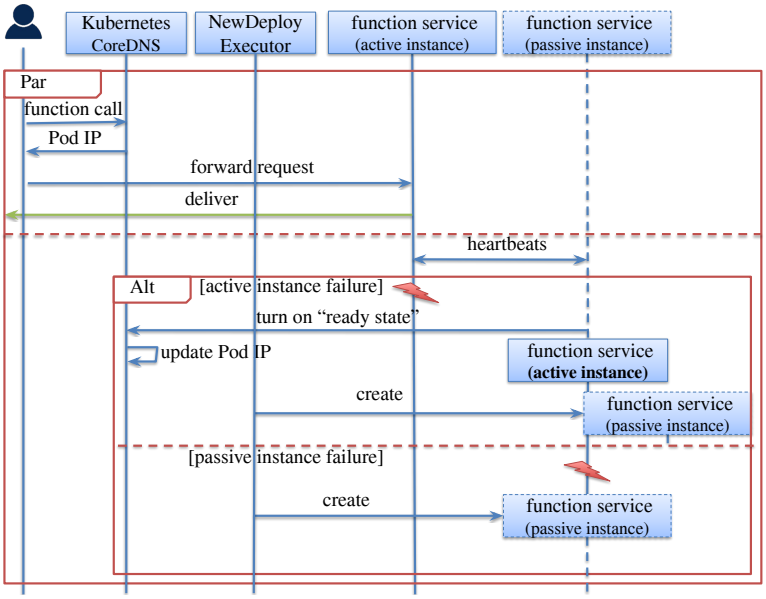


Figure 5.2 – Sequential diagram of Active-Standby mechanism using the Kubernetes *CoreDNS*

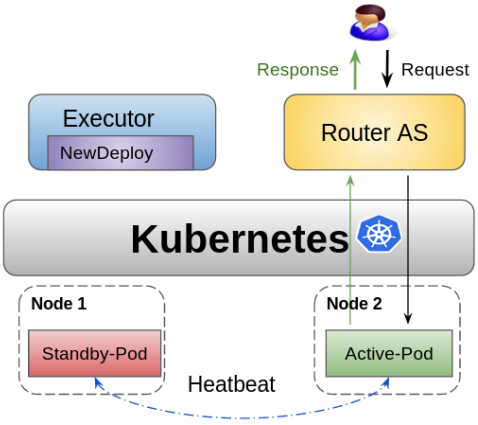


Figure 5.3 – Overview of the Active-Standby mechanism in Fission (Implementation 2)

mentation of AS was useful to enable a fair comparison with the retry and the Request Replication approaches, which also use a router (see Chapter 6).

5.3 Request Replication in Fission

In this section, we present the Request Replication approach, along with details of its implementation in Fission framework.

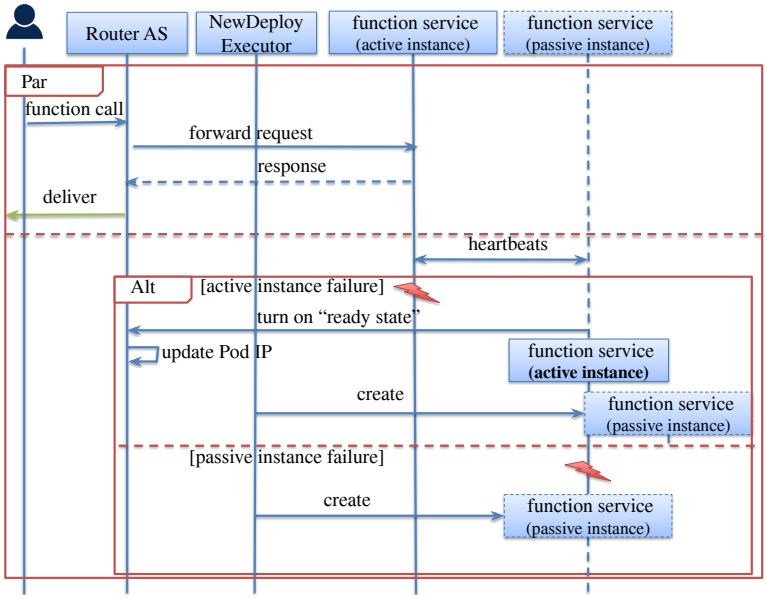


Figure 5.4 – Sequential diagram of Active-Standby mechanism using a router

5.3.1 Description

Request Replication consists in having a number of replicas processing a request at the same time. The number of replicas depends on the number of simultaneous failures to be tolerated. The Request Replication (RR) solution is divided into two phases. First, the client sends a request, and the request is received and processed simultaneously by all replicas. Second, the first response produced by any replica is delivered to the client. The client can thus receive a response despite the failure of some replicas.

5.3.2 Implementation in Fission

To implement the RR approach in Fission, we used the NewDeploy Executor as it allows to create many pod replicas. We replaced the default Router with a new one, called Router Request Replication (Router RR). This Router replicates each received request on all function pod replicas, in order to process it in parallel. Then it sends the first received response to the user, as shown in Figure 5.5. To tolerate K failures using this approach, it is necessary to have a minimum of K+1 replicas, so that the Router can ensure that the user always receives a response. If one of the pods fails, then another one is created to replace the failed one. Figure 5.6 illustrates the implementation of the request replication approach in Fission.

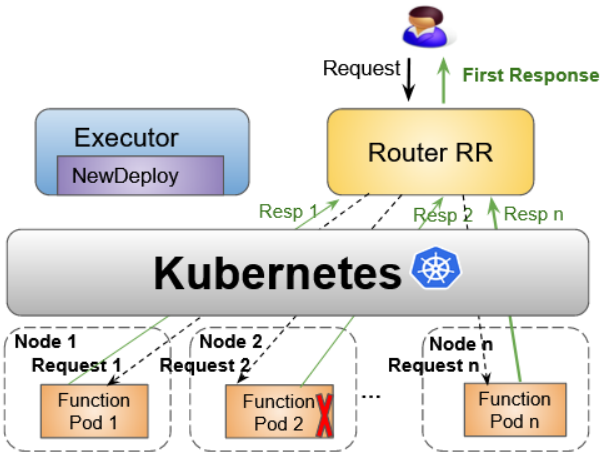


Figure 5.5 – Overview of the Request Replication mechanism in Fission

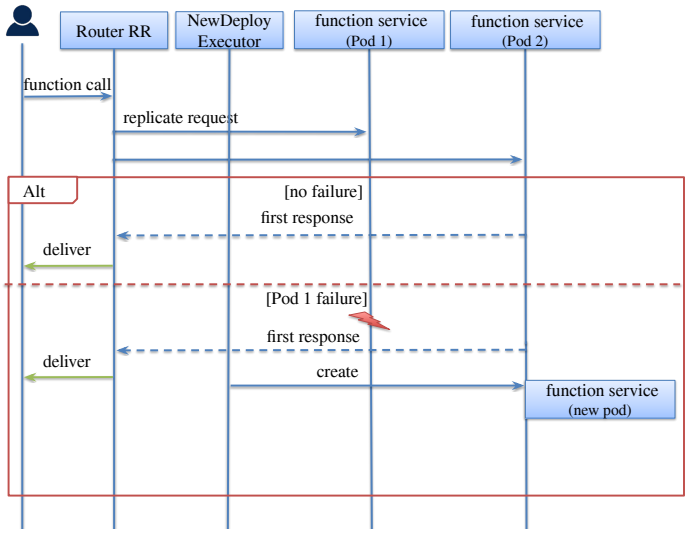


Figure 5.6 – Sequential diagram of Request Replication mechanism in Fission

5.4 Summary

In this chapter, we covered the proposed fault tolerance approaches for FaaS and their implementation in an open source FaaS framework, namely Fission. Fission implements a built-in retry mechanism. This technique deals with transient failures (e.g., network failures) by transparently retrying a failed request. However, ensuring high availability of FaaS functions requires the use of other fault tolerance techniques such as replication strategies. Two replication-based approaches and their implementation in Fission were discussed in this chapter. The first approach, called Active-Standby is based on the use of

passive replication. It consists of creating two function replicas, one is active and another is standby. This approach was implemented in two versions: the first version uses the Kubernetes *CoreDNS* component to communicate directly with the active pod, the second version uses an implemented router component to send requests to the active pod. The second fault tolerance approach, named Request Replication is based on the active replication. It relies on replicating requests in which every request is executed concurrently by every function replica. We present a comparative evaluation of these approaches in the next chapter.

Chapter 6

Evaluation

This chapter presents the evaluation of the fault tolerance approaches presented in the previous chapter. Our experiments evaluate the effectiveness of the proposed FaaS fault tolerance approaches and compare them with the retry mechanism in the context of their implementation in Fission.

In the following, we describe an experimental study which is conducted on the Grid'5000 testbed. Two different experiments are performed. In the first experiment, we compare the first implementation of the Active-Standby approach, which uses the Kubernetes *CoreDNS* service, with the retry mechanism. In the second experiment, we evaluate the second implementation of the Active-Standby approach, which uses a router component, with the Request Replication approach and the retry mechanism.

The chapter is organized as follows. Section 6.1 provides a detailed description of the setup of the two experiments. Section 6.2 discusses the results of the first experiment. Section 6.3 analyses the results of the second experiment. Section 6.4 provides the lessons learned from our experiments. Section 6.5 summarizes the chapter.

6.1 Experimental Setup

6.1.1 Environment

We used 5 nodes on the Lyon site, each node having 2 CPUs Intel Xeon E5-2620 v4 with 8 cores/CPU and 64 GB memory, to deploy Kubernetes [111] (version 1.11 in Experiment 1 and version 1.19 in Experiment 2). In this cluster, we have one node for the Kubernetes master and one node is used for Fission components. The three other nodes are workers, where the function pod is placed. For each experiment we use either Fission AS (Active-Standby), Fission RR (Request-Replication) or the original version of Fission (vanilla) with version 1.5.0 in Experiment 1 and version 1.10.0 in Experiment 2 (the latest stable release at the time of their implementation). We set up 2 additional nodes; one is

used as a client in order to invoke functions and another one to inject faults.

We used the same environment as the one used in the Experiment 1. However, we updated the version of Kubernetes to a more recent version (version 1.19) and we implemented the AS and the Request Replication approaches in a more recent version of Fission (version 1.10.0).

6.1.2 Applications

(a) **Experiment 1**

We used two applications. The first one is a CPU-intensive HTTP-triggered function that computes the Fibonacci sequence (a series of numbers where each number is the sum of the two preceding ones). The function takes n=15 as an input, computing the 15th term of the sequence. The second one is the Guestbook application, composed of two functions GET and ADD to read and write text messages, which are stored in a Redis database [86].

(b) **Experiment 2**

In this experiment, we used only the Fibonacci function because the Guestbook application code would need to be modified to handle the duplicated requests to Redis when using it with the Request Replication approach.

6.1.3 Workload

(a) **Experiment 1**

In this experiment, we generated 3000 requests during 5 minutes.

The workload is generated with Tsung [181], an open source load testing tool written in Erlang. Tsung allows to test the scalability and performance of applications and databases, ect. It can be used to stress HTTP, WebDAV, SOAP, PostgreSQL, MySQL, LDAP, and Jabber/XMPP servers.

(b) **Experiment 2**

In this experiment, we used Tsung to generate 60000 requests during 10 minutes with 100 concurrent users created every second.

6.1.4 Failure Scenarios

(a) **Experiment 1**

- **Pod failure**: the application failure is due to a pod failure. In this scenario, we use the PowerfulSeal tool [153], a failure injection tool for Kubernetes to inject faults to pods. The failure is simulated by killing the function pod at a random time between 30 seconds and 60 seconds from the beginning of the workload execution. The failure is injected randomly in the active and passive pod for AS.
- **Node failure**: the application failure is due to a node failure. In this scenario, we use a script to crash nodes. The failure is simulated by killing the node hosting the function instance 30 seconds after the beginning of the workload execution.

(b) **Experiment 2**

- **Pod failure:** in this scenario, the function pod is killed at the 5th minute from the beginning of the workload execution.
- **Node failure:** in this scenario, a script is executed to kill the node hosting the function instance 5 minutes after the beginning of the workload execution.
- **Network delay:** in this scenario, we injected latency at the 5th minute and it lasts for 10 seconds. The injected latency values are 50 ms, 100 ms, and 200 ms. We note that the injected latency causes a delay for all responses coming from the function pod. We added this scenario in this experiment to see how the three approaches react to network issues.

  To execute the pod failure and the network delay scenario, we used the Chaos Mesh tool [46]. It is an open source tool for injecting various failure scenarios including the network delay. The network delay scenario is not available in the experiment list of PowerfulSeal tool. For that reason, we used Chaos Mesh instead of PowerfulSeal.

  In the three scenarios, the failure is injected in the active pod for AS and in one of the two pods for RR.

Each scenario was repeated at least 5 times with the deployed applications in Fission vanilla, AS, and RR. The averages of the measurements are shown in the figures and tables of this chapter.

6.1.5 Metrics

- Performance: The performance is measured using throughput and response time

values.

- Availability: The availability is measured using the recovery time, which is the time between the first reaction to failure and the time when the service is available again.
- Resource consumption: The resource consumption is measured as the amount of CPU and memory consumed by the 5 nodes (i.e., the master and the 4 workers) during the execution of the workload.
- Error rate: The error rate is measured based on the failed requests (those with HTTP 5xx response code). This metric is a useful measure of the approaches' performance (only used in Experiment 2).

6.2 Experiment 1: Active-Standby with CoreDNS versus Retry

In this experiment we compare the first implementation of the Active-Standby approach using the *CoreDNS* service with the native retry mechanism in Fission. Subsection 6.2.1 provides the performance results. Subsection 6.2.2 presents the availability results. Subsection 6.2.3 discusses the resource consumption results.

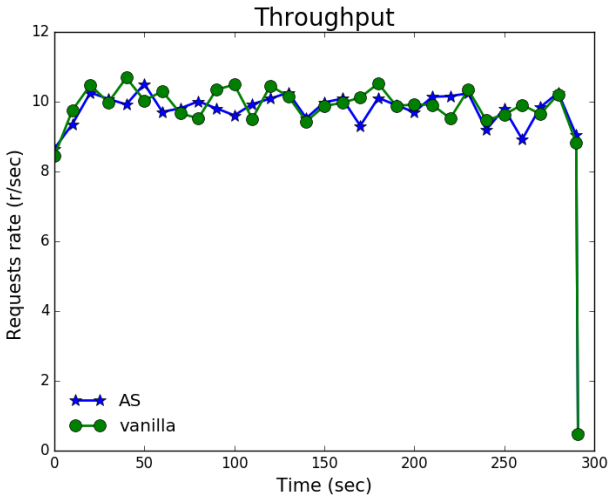
6.2.1 Performance Results

This section presents the result analysis of throughput and response time of the performed experiments with and without failures.

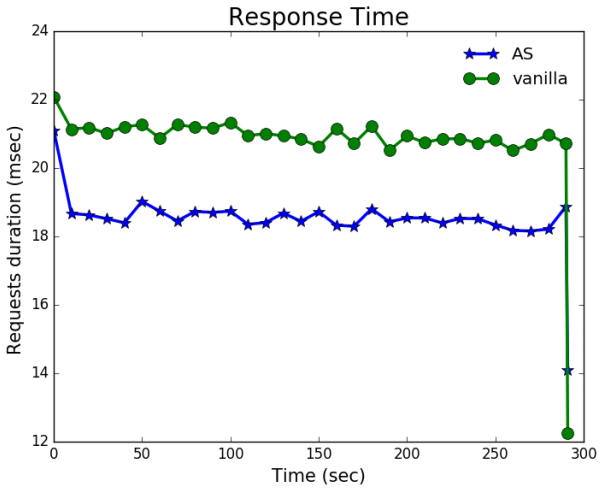
(a) **Results without Failures**

Figures 6.1 and 6.2 present the throughput and average response time of the Fibonacci and Guestbook applications deployed with both Fission vanilla and AS without failures. From Figure 6.1a and Figure 6.2a, we can observe that the throughput for the two functions in the two versions of Fission are quite similar. AS and vanilla are both capable of processing in average 11 requests per second. Figures 6.1b and 6.2b show the response times of the two functions in AS and vanilla. Both functions, Fibonacci and Guestbook, have a lower response time with Fission AS; the difference is about 2 ms and 16 ms respectively. The higher response time obtained with Fission vanilla is due to the use of the Router component to route the request

to function instances. AS performs better than vanilla in this scenario and provides faster response times.

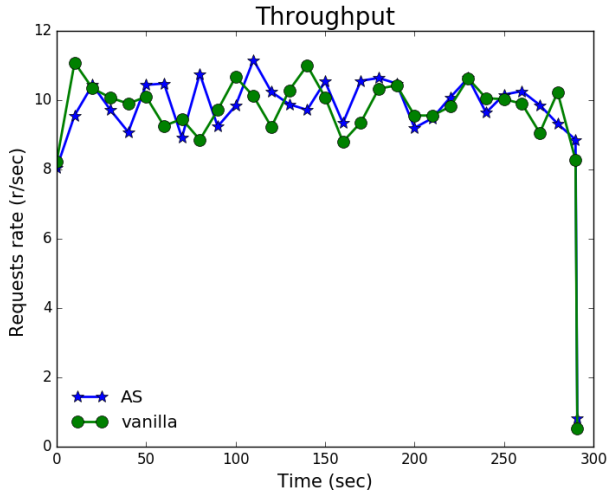


(a) Throughput

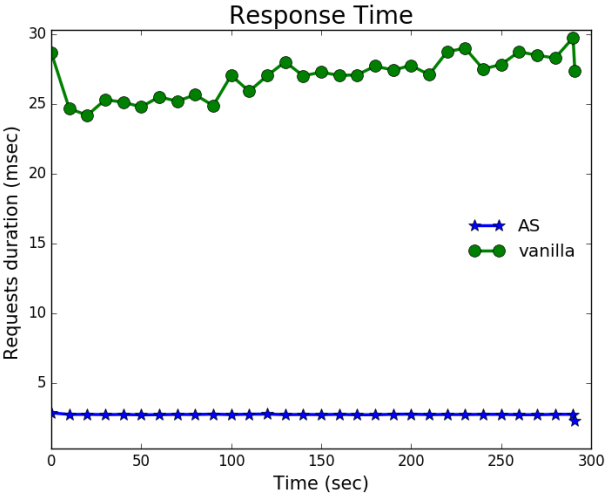


(b) Response Time

Figure 6.1 – Fibonacci application without failures



(a) Throughput



(b) Response Time

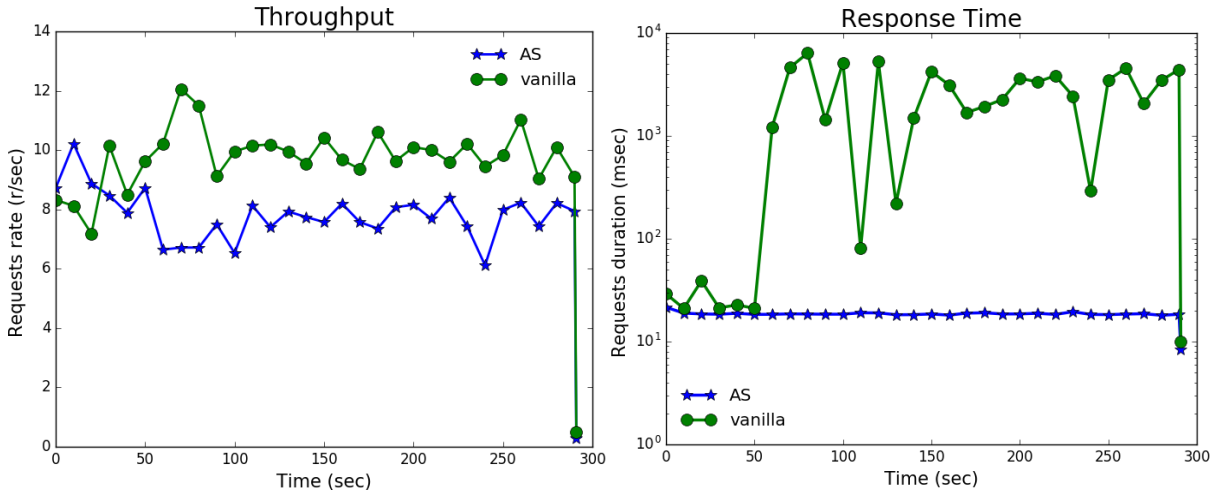
Figure 6.2 – Guestbook application without failures

(b) Results with Failures

1) Pod Failure Scenario

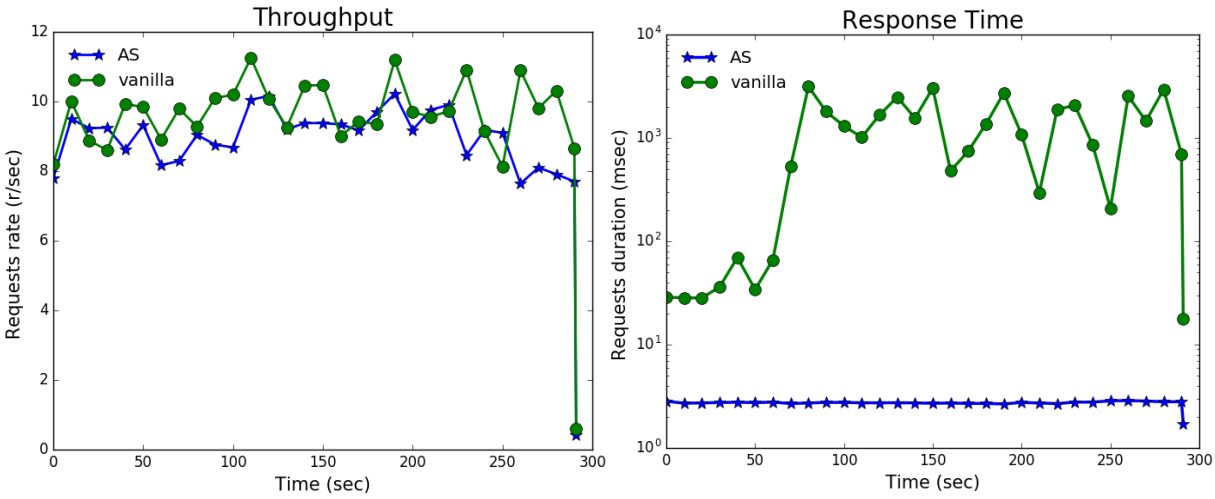
Figures 6.3 and 6.4 show the throughput and average response time of the Fibonacci and Guestbook applications with Fission vanilla and AS, with pod failures. We see that AS and vanilla react to the failure differently. This can

be explained by the fact that vanilla retries many times the function execution until reaching the maximum number of attempts, then removing the function instance from the cache and recreating a new one. In vanilla, recreating a new function instance involves initialization of the function environment, which results in increased latency (the requests are queued for longer amounts of time) compared to AS, where the standby instance is prepared to take over at any time and serves requests.



(a) Throughput (b) Response Time

Figure 6.3 – Fibonacci application with pod failures

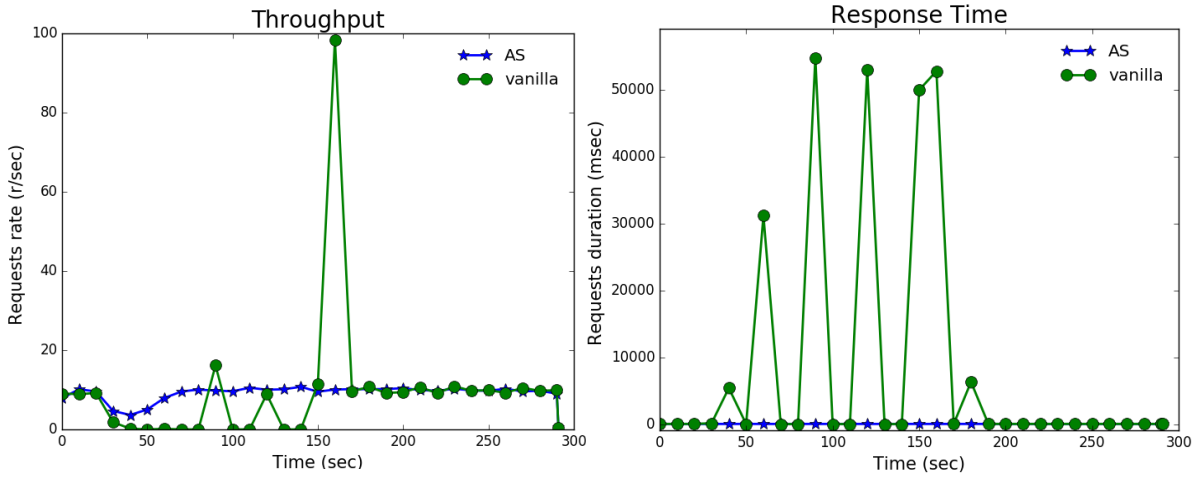


(a) Throughput (b) Response Time

Figure 6.4 – Guestbook application with pod failures

2) **Node Failure Scenario**

Figures 6.5 and 6.6 show the throughput and average response time of Fibonacci and Guestbook applications with Fission vanilla and AS, with node failures. In Figure 6.5a and Figure 6.6a, we notice peaks in the throughput for both functions in vanilla. This can be explained as follows. After a node crash, requests are queued, creating unbalanced traffic. Thus, the waiting time of queued requests is increased and consequently their response time, as can be seen in Figures 6.5b and 6.6b. However, in Fission AS the response rate is almost constant as the requests are just redirected to the standby instance.



(a) Throughput

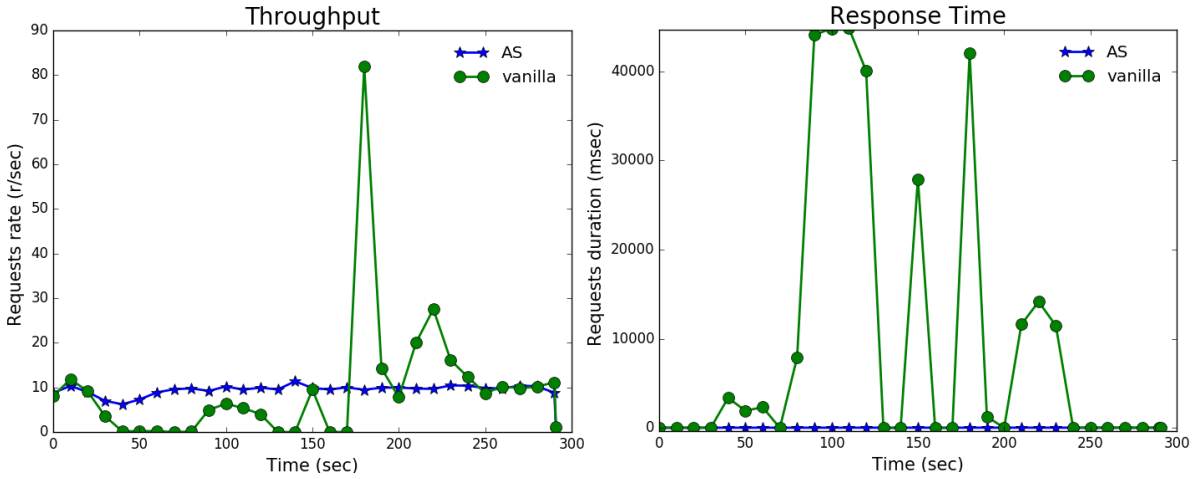
(b) Response Time

Figure 6.5 – Fibonacci application with node failure

6.2.2 Availability Results

The recovery time is the required time for a service to recover from failures and become available again. This covers the time between failure detection and resuming service operation. The recovery time is measured for the two approaches as follows. For vanilla, after the failure, the pod becomes unhealthy (see Figure 6.7). In reaction to that failure, the router retries the failed requests. When the maximum number of retries is reached, the pod is considered as failed and the service URL is deleted from the router cache. The service becomes available again when a new pod is created and added to the router cache.

For AS, the failure is detected by the heartbeat mechanism (see Figure 6.8). The reaction is the failover to the standby pod and the update of the CoreDNS cache. Once



(a) Throughput (b) Response Time

Figure 6.6 – Guestbook application with node failure

the CoreDNS cache is updated with the IP address of the active pod (Active-IP), the service becomes available.

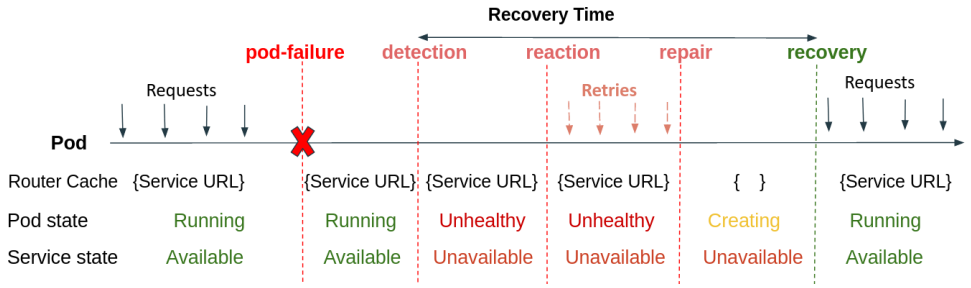


Figure 6.7 – Recovery time in vanilla

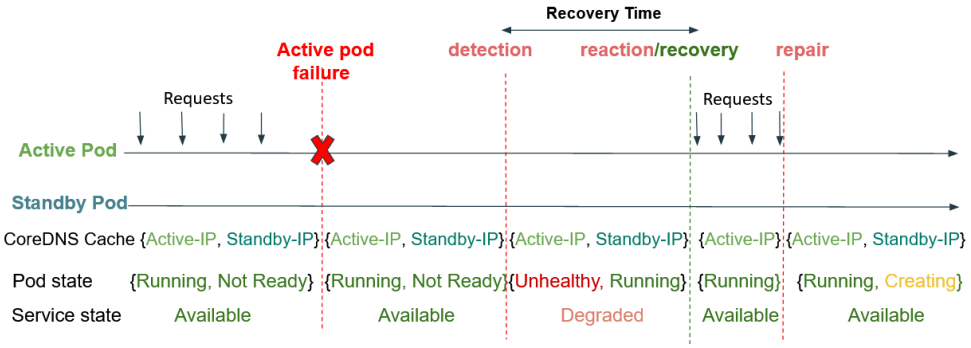


Figure 6.8 – Recovery time in AS

1) **Pod Failure Scenario**

The recovery time (RT) in Fission vanilla and AS with pod failures is shown in Table 6.1. We can see that the RT of Fibonacci and Guestbook functions under AS is 1.814 seconds and 1.528 seconds, respectively, whereas under vanilla is 2.840 seconds and 3.614 seconds, respectively. These results show that AS enables faster recovery than the retry mechanism used in vanilla. This is because in AS, the failover is triggered immediately just after the active pod failure is detected. However, vanilla takes time to recover from pod failure, since recovery requires the recreation of a new function instance from scratch.

Table 6.1 – Recovery time in Fission vanilla and AS with pod failures

	Fission vanilla	Fission AS
Fibonacci Function	2.840s	1.814s
Guestbook Application	3.614s	1.528s

2) **Node Failure Scenario**

The recovery time in Fission vanilla and AS with node failures is shown in Table 6.2. We can see that RT of Fibonacci and Guestbook functions under AS is 6.384 seconds and 6.194 seconds, respectively, whereas under vanilla is 3 minutes and 7 seconds and 2 minutes and 39 seconds, respectively. We clearly see that AS performs much better than vanilla in terms of availability.

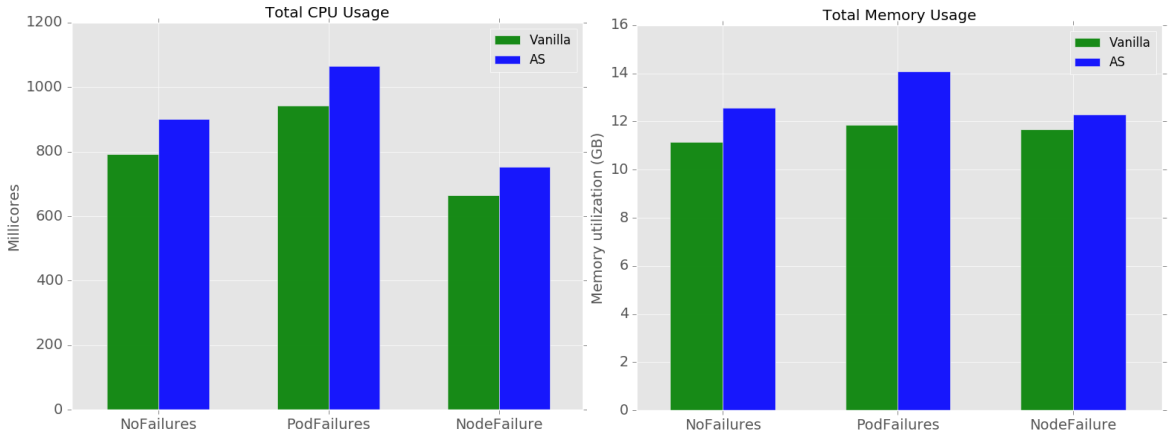
Table 6.2 – Recovery time in Fission vanilla and AS with node failures

	Fission vanilla	Fission AS
Fibonacci Function	3min7s	6.384s
Guestbook Application	2min39s	6.194s

6.2.3 Resource Consumption Analysis

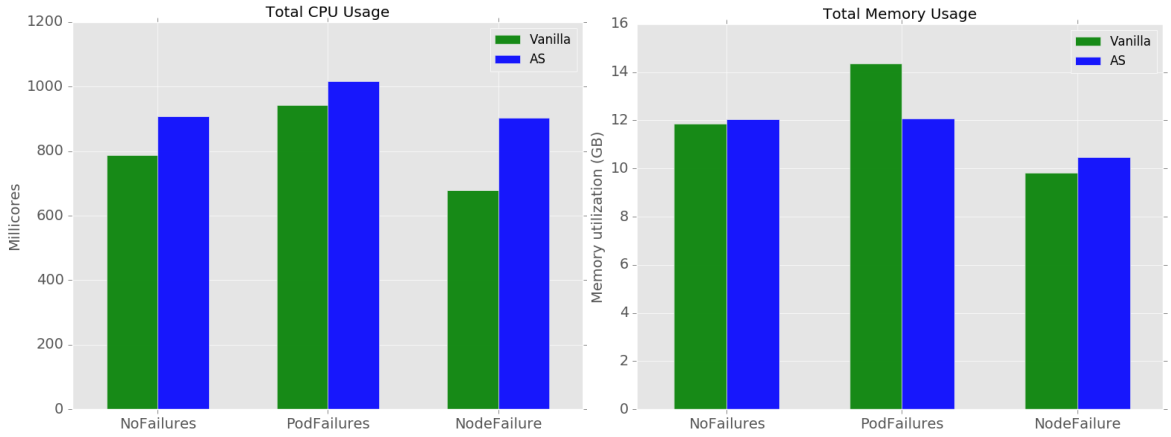
Figure 6.9 and Figure 6.10 show the resource consumption (CPU, memory) in the Kubernetes cluster for Fibonacci and Guestbook applications executed on top of Fission AS and vanilla without and with pod and node failures. In the scenarios without and with pod failures, we measured the overall CPU and memory usage of the 5 nodes during the execution of the workload. In the node failure scenario, we took measures of only 4 nodes

(we excluded the consumption of the failed node). We notice that for both functions the cluster uses more CPU and memory with AS compared to vanilla in the three scenarios. For example, when there are no failures, the overhead of using AS is up to 15% in CPU and 12% in memory consumption. This is due to the creation of two instances of each deployed function in Fission AS instead of one in vanilla.



(a) CPU consumption (b) Memory consumption

Figure 6.9 – Resource consumption of Fibonacci in Fission vanilla and AS without and with pod and node failures



(a) CPU consumption (b) Memory consumption

Figure 6.10 – Resource consumption of Guestbook application in Fission vanilla and AS without and with pod and node failures

6.3 Experiment 2: Active-Standby with Router versus Request Replication and Retry

In this experiment, we compare the second implementation of the Active-Standby approach based on the router with the request replication approach and the native retry mechanism in Fission. Subsection 6.3.1 presents the performance results. Subsection 6.3.2 discusses the availability results. Subsection 6.3.3 analyses the resource consumption results.

6.3.1 Performance Results

This section presents the results obtained from the experimental comparison of our proposed fault tolerance strategies AS and RR with the native retry mechanism of Fission.

(a) **Results with no Failures**

Figure 6.11 shows the response time of Fission AS, vanilla, and RR with no failures. From this figure we can notice that Fission RR is faster than Fission AS and vanilla because it has two replicas doing the job and once it receives the first response from one of the function replicas, it forwards it to the user. Vanilla is slower than AS and RR. This may be explained by the fact that the router of vanilla uses the Kubernetes service to send the request to the pod belonging to the function, which adds another hop compared to AS and RR. In AS and RR the request is sent directly to the function pod by the router.

For the throughput, Fission vanilla, AS, and RR handle the same throughput with values around 100 request/second (the expected throughput).

As a conclusion we can say that RR performs better than AS and vanilla in terms of response time when there are no failures.

(b) **Results with Failures**

1) **Pod Failure Scenario**

Figures 6.12 and 6.13 illustrate the throughput and response time of Fission vanilla, AS, and RR with only one pod failure. In Figure 6.12, we can observe a small degradation in the throughput of AS. This is because of the failover of the active pod to the standby pod. We also notice a degradation in the throughput of vanilla when the pod fails after 300 seconds as there is no available pod to

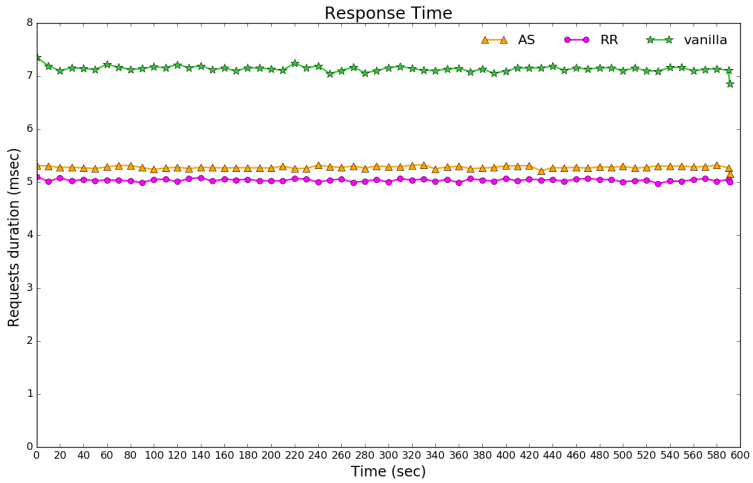


Figure 6.11 – Response time of AS, vanilla and RR with no failure

serve the requests. RR provides stable throughput despite the pod failure since all traffic is executed by the healthy replica.

In Figure 6.13, we notice some spikes in the response time of vanilla during almost 30 seconds. This is attributed to the fact that once the pod failure is detected, the router starts the retries. When the function pod recovers, we see that the response time drops off at around 7 ms. In contrast, RR and AS provide stable response times with values around 5 ms.

2) **Node Failure Scenario**

Figures 6.14 and 6.15 present performance results of Fission vanilla, AS, and RR with a node failure. Figure 6.14 shows a degradation in the throughput with AS when the node hosting the active pod crashes. This is because of the required actions to switch the passive pod to the active state. In vanilla, the throughput drops when the function pod stops serving requests. The router then starts the retries and the requests are queued until a new pod starts running on a healthy node. This causes a spike in throughput that reaches 1300 requests/sec, and then drops back to a normal state. The throughput of RR remains constant because the failure is masked by the presence of the other replica that continues to process the user's requests.

Figure 6.15 shows spikes in the latency of vanilla. This is because the router re-

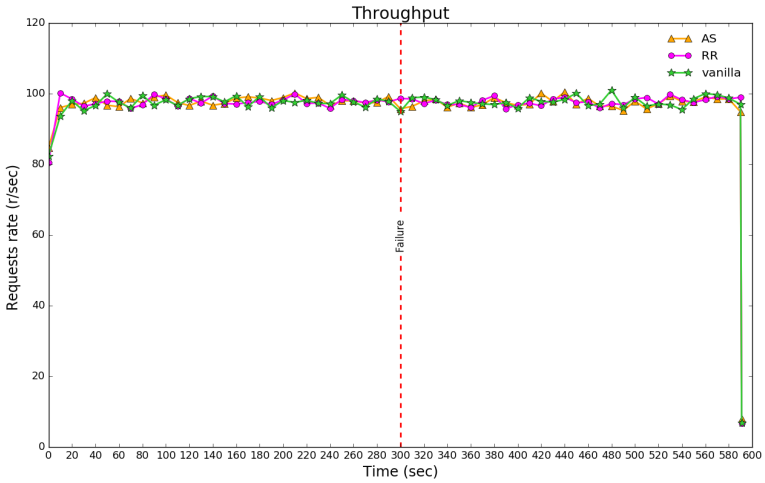


Figure 6.12 – Throughput of Fission vanilla, AS, and RR with pod failure

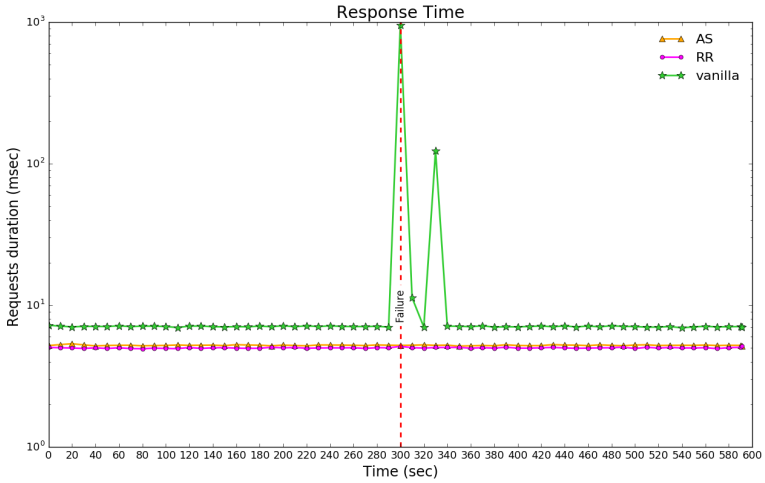


Figure 6.13 – Response time of Fission vanilla, AS, and RR with pod failure

tries many requests, where the wait time is increased exponentially after every attempt. We deduce that the response time of the queued requests is increased when the pod recovers. The response time of AS and RR is stable since the requests are served by the standby pod in AS and by the second replica in RR.

3) **Network Delay Scenario**

Figures 6.16, 6.17, and 6.18 show the response time of Fission vanilla, AS, and

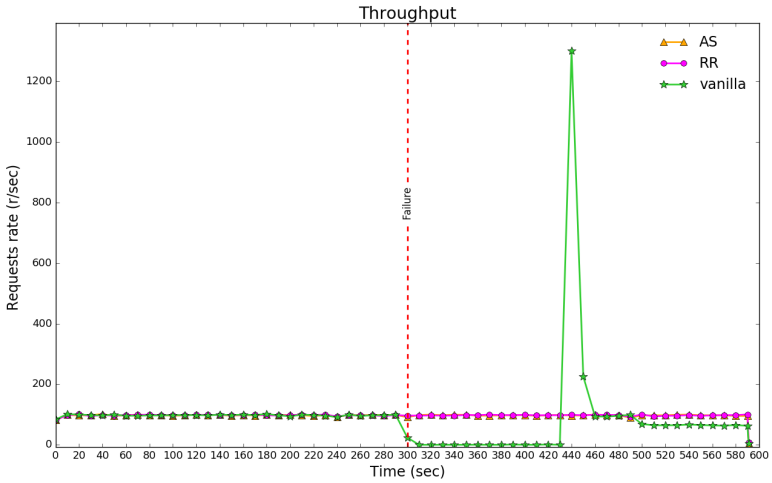


Figure 6.14 – Throughput of Fission vanilla, AS, and RR with node failure

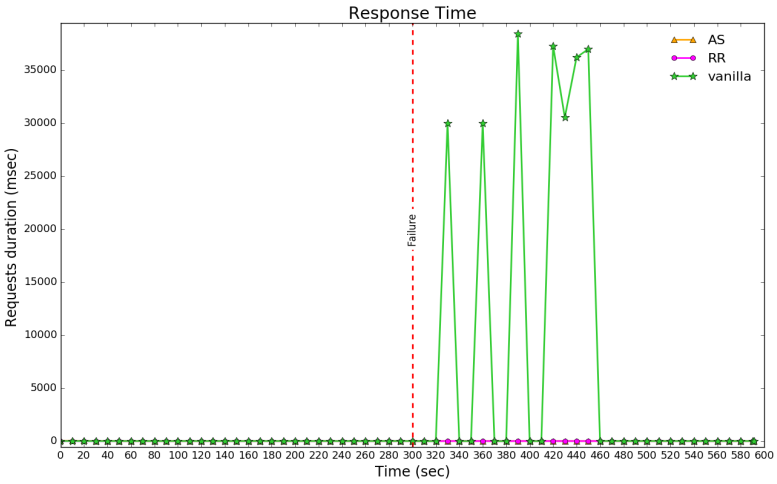


Figure 6.15 – Response time of Fission vanilla, AS, and RR with node failure

RR respectively with the injected latency values. When we increase the value of latency, we can see a significant change in the response time of vanilla. For example, 200ms of latency doubles the response time of vanilla from 500ms to 1000ms (see Figures 6.17 and 6.18). The reason for this behaviour is that the router retries requests with exponential backoff, increasing the waiting time between retries which leads to performance degradation. Looking at the response time of AS, we notice a peak when the latency is added because the active pod responds too late.

In RR, we see no impact on the response time when we add latency on one of the replicas, because the delay of a single replica is masked by the response of the other replica.

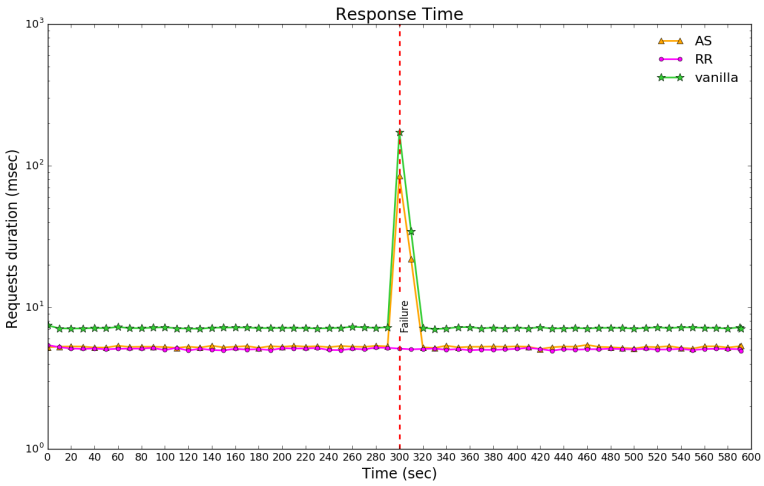


Figure 6.16 – Response time with 50ms of latency

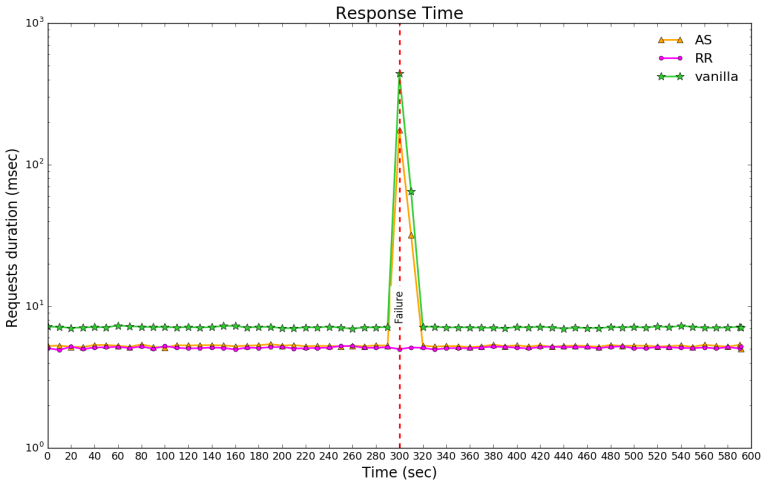


Figure 6.17 – Response time with 100ms of latency

6.3.2 Availability Results

In this section, we present the recovery time and the error rate of Fission vanilla, AS, and RR with pod and node failures.

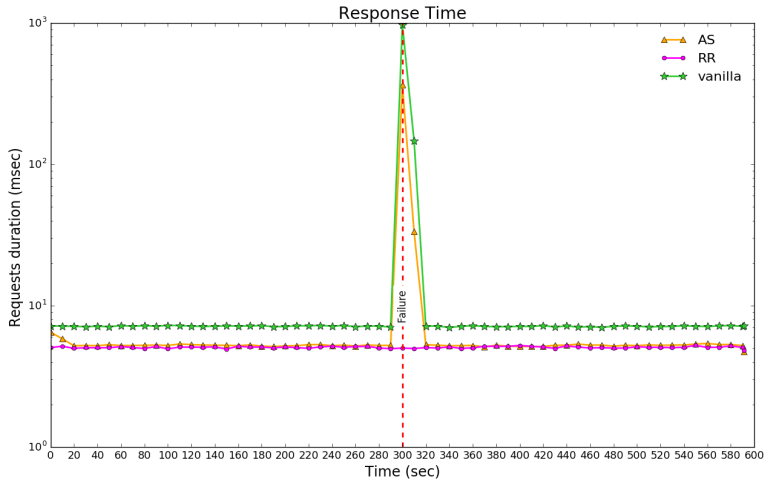


Figure 6.18 – Response time with 200ms of latency

a) **Recovery time**

The recovery time for vanilla is measured the same way as described in Experiment 1 (see Figure 6.7). For AS, the same method as the one described in Experiment 1 is used to measure the recovery time. The only difference is the update of the router cache instead of the CoreDNS cache (see Figure 6.19). For RR, no recovery is necessary as the failure of one of the replicas does not affect service availability (see Figure 6.20). The service remains available because the second pod serves the requests.

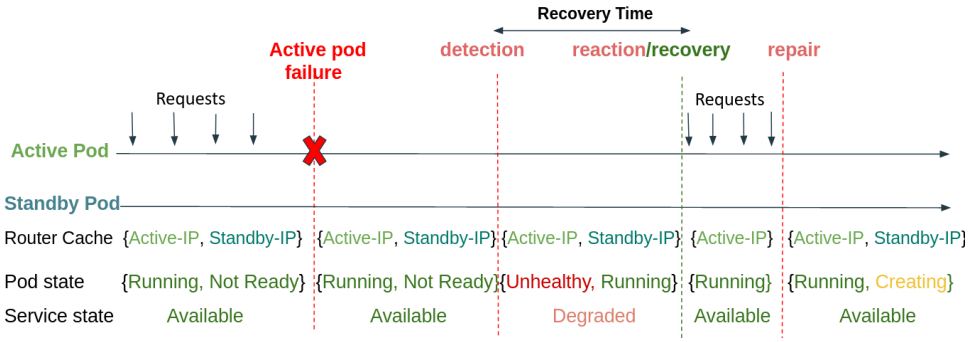


Figure 6.19 – Recovery time in AS

1) **Pod Failure Scenario**

Table 6.3 presents the recovery time of Fission vanilla, AS, and RR with a pod failure. The measured recovery time of AS is significantly lower than the one

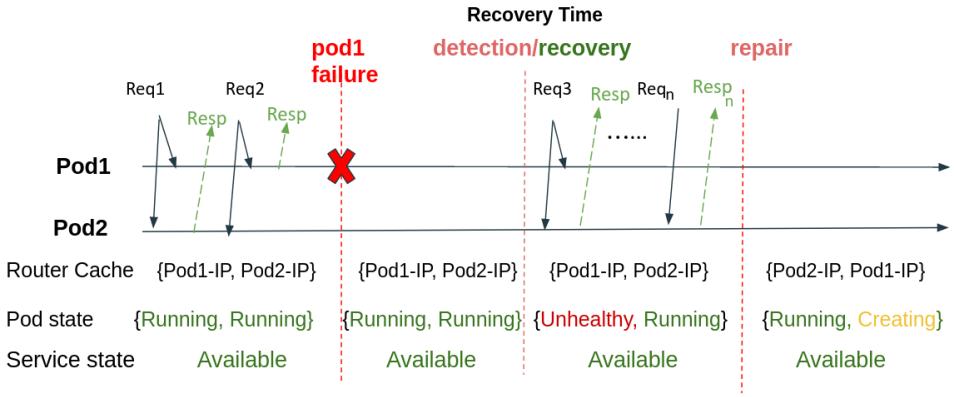


Figure 6.20 – Recovery time in RR

of vanilla. The reason is that with AS, there is already a standby pod, and the service is recovered and becomes active as soon as the standby detects the failure of the active pod. In contrast, the recovery in vanilla depends on the replacement of the failed pod. For RR, the second replica continues to serve requests. Therefore, in this approach, the recovery time is zero.

Table 6.3 – Recovery time of AS, vanilla and RR with pod failure

Fission vanilla	Fission AS	Fission RR
7s	1.81s	0s

2) **Node Failure Scenario**

Table 6.4 presents the recovery time of Fission vanilla, AS, and RR with node failure. The recovery time of vanilla is significantly higher than that of AS and RR for similar reasons for the pod failure scenario. The retry mechanism spends time retrying to submit the request before recreating a new pod. AS takes less than 3 seconds to recover from a node crash, whereas vanilla takes more than 2 minutes. RR recovery time is 0 as there is no unavailability of the service.

b) **Error Rate**

In the following, we present the error rate of Fission vanilla, AS, and RR with pod and node failures. To measure the rate of requests that fail, we count the number of requests that return a HTTP status with a response of 5xx (means the request

Table 6.4 – Recovery time of AS, vanilla and RR with node failure

Fission vanilla	Fission AS	Fission RR
2m19s	2.80s	0s

cannot be fulfilled due to a server error). However, if a request has been successfully handled, the HTTP status code returned in the response is from the 2xx class of status code.

1) **Pod Failure Scenario**

Table 6.5 shows the error rates in Fission vanilla, AS, and RR with pod failures. Vanilla has a 0.01% error rate (i.e., some HTTP requests failed with code 502 bad gateway error) which indicates that the router can't reach the requested pod. The error rate for AS and RR is 0% (i.e., all requests succeeded with a returned code 200).

Table 6.5 – Error rate in Fission vanilla, AS and RR with pod failures

Fission vanilla	Fission AS	Fission RR
0.01%	0%	0%

2) **Node Failure Scenario**

Table 6.6 shows the error rate in Fission vanilla, AS, and RR with a node failure.

In this scenario, request errors occur due to the node crash. In vanilla, the error rate is 1.26%. Once the requests are retried, some of them return a 502 status code. The error rate in AS and RR is 0%, as both tolerate better a node crash because of the presence of a replica. Thus, all requests are served with success and return the code 200.

Table 6.6 – Error rate in Fission vanilla, AS and RR with node failure

Fission vanilla	Fission AS	Fission RR
1.26%	0%	0%

6.3.3 Resource Consumption Analysis

We measured CPU and memory usage in order to analyse the amount of resources required to support fault tolerance in each approach. Figures 6.21 and 6.22 show CPU and memory consumption in Fission vanilla, AS, and RR without and with pod and node failures. The numbers shown are the overall CPU and memory usage of the 5 nodes hosting Kubernetes and the Fission platform during the execution of the workload.

In the three scenarios (i.e., no failure, pod failure, and node failure), we observe that RR consumes more CPU and memory compared to vanilla and AS. When there are no failures, the overhead of using RR is 180% in CPU and 52% in memory consumption compared to vanilla. This is because of the additional resources allocated to the second replica. In vanilla, only one replica executes requests. AS has an overhead of 141% in CPU consumption and 39% in memory consumption compared to vanilla. Note that in AS, the standby replica is hot, which means that it is loaded in memory. The replica does not process requests (like the active replica of RR), but it does perform regular heartbeats, which consumes resources.

Figures 6.23, 6.24, and 6.25 show the average CPU consumption over time separately for each node: the Kubernetes master node, the Fission node, and the 3 worker nodes for all approaches with a pod failure.

The CPU consumption of AS and RR are similar and vanilla shows the lowest CPU utilization. We notice that on average, the coordinator nodes (i.e., master and Fission nodes) need more resources compared to the worker nodes because the services that manage the cluster are located in the master and the services that manage the functions are located in the Fission node. Especially for AS and RR, their coordinator nodes are experiencing high CPU usage compared to vanilla. This is due to the CPU consumption of the Router in the Fission node when calling the Kubernetes API server to get updates on the IP addresses of the function pods.

Regarding the CPU consumption of worker 2 and worker 3 in AS (see Figure 6.24), we note that after the failure, worker 3 starts to consume more CPU while worker 2 consumes less CPU, which shows the behavior of the failover to the standby pod.

In RR, when the pod 1 fails, another one is created in the same node (worker 2) for that reason we observe a short peak in the CPU usage at the 6th minute, as shown in Figure 6.25.

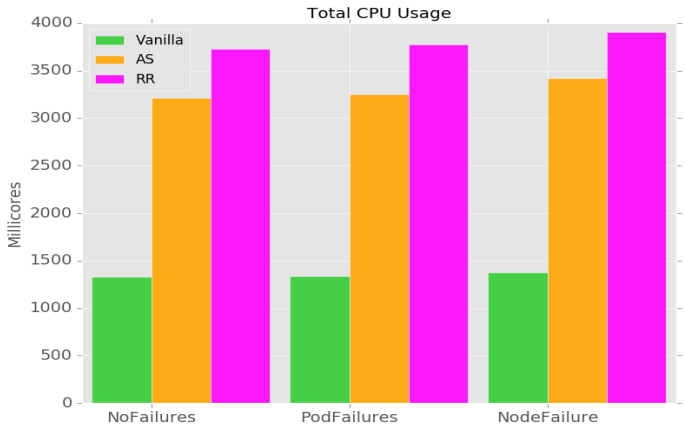


Figure 6.21 – CPU consumption without and with pod and node failures

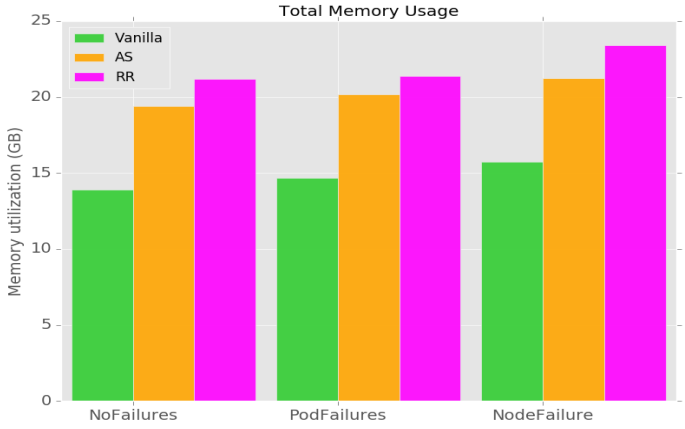


Figure 6.22 – Memory consumption without and with pod and node failures

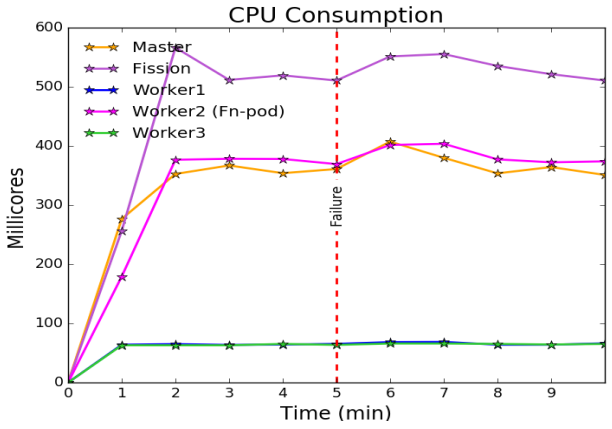


Figure 6.23 – CPU consumption per node in vanilla with pod failure

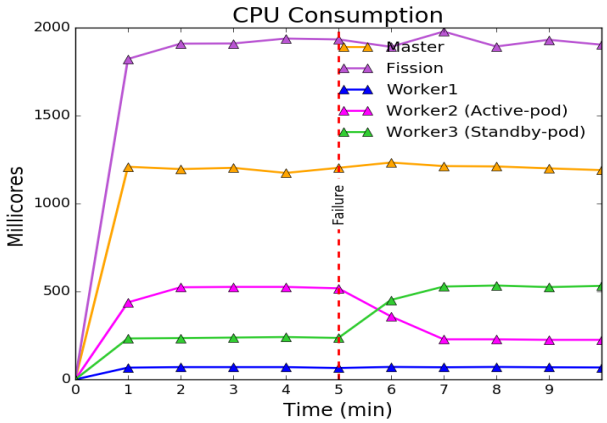


Figure 6.24 – CPU consumption per node in AS with pod failure

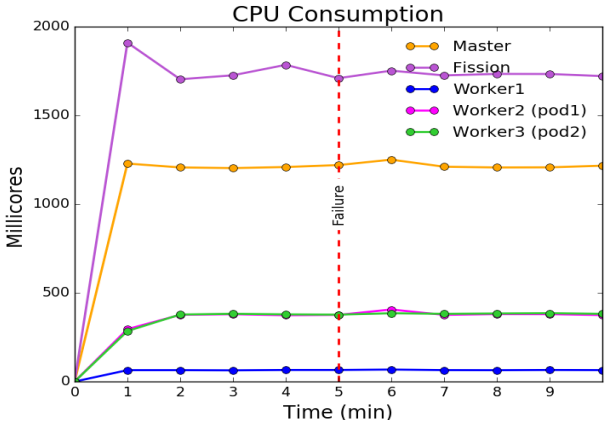


Figure 6.25 – CPU consumption per node in RR with pod failure

6.4 Lessons Learned

From our experimental comparison of the three fault tolerance approaches (i.e., retry, Active-Standby and Request Replication), we note that each approach has different properties and is most effective under different conditions. The retry mechanism is well suited for transient faults that last a short time. This approach consumes less resources than the Active-Standby and the Request Replication approaches. The Active-Standby approach offers better availability (independently of the implementation) in the presence of long-lasting faults, compared to the retry approach, but at the cost of higher resource consumption. For instance, in our experiments, the Active-Standby approach consumes more than two times the CPU consumed by the retry approach. The Request Replication approach offers the best availability for any kind of failure. Indeed, when the fault does not affect all replicas, there is almost no impact on the overall availability. This approach

also offers the best, and most stable performance. On the other hand, the Request Replication approach incurs the highest resource consumption. In general, we observe that availability and resource consumption in the three approaches are inversely related. The recommended type of functions to use with the three approaches is idempotent functions, that is, functions that give the same output when called multiple times with the same input. This type of function is useful to prevent inconsistent, duplicated, and lost data in the application.

6.5 Summary

In this chapter, we presented an experimental study to evaluate three fault tolerance approaches (i.e., retry, Active-Standby and Request Replication) implemented in Fission FaaS framework. The experiments were carried out using several failures scenarios and metrics (performance, availability, and resource consumption) in order to evaluate the ability of each approach to tolerate faults. The obtained results show that the Request Replication approach performs better in the three sets of failure scenarios (pod failure, node failure and network delay scenarios) pointing to its effectiveness. This approach is able to ensure a high availability of functions, compared to the Active-Standby and retry approaches. However, Request Replication requires a significant amount of extra resources to achieve fault-tolerant FaaS applications.

The next chapter concludes this thesis providing an outlook of future research directions in the field.

Chapter 7

Conclusion and Perspectives

In this chapter, we conclude the presented work and give an overview of some future research directions.

7.1 Conclusion

Function as a Service offers many benefits to its users, including the facility of deploying applications without having to manage servers. It also provides an attractive pay-per-use pricing model. As a result, an increasing number of customers are rapidly adopting the FaaS model. However, an important aspect of the FaaS environment, namely fault tolerance, requires a deeper study. Since fault tolerance is handled by the FaaS provider, the provider has to ensure high availability for the deployed functions. Indeed, most FaaS systems support a basic form of fault tolerance through retrying function executions. This makes them lack the ability to tolerate failures of different types, such as permanent failures. For example, if a function execution fails due to an infrastructure failure, then the function request would be pointlessly retried many times. In addition, the recovery of FaaS systems after such a failure was not covered in the literature and was thus addressed in our work.

In our work, we investigated the current fault tolerance mechanisms used in FaaS systems; we found that there is a limited support for fault tolerance and most platforms do not provide any mechanism to tolerate permanent faults such as node failures. This is an important limitation for attaining high availability of functions. Therefore, we studied the existing open source FaaS frameworks in order to select the appropriate one for hosting our approaches, based on a defined set of criteria, including extensibility, popularity, documentation and community size. We selected Fission for our work because it is flexible and allows modifying the source code. Fission uses the retry fault tolerance mechanism, which consists basically in restarting the entire submission process of a failed request and that we compared with our approaches.

The next step was to study the applicability of existing replication-based strategies in FaaS systems. Our first decision was to use a passive replication approach (Active-Standby) that provides a standby replica to take over when the active replica fails. However, this approach has a slow reaction to failures because when the active replica fails, the failure must be detected by the standby replica. In order to make the functions available and the failures masked to users, it is desirable to provide fault tolerance mechanisms that allow the functions to continue executing requests with no interruption in the event of failures. For doing so, we proposed another approach also based on replication but this time on an active mode (Request Replication), where the requests are executed by all the function replicas.

The proposed fault tolerance approaches were presented in Chapter 5, as well as an overview of their integration in Fission. Based on our implementation of the two replication approaches in Fission, we experimentally compared them with the basic retry mechanism natively implemented in Fission, in terms of different metrics, and under different failure scenarios.

The obtained results highlighted the differences among the three approaches, especially in terms of their reaction to failures. Notably, they showed that the retry mechanism is not sufficient for providing high availability of functions. The reason is that the default behavior of the retry mechanism results in a high recovery time in the event of node failures. The retry mechanism is better suited for transient failures as seen in the network delay scenario. With Active-Standby (AS), the recovery time is decreased because the service becomes available shortly after the standby replica detects the failure of the active replica. With Request-Replication (RR), the service always remains available as long as another replica continues to respond to users and recovery does not depend on replacing the faulty replica. Both replication approaches use more resources than the retry mechanism.

Based on the previous discussion, each of the fault tolerance approaches has its own advantages and disadvantages and can be employed for specific fault tolerance scenarios. We can see that choosing the appropriate fault tolerance approach in the context of FaaS depends necessarily on the requirements that must be addressed. If the emphasis is on good performance (e.g., for latency sensitive applications), the preferred approach is RR; if it is on reduced resources (e.g., for resource-constrained environments, such as edge) and need for high availability, the preferred approach is AS. Retry can be used when the requirement is for resource saving and limited availability.

7.2 Perspectives

There are many possibilities to enrich our research in various aspects. In this section, we suggest future work directions to extend the work presented in this thesis.

7.2.1 Short-Term Perspectives

Chapter 6 covered the resource consumption measurements of the proposed approaches and the retry mechanism. However, energy consumption has not been measured in this work. It would be interesting to measure how much energy the approaches need to provide fault tolerance for FaaS functions. Therefore, an evaluation in terms of energy consumption is a recommendation for future work that could be done by using the Kwollect framework [114] included in the Grid'5000 platform.

In our experiments, the three approaches were tested with a computational application that computes a Fibonacci sequence. Nevertheless, to strengthen the value of this work, it is important to test the presented approaches using stateless applications from different domains (e.g., IoT, video processing, and machine learning). In this way, further flaws of each approach could be identified and addressed.

Another avenue for future work is related to cold starts. With cold start being a frequently discussed topic among FaaS challenges [128, 135], it is of interest to deeply investigate what is the proper way to mitigate cold starts in the event of failures. If a function fails, subsequent invocations may lead to cold starts, which delay execution. Thus, a further measurement of cold start after failures to determine the best fault tolerance approach could serve as a useful contribution to the topic.

7.2.2 Mid-Term Perspectives

Another direction for future work is related to how the FaaS model can be applied in the context of edge computing while ensuring the availability of functions. Edge computing is a distributed architecture that enables computation to be as close as possible to end users. This allows users to benefit from faster response times. With FaaS, there is no need for developers to maintain servers or instances, which provides efficient resource usage and associated cost savings. These benefits could make the FaaS model well-suited for adoption in edge computing. For instance, in [102], [22] the authors found that FaaS would be particularly beneficial for edge computing and enables developers to use edge

resources with less complexity and effort. However, the fault tolerance of FaaS functions in edge computing environments is still an open issue. More research is required to adapt the fault tolerance techniques of FaaS platforms to the edge context and thereby provide an effective solution to improve the fault tolerance of functions deployed in the edge.

7.2.3 Long-Term Perspectives

In our work, we considered stateless and idempotent applications, where the same input always gives the same output. To satisfy further use cases, FaaS architectures and offerings are recently evolving to address state. Therefore, it will be highly desirable to design fault tolerance approaches for stateful FaaS applications. With stateful applications, a state is typically maintained in external storage services, such as NoSQL databases [200]. Using the Request Replication approach for such applications seems challenging. The reason is that concurrent accesses increase the load on the storage service and introduce overhead for maintaining consistency. This may result in reduced performance in the case of normal, fault-free operation compared to using the Active-Standby or retry approaches. Integrating caching into stateful functions could mitigate this problem [174].

With this thesis's focus on different fault tolerance approaches, a natural extension would be to design a fault tolerant system for FaaS that simultaneously supports multiple approaches, such as retry, Active-Standby and Request Replication, and uses one or another according to specific factors while meeting users's requirements (e.g., performance, availability, resource consumption). These factors may include application types (e.g., stateful or stateless) and operating conditions (e.g., fault rates, network latencies).

In this thesis, the proposed approaches are both based on the replication. Hence, it would be interesting to extend the work presented in this thesis by further investigation of other fault tolerance strategies that are available in the literature and have also been mentioned in Chapter 2, such as checkpointing. This technique is typically used to tolerate failures by periodically saving an application's state, known as checkpoint, to be used in the presence of a failure. In the context of FaaS, some works [202] and [105] propose the usage of checkpointing to restart functions from where they timed out, which is useful for functions that execute long-running computations. It would be useful to study the applicability of checkpointing to typical FaaS functions, the majority of which currently run for under one minute [64]. One interesting idea could be implementing the checkpointing mechanism in FaaS environments with an adaptive checkpoint interval that uses the timings of already occurred failures to estimate the occurrences of the next ones [37].

BIBLIOGRAPHY

[1] *Building a Multi-region Serverless Application with Amazon API Gateway and AWS Lambda*, https://aws.amazon.com/blogs/compute/building-a-multi-region-serverless-application-with-amazon-api-gateway-and-aws-lambda/, [Online; accessed 10-january-2022].

[2] *Implementing Multi-Region Disaster Recovery Using Event-Driven Architecture*, https://aws.amazon.com/blogs/architecture/implementing-multi-region-disaster-recovery-using-event-driven-architecture/, [Online; accessed 10-january-2022].

[3] *Alibaba Functions*, https://www.alibabacloud.com/fr/product/function-compute, [Online; accessed 07-november-2021].

[4] *Amazon DynamoDB*, https://aws.amazon.com/fr/dynamodb/, [Online; accessed 04-november-2021].

[5] *Amazon S3*, https://aws.amazon.com/fr/s3/, [Online; accessed 04-november-2021].

[6] *Amazon SageMaker*, https://aws.amazon.com/pm/sagemaker/, [Online; accessed 8-january-2022].

[7] *Amazon Serverless ML Training*, https://aws.amazon.com/fr/blogs/machine-learning/code-free-machine-learning-automl-with-autogluon-amazon-sagemaker-and-aws-lambda/, [Online; accessed 07-november-2021].

[8] *Amazon SNS*, https://aws.amazon.com/sns/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=desc, [Online; accessed 4-january-2022].

[9] *Error Handling and Automatic Retries in AWS Lambda*, https://docs.aws.amazon.com/lambda/latest/dg/invocation-retries.html, [Online; accessed 07-july-2021], 2020.

[10] *AWS Lambda Features*, https://aws.amazon.com/lambda/features/, [Online; accessed 07-july-2021], 2020.

[11] *AWS S3*, https://https://aws.amazon.com/fr/s3/, [Online; accessed 19-october-2021].

[12] Zeeshan Amin, Harshpreet Singh, and Nisha Sethi, « Review on fault tolerance techniques in cloud computing », *in*: *International Journal of Computer Applications* 116.*18* (2015).

[13] Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter, « Sprocket: A serverless video processing framework », *in*: *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 263–274.

[14] *Apache*, https://httpd.apache.org/docs/2.4/programs/ab.html, [Online; accessed 29-september-2021].

[15] *Apache Flink StateFu*, https://flink.apache.org/stateful-functions.html, [Online; accessed 14-november-2021].

[16] *Apache Kafka*, https://kafka.apache.org/, [Online; accessed 4-january-2022].

[17] *Apache Mesos*, http://mesos.apache.org/, [Online; accessed 29-november-2021].

[18] *Apache OpenWhisk*, https://openwhisk.apache.org, [Online; accessed 07-july-2021].

[19] *Apache OpenWhisk*, https://openwhisk.apache.org/, [Online; accessed 07-november-2021].

[20] *Apache OpenWhisk Composer*, https://github.com/apache/openwhisk-composer, [Online; accessed 07-july-2021].

[21] *Aqua*, https://www.aquasec.com/aqua-cloud-native-security-platform/, [Online; accessed 02-november-2021].

[22] Mohammad S Aslanpour et al., « Serverless edge computing: vision and challenges », *in*: *2021 Australasian Computer Science Week Multiconference*, 2021, pp. 1–10.

[23] *AutoGluon*, https://auto.gluon.ai/stable/index.html, [Online; accessed 8-january-2022].

[24] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr, « Basic concepts and taxonomy of dependable and secure computing », *in*: *IEEE transactions on dependable and secure computing* 1.*1* (2004), pp. 11–33.

[25] *Using AWS Serverless Technology as an Enabler for Cloud Adoption*, https://aws.amazon.com/blogs/apn/using-aws-serverless-technology-as-an-enabler-for-cloud-adoption/, [Online; accessed 07-july-2021], 2019.

[26] *Azure Functions geo-disaster recovery*, https://docs.microsoft.com/en-us/azure/azure-functions/functions-geo-disaster-recovery, [Online; accessed 16-november-2021], 2020.

[27] *AWS Step Functions*, https://aws.amazon.com/step-functions, [Online; accessed 07-july-2021].

[28] *Azure Blob Storage*, https://azure.microsoft.com/en-us/services/storage/blobs/, [Online; accessed 04-november-2021].

[29] *Azure Queue Storage*, https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-queue, [Online; accessed 8-january-2022].

[30] *Azure SQL Database*, https://azure.microsoft.com/en-us/products/azure-sql/database/, [Online; accessed 04-november-2021].

[31] *Azure Functions Timeout*, https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale#timeout, [Online; accessed 19-october-2021].

[32] *Azure Durable Functions*, https://docs.microsoft.com/en-us/azure/azure-functions/durable, [Online; accessed 07-july-2021].

[33] *Azure Functions*, https://azure.microsoft.com/fr-fr/services/functions/, [Online; accessed 07-july-2021], 2020.

[34] Timon Back and Vasilios Andrikopoulos, « Using a microbenchmark to compare function as a service solutions », *in*: *European Conference on Service-Oriented and Cloud Computing*, Springer, 2018, pp. 146–160.

[35] Ioana Baldini, Perry Cheng, Stephen J Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu, « The serverless trilemma: Function composition for serverless computing », *in*: *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2017, pp. 89–103.

[36] David Balla, Markosz Maliosz, and Csaba Simon, « Open Source FaaS Performance Aspects », *in*: *2020 43rd International Conference on Telecommunications and Signal Processing (TSP)*, IEEE, 2020, pp. 358–364.

[37] Mohamad Imran bin Bandan, Subhasis Bhattacharjee, Dhiraj K Pradhan, and Jimson Mathew, « Adaptive checkpoint interval algorithm considering task deadline and lifetime reliability for real-time system », *in*: *Procedia Computer Science* 70 (2015), pp. 821–828.

[38] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López, « On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures », *in*: *Proceedings of the 20th International Middleware Conference*, Middleware '19, Davis, CA, USA: Association for Computing Machinery, 2019, pp. 41–54, ISBN: 9781450370097, DOI: 10.1145/3361525.3361535, URL: https://doi.org/10.1145/3361525.3361535.

[39] Yasmina Bouizem, Djawida Dib, Nikos Parlavantzas, and Christine Morin, « Active-Standby for High-Availability in FaaS », *in*: *Sixth International Workshop on Serverless Computing (WoSC6) 2020*, Delft, Netherlands, Dec. 2020, DOI: 10.1145/3429880.3430097, URL: https://hal.inria.fr/hal-03043479.

[40] Abderraouf Boussif, Mohamed Ghazel, and Joao Carlos Basilio, « Intermittent fault diagnosability of discrete event systems: an overview of automaton-based approaches », *in*: *Discrete Event Dynamic Systems* 31.*1* (2021), pp. 59–102.

[41] Dario Bruneo, Salvatore Distefano, Francesco Longo, Antonio Puliafito, and Marco Scarpa, « Workload-based software rejuvenation in cloud systems », *in*: *IEEE Transactions on Computers* 62.*6* (2013), pp. 1072–1085.

[42] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo, « SEUSS: skip redundant paths to make serverless fast », *in*: *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–15.

[43] Jiajun Cao, Matthieu Simonin, Gene Cooperman, and Christine Morin, « Checkpointing as a service in heterogeneous cloud environments », *in*: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, IEEE, 2015, pp. 61–70.

[44] John Carnell and Illary Sanchez, *Spring microservices in action*, Simon and Schuster, 2021.

[45] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng, « Wukong: A scalable and locality-enhanced framework for serverless paral-

lel computing », *in*: *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 1–15.

[46] *chaos-mesh*, `https://github.com/chaos-mesh/chaos-mesh`, [Online; accessed 5-may-2021].

[47] Ryan Chard, Tyler J Skluzacek, Zhuozhao Li, Yadu Babuji, Anna Woodard, Ben Blaiszik, Steven Tuecke, Ian Foster, and Kyle Chard, « Serverless supercomputing: High performance function as a service for science », *in*: *arXiv preprint arXiv:1908.04907* (2019).

[48] 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)*, IEEE, 2019, pp. 28–35.

[49] Mehdi Nazari Cheraghlou, Ahmad Khadem-Zadeh, and Majid Haghparast, « A survey of fault tolerance architecture in cloud computing », *in*: *Journal of Network and Computer Applications* 61 (2016), pp. 81–92.

[50] Susanta Nanda Tzi-cker Chiueh and Stony Brook, « A survey on virtualization technologies », *in*: *Rpe Report* 142 (2005).

[51] *Retry pattern*, `https://docs.microsoft.com/en-us/azure/architecture/patterns/retry`, [Online; accessed 07-july-2021], 2020.

[52] *Cloud Functions*, `https://cloud.google.com/functions`, [Online; accessed 04-november-2021].

[53] *CloudEvents*, `https://cloudevents.io/`, [Online; accessed 4-january-2022].

[54] Mikhail Shilkov, *Comparison of Cold Starts in Serverless Functions across AWS, Azure, and GCP*, `https://mikhail.io/serverless/coldstarts/big3/`, [Online; accessed 29-october-2021].

[55] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler, « Sebs: A serverless benchmark suite for function-as-a-service computing », *in*: *Proceedings of the 22nd International Middleware Conference*, 2021, pp. 64–78.

[56] *CoreDNS: DNS and Service Discovery*, `https://coredns.io/`, [Online; accessed 20-march-2022].

[57] *Running Automated Tasks with a CronJob*, https://kubernetes.io/docs/tasks/job/automated-tasks-with-cron-jobs/, [Online; accessed 07-october-2021].

[58] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates, « Valve: Securing function workflows on serverless computing platforms », *in*: *Proceedings of The Web Conference 2020*, 2020, pp. 939–950.

[59] *Docker*, https://www.docker.com/, [Online; accessed 26-november-2021].

[60] *Docker Swarm*, https://docs.docker.com/engine/swarm/, [Online; accessed 29-november-2021].

[61] *Many-faced threats to Serverless security*, https://hackernoon.com/many-faced-threats-to-serverless-security-519e94d19dba, [Online; accessed 02-november-2021].

[62] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen, « Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting », *in*: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 467–481.

[63] Elena Dubrova et al., « Fault tolerant design: An introduction », *in*: *Department of Microelectronics and Information Technology, Royal Institute of Technology, Stockholm, Sweden* (2008), pp. 22–3.

[64] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina Abad, and Alexandru Iosup, « The State of Serverless Applications: Collection, Characterization, and Community Consensus », *in*: *IEEE Transactions on Software Engineering* (2021).

[65] *Epsagon*, https://epsagon.com/, [Online; accessed 02-november-2021].

[66] *Event Injection: Protecting your Serverless Applications*, https://www.jeremydaly.com/event-injection-protecting-your-serverless-applications/, [Online; accessed 02-november-2021].

[67] *Faas-flow*, https://github.com/s8sg/faas-flow, [Online; accessed 07-july-2021].

[68] Pascal Felber, Xavier Défago, Patrick Eugster, and André Schiper, « Replicating CORBA objects: a marriage between active and passive replication », *in*: *IFIP International Conference on Distributed Applications and Interoperable Systems*, Springer, 1999, pp. 375–387.

[69] Pascal Felber and Priya Narasimhan, « Experiences, strategies, and challenges in building fault-tolerant CORBA systems », *in*: *IEEE transactions on Computers* 53.*5* (2004), pp. 497–511.

[70] Kamil Figiela, Adam Gajek, Adam Zima, Beata Obrok, and Maciej Malawski, « Performance evaluation of heterogeneous cloud functions », *in*: *Concurrency and Computation: Practice and Experience* 30.*23* (2018), e4792.

[71] *Firebase*, `https://firebase.google.com/`, [Online; accessed 04-november-2021].

[72] *Firecracker Snapshotting*, `https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/snapshot-support.md`, [Online; accessed 29-october-2021].

[73] *Fission*, `https://docs.fission.io/docs/`, [Online; accessed 07-july-2021], 2019.

[74] *Fission*, `https://github.com/fission/fission/tree/master`, [Online; accessed 04-october-2021].

[75] *Fission Router*, `https://godoc.org/github.com/fission/fission/pkg/router`, [Online; accessed 07-july-2021], 2020.

[76] *Fission Workflows*, `https://github.com/fission/fission-workflows`, [Online; accessed 30-september-2021].

[77] *Google Cloud Quotas*, `https://cloud.google.com/functions/quotas`, [Online; accessed 19-october-2021].

[78] Debanjan Ghosh, Raj Sharman, H Raghav Rao, and Shambhu Upadhyaya, « Self-healing systems—survey and synthesis », *in*: *Decision support systems* 42.*4* (2007), pp. 2164–2185.

[79] Mohit Kumar Gokhroo, Mahesh Chandra Govil, and Emmanuel S Pilli, « Detecting and mitigating faults in cloud computing environment », *in*: *2017 3rd International Conference on Computational Intelligence & Communication Technology (CICT)*, IEEE, 2017, pp. 1–9.

[80] Muhammed Golec, Ridvan Ozturac, Zahra Pooranian, Sukhpal Singh Gill, and Rajkumar Buyya, « iFaaSBus: A Security and Privacy based Lightweight Framework for Serverless Computing using IoT and Machine Learning », *in*: *IEEE Transactions on Industrial Informatics* (2021).

[81] *Retrying Background Functions*, `https://cloud.google.com/functions/docs/bestpractices/retries`, [Online; accessed 07-july-2021], 2019.

[82] *Google Cloud Workflows*, `https://cloud.google.com/workflows`, [Online; accessed 07-july-2021].

[83] *Grafana*, `https://grafana.com/`, [Online; accessed 31-january-2022].

[84] Martin Grambow, Tobias Pfandzelter, Luk Burchard, Carsten Schubert, Max Zhao, and David Bermbach, « Befaas: An application-centric benchmarking framework for faas platforms », *in*: *2021 IEEE International Conference on Cloud Engineering (IC2E)*, IEEE, 2021, pp. 1–8.

[85] *Grid5000*, `https://www.grid5000.fr/w/Grid5000:Home`, [Online; accessed 07-july-2021], 2020.

[86] *Guestbook Application*, `https://github.com/fission/examples/tree/master/python/guestbook`, [Online; accessed 30-december-2021].

[87] *gVisor*, `https://gvisor.dev/`, [Online; accessed 04-november-2021].

[88] Luis Felipe Herrera-Quintero, Julian Camilo Vega-Alfonso, Klaus Bodo Albert Banse, and Eduardo Carrillo Zambrano, « Smart its sensor for the transportation planning based on iot approaches using serverless and microservices architecture », *in*: *IEEE Intelligent Transportation Systems Magazine* 10.*2* (2018), pp. 17–27.

[89] Martijn de Heus, Kyriakos Psarakis, Marios Fragkoulis, and Asterios Katsifodimos, « Distributed transactions on serverless stateful functions », *in*: *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*, 2021, pp. 31–42.

[90] Kelsey Hightower, Brendan Burns, and Joe Beda, *Kubernetes: up and running: dive into the future of infrastructure*, " O'Reilly Media, Inc.", 2017.

[91] *History Table*, `https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-orchestrations?tabs=csharp#history-table`, [Online; accessed 14-november-2021].

[92] Michael Hogan, Fang Liu, Annie Sokol, and Jin Tong, « Nist cloud computing standards roadmap », *in*: *NIST Special Publication* 35 (2011), pp. 6–11.

[93] *Hypervisor*, `https://www.geeksforgeeks.org/hypervisor/`, [Online; accessed 26-november-2021].

[94] *IBM Cloud Functions*, `https://www.ibm.com/cloud/functions`, [Online; accessed 07-november-2021].

[95] *IBM Composer*, `https://github.com/ibm-functions/composer`, [Online; accessed 01-november-2021].

[96] *IFTTT*, `https://ifttt.com/`, [Online; accessed 4-january-2022].

[97] *Istio*, `https://istio.io/`, [Online; accessed 04-november-2021].

[98] David Jackson and Gary Clynch, « An investigation of the impact of language runtime on the performance and cost of serverless functions », *in*: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, IEEE, 2018, pp. 154–160.

[99] Amir Javadpour, Sanaz Kazemi Abharian, and Guojun Wang, « Feature selection and intrusion detection in cloud environment based on machine learning algorithms », *in*: *2017 IEEE international symposium on parallel and distributed processing with applications and 2017 IEEE international conference on ubiquitous computing and communications (ISPA/IUCC)*, IEEE, 2017, pp. 1417–1421.

[100] Deepak Sirone Jegan, Liang Wang, Siddhant Bhagat, Thomas Ristenpart, and Michael Swift, « Guarding Serverless Applications with SecLambda », *in*: *arXiv preprint arXiv:2011.05322* (2020).

[101] Zhipeng Jia and Emmett Witchel, « Boki: Stateful Serverless Computing with Shared Logs », *in*: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, 2021, pp. 691–707.

[102] Runyu Jin, Qirui Yang, and Ming Zhao, « Is faas suitable for edge computing », *in*: *USENIX Association, June* (2020).

[103] Eric Jonas et al., « Cloud programming simplified: A berkeley view on serverless computing », *in*: *arXiv preprint arXiv:1902.03383* (2019).

[104] Shannon Joyner, Michael MacCoss, Christina Delimitrou, and Hakim Weatherspoon, « Ripple: A Practical Declarative Programming Framework for Serverless Compute », *in*: *arXiv:2001.00222 [cs.DC]*, Jan. 2020.

[105] Pekka Karhula, Jan Janak, and Henning Schulzrinne, « Checkpointing and Migration of IoT Edge Functions », *in*: *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '19, Dresden, Germany: Association for Computing Machinery, 2019, pp. 60–65, ISBN: 9781450362757, DOI: 10.1145/3301418.3313947, URL: https://doi.org/10.1145/3301418.3313947.

[106] B. W. Kernighan and M. D. McIlroy, *UNIX Time-Sharing System: UNIX Programmer's Manual (7th ed.)* Bell Telephone Laboratories, 1979.

[107] Jeongchul Kim and Kyungyong Lee, « Functionbench: A suite of workloads for serverless cloud function service », *in*: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, IEEE, 2019, pp. 502–504.

[108] *Knative*, https://knative.dev/docs/, [Online; accessed 04-november-2021].

[109] *Kubeless*, https://github.com/kubeless/kubeless, [Online; accessed 04-october-2021].

[110] *Kubeless*, https://kubeless.io/, [Online; accessed 07-october-2021].

[111] *Kubernetes*, https://kubernetes.io/, [Online; accessed 5-may-2021].

[112] *Configure Liveness, Readiness and Startup Probes*, https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/, [Online; accessed 07-july-2021].

[113] Priti Kumari and Parmeet Kaur, « A survey of fault tolerance in cloud computing », *in*: *Journal of King Saud University-Computer and Information Sciences* 33.*10* (2021), pp. 1159–1176.

[114] *Kwollect*, https://gitlab.inria.fr/grid5000/kwollect, [Online; accessed 22-febuary-2022].

[115] *Lambda Quotas*, https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html, [Online; accessed 19-october-2021].

[116] *Lambda@Edge*, https://aws.amazon.com/fr/lambda/edge/, [Online; accessed 04-november-2021].

[117] Hyungro Lee, Kumar Satyam, and Geoffrey Fox, « Evaluation of production serverless computing environments », *in*: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, IEEE, 2018, pp. 442–450.

[118] Yen-Lin Lee, Deron Liang, and Wei-Jen Wang, « Optimal Online Liveness Fault Detection for Multilayer Cloud Computing Systems », *in*: *IEEE Transactions on Dependable and Secure Computing* (2021).

[119] Philipp Leitner, Erik Wittern, Josef Spillner, and Waldemar Hummer, « A mixed-method empirical study of Function-as-a-Service software development in industrial practice », *in*: *Journal of Systems and Software* 149 (2019), pp. 340–359.

[120] Junfeng Li, Sameer G Kulkarni, KK Ramakrishnan, and Dan Li, « Analyzing Open-Source Serverless Platforms: Characteristics and Performance », *in*: *arXiv preprint arXiv:2106.03601* (2021).

[121] Junfeng Li, Sameer G Kulkarni, KK Ramakrishnan, and Dan Li, « Understanding open source serverless platforms: Design considerations and performance », *in*: *Proceedings of the 5th International Workshop on Serverless Computing*, 2019, pp. 37–42.

[122] Ping-Min Lin and Alex Glikson, « Mitigating cold starts in serverless platforms: A pool-based approach », *in*: *arXiv preprint arXiv:1903.12221* (2019).

[123] *LXC: Linux container*, `https://linuxcontainers.org/`, [Online; accessed 26-november-2021].

[124] Theo Lynn, Pierangelo Rosati, Arnaud Lejeune, and Vincent Emeakaroha, « A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms », *in*: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, 2017, pp. 162–169.

[125] Iran Mahallat, « Fault-tolerance techniques in cloud storage: a survey », *in*: *Int J Database Theory Appl* 8.*4* (2015), pp. 183–190.

[126] 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*, 2020, pp. 73–84.

[127] Maciej Malawski, Adam Gajek, Adam Zima, Bartosz Balis, and Kamil Figiela, « Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions », *in*: *Future Generation Computer Systems* 110 (2020), pp. 502–514.

[128] Johannes Manner, Martin Endreß, Tobias Heckel, and Guido Wirtz, « Cold start influencing factors in function as a service », *in*: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, IEEE, 2018, pp. 181–188.

[129] Johannes Manner, Stefan Kolb, and Guido Wirtz, « Troubleshooting serverless functions: a combined monitoring and debugging approach », *in*: *SICS Software-Intensive Cyber-Physical Systems* 34.*2* (2019), pp. 99–104.

[130] *Marathon*, `https://mesosphere.github.io/marathon/`, [Online; accessed 28-febuary-2022].

[131] Garrett McGrath and Paul R Brenner, « Serverless computing: Design, implementation, and performance », *in*: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, IEEE, 2017, pp. 405–410.

[132] Peter Mell, Tim Grance, et al., « The NIST definition of cloud computing », *in*: (2011).

[133] Richard A. Meyer and Love H. Seawright, « A virtual machine time-sharing system », *in*: *IBM Systems Journal* 9.*3* (1970), pp. 199–218.

[134] *Minio*, `https://min.io/`, [Online; accessed 4-january-2022].

[135] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov, « Agile cold starts for scalable serverless », *in*: *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[136] Sunil Kumar Mohanty, Gopika Premsankar, Mario Di Francesco, et al., « An Evaluation of Open Source Serverless Computing Frameworks. », *in*: *CloudCom*, 2018, pp. 115–120.

[137] Roberto Morabito, « Power consumption of virtualization technologies: an empirical investigation », *in*: *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, IEEE, 2015, pp. 522–527.

[138] Roberto Morabito, Jimmy Kjällman, and Miika Komu, « Hypervisors vs. lightweight virtualization: a performance comparison », *in*: *2015 IEEE International Conference on Cloud Engineering*, IEEE, 2015, pp. 386–393.

[139] Mukosi Abraham Mukwevho and Turgay Celik, « Toward a Smart Cloud: A Review of Fault-Tolerance Methods in Cloud Systems », *in*: *IEEE Transactions on Services Computing* 14.*2* (2021), pp. 589–605, DOI: 10.1109/TSC.2018.2816644.

[140] Mukosi Abraham Mukwevho and Turgay Celik, « Toward a smart cloud: A review of fault-tolerance methods in cloud systems », *in*: *IEEE Transactions on Services Computing* 14.*2* (2018), pp. 589–605.

[141] *Nginx*, https://www.nginx.com/, [Online; accessed 24-october-2021].

[142] *Asynchronous Functions*, https://docs.openfaas.com/reference/async/, [Online; accessed 05-january-2022], 2021.

[143] *OpenFaaS*, https://www.openfaas.com, [Online; accessed 08-october-2021].

[144] *faas-netes*, https://github.com/openfaas/faas-netes, [Online; accessed 28-september-2021].

[145] *OpenFaas*, https://github.com/openfaas/faas, [Online; accessed 04-october-2021].

[146] *Oracle Functions*, https://docs.oracle.com/en-us/iaas/Content/Functions/home.htm, [Online; accessed 07-november-2021].

[147] Andrei Palade, Aqeel Kazmi, and Siobhán Clarke, « An evaluation of open source serverless computing frameworks support at the edge », *in*: *2019 IEEE World Congress on Services (SERVICES)*, vol. 2642, IEEE, 2019, pp. 206–211.

[148] Prasenjit Kumar Patra, Harshpreet Singh, and Gurpreet Singh, « Fault tolerance techniques and comparative implementation in cloud computing », *in*: *International Journal of Computer Applications* 64.*14* (2013).

[149] Roland Pellegrini, Igor Ivkic, and Markus Tauber, « Function-as-a-service benchmarking framework », *in*: *arXiv preprint arXiv:1905.11707* (2019).

[150] Per Persson and Ola Angelsmark, « Kappa: serverless iot deployment », *in*: *Proceedings of the 2nd International Workshop on Serverless Computing*, 2017, pp. 16–21.

[151] Tobias Pfandzelter and David Bermbach, « tinyfaas: A lightweight faas platform for edge environments », *in*: *2020 IEEE International Conference on Fog Computing (ICFC)*, IEEE, 2020, pp. 17–24.

[152] Kassian Plankensteiner, Radu Prodan, and Thomas Fahringer, « A new fault tolerance heuristic for scientific workflows in highly distributed environments based on resubmission impact », *in*: *2009 Fifth IEEE International Conference on e-Science*, IEEE, 2009, pp. 313–320.

[153] *PowerfulSeal*, `https://powerfulseal.github.io/powerfulseal/`, [Online; accessed 30-december-2021].

[154] *Prometheus*, `https://prometheus.io/`, [Online; accessed 26-october-2021].

[155] Antonio Puliafito and Kishor S Trivedi, *Systems Modeling: Methodologies and Tools*, Springer, 2018.

[156] M Saifur Rahman, Md Yusuf Sarwar Uddin, Tahmid Hasan, M Sohel Rahman, and M Kaykobad, « Using adaptive heartbeat rate on long-lived TCP connections », *in*: *IEEE/ACM Transactions on Networking* 26.*1* (2017), pp. 203–216.

[157] Ganesan Ramalingam and Kapil Vaswani, « Fault tolerance via idempotence », *in*: *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2013, pp. 249–262.

[158] Thomas Rausch, Waldemar Hummer, Vinod Muthusamy, Alexander Rashed, and Schahram Dustdar, « Towards a serverless platform for edge {AI} », *in*: *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

[159] *Redis*, `https://redis.io/`, [Online; accessed 4-january-2022].

[160] Elena Reshetova, Janne Karhunen, Thomas Nyman, and N Asokan, « Security of OS-level virtualization technologies », *in*: *Nordic Conference on Secure IT Systems*, Springer, 2014, pp. 77–93.

[161] *Resume AWS Step Functions from Any State*, `https://aws.amazon.com/blogs/compute/resume-aws-step-functions-from-any-state/`, [Online; accessed 07-july-2021].

[162] *Rkt*, `https://coreos.com/rkt/`, [Online; accessed 26-november-2021].

[163] Lakshmi Prasad Saikia and Yumnam Langlen Devi, « Fault tolerance techniques and algorithms in cloud computing », *in*: *International Journal of Computer Science & Communication Networks* 4.*1* (2014), pp. 01–08.

[164] Ruben Salvador, Andres Otero, Javier Mora, Eduardo de la Torre, Lukas Sekanina, and Teresa Riesgo, « Fault tolerance analysis and self-healing strategy of autonomous, evolvable hardware systems », *in*: *2011 International Conference on Reconfigurable Computing and FPGAs*, IEEE, 2011, pp. 164–169.

[165] Arnav Sankaran, Pubali Datta, and Adam Bates, « Workflow Integration Alleviates Identity and Access Management in Serverless Computing », *in*: *Annual Computer Security Applications Conference*, 2020, pp. 496–509.

[166] Muhammad Asim Shahid, Noman Islam, Muhammad Mansoor Alam, M.S. Mazliham, and Shahrulniza Musa, « Towards Resilient Method: An exhaustive survey of fault tolerance methods in the cloud computing environment », *in*: *Computer Science Review* 40 (2021), p. 100398, ISSN: 1574-0137, DOI: https://doi.org/10.1016/j.cosrev.2021.100398, URL: https://www.sciencedirect.com/science/article/pii/S1574013721000381.

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

[168] Mikhail Shilkov, *What Is a Cold Start?*, https://mikhail.io/serverless/coldstarts/define/, [Online; accessed 10-october-2021], 2019.

[169] Mounya Smara, Makhlouf Aliouat, Al-Sakib Khan Pathan, and Zibouda Aliouat, « Acceptance test for fault detection in component-based cloud computing and systems », *in*: *Future Generation Computer Systems* 70 (2017), pp. 74–93.

[170] *Snyk*, https://snyk.io/, [Online; accessed 02-november-2021].

[171] Mbarka Soualhia, Foutse Khomh, and Sofiène Tahar, « A dynamic and failure-aware task scheduling framework for hadoop », *in*: *IEEE Transactions on Cloud Computing* 8.*2* (2018), pp. 553–569.

[172] Josef Spillner, Cristian Mateos, and David A Monge, « Faaster, better, cheaper: The prospect of serverless scientific computing and hpc », *in*: *Latin American High Performance Computing Conference*, Springer, 2017, pp. 154–168.

[173] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro, « A Fault-Tolerance Shim for Serverless Computing », *in*: *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, Heraklion, Greece: Association for Computing Machinery, 2020, ISBN: 9781450368827, DOI: 10.1145/3342195.3387535, URL: https://doi.org/10.1145/3342195.3387535.

[174] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov, « Cloudburst: Stateful functions-as-a-service », *in*: *arXiv preprint arXiv:2001.04592* (2020).

[175] *SteamAlert*, https://www.streamalert.io/, [Online; accessed 02-november-2021].

[176] *Sysdig*, https://sysdig.com/products/secure/, [Online; accessed 02-november-2021].

[177] Yue Tan, David Liu, Nanqinqin Li, and Amit Levy, « How Low Can You Go? Practical cold-start performance limits in FaaS », *in*: *arXiv preprint arXiv:2109.13319* (2021).

[178] *The Apache Software Foundation*, https://www.apache.org/, [Online; accessed 07-november-2021].

[179] *Thundra*, https://www.thundra.io/, [Online; accessed 02-november-2021].

[180] *Transient Fault Handling*, https://docs.microsoft.com/en-us/azure/architecture/best-practices/transient-faults, [Online; accessed 03-december-2021].

[181] *Tsung*, http://tsung.erlang-projects.org/user_manual/, [Online; accessed 5-may-2021].

[182] Zhucheng Tu, Mengping Li, and Jimmy Lin, « Pay-per-request deployment of neural network models using serverless architectures », *in*: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, 2018, pp. 6–10.

[183] Blair Felter, *The Different Types of Cloud Computing and How They Differ*, https://www.vxchnge.com/blog/different-types-of-cloud-computing, [Online; accessed 23-november-2021].

[184] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot, « Benchmarking, analysis, and optimization of serverless function snapshots », *in*: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 559–572.

[185] *Virtualization Overview*, `https://www.vmware.com/pdf/virtualization.pdf`, [Online; accessed 23-november-2021].

[186] *VMware vCenter Server*, `https://www.vmware.com/products/vcenter-server.html`, [Online; accessed 4-january-2022].

[187] Bin Wang, Ahmed Ali-Eldin, and Prashant Shenoy, « LaSS: Running Latency Sensitive Serverless Computations at the Edge », *in*: *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, 2020, pp. 239–251.

[188] Hao Wang, Di Niu, and Baochun Li, « Distributed machine learning with a serverless architecture », *in*: *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, IEEE, 2019, pp. 1288–1296.

[189] 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)*, 2018, pp. 133–146.

[190] *What is Serverless Security?*, `https://www.checkpoint.com/cyber-hub/cloud-security/what-is-serverless-security/`, [Online; accessed 02-november-2021].

[191] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein, « Anna: A kvs for any scale », *in*: *IEEE Transactions on Knowledge and Data Engineering* (2019).

[192] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein, « Transactional causal consistency for serverless computing », *in*: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 83–97.

[193] Shelly Xiaonan Wu and Wolfgang Banzhaf, « The use of computational intelligence in intrusion detection systems: A review », *in*: *Applied soft computing* 10.*1* (2010), pp. 1–35.

[194] Lai Xinming, Wang Haitao, Zhao Jing, Zhang Fan, Zhao Chao, and Wu Gang, « Research on High Availability Architecture of Cloud Platform », *in*: *Journal of Physics: Conference Series*, vol. 1345, 2, IOP Publishing, 2019, p. 022044.

[195] Fei Xu, Yiling Qin, Li Chen, Zhi Zhou, and Fangming Liu, « lambda DNN: Achieving Predictable Distributed DNN Training with Serverless Architectures », *in*: *IEEE Transactions on Computers* (2021).

[196] Zhengjun Xu, Haitao Zhang, Xin Geng, Qiong Wu, and Huadong Ma, « Adaptive function launching acceleration in serverless computing platforms », *in*: *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, IEEE, 2019, pp. 9–16.

[197] Rimmy Yadav and Avtar Singh Sidhu, « Fault tolerant algorithm for replication management in distributed cloud system », *in*: *2015 IEEE 3rd International Conference on MOOCs, Innovation and Technology in Education (MITE)*, IEEE, 2015, pp. 78–83.

[198] Vladimir Yussupov, Uwe Breitenbücher, Ayhan Kaplan, and Frank Leymann, « SEAPORT: Assessing the Portability of Serverless Applications. », *in*: *CLOSER*, 2020, pp. 456–467.

[199] Vladimir Yussupov, Uwe Breitenbücher, Frank Leymann, and Christian Müller, « Facing the unplanned migration of serverless applications: A study on portability problems, solutions, and dead ends », *in*: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, 2019, pp. 273–283.

[200] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu, « Fault-tolerant and transactional stateful serverless workflows », *in*: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, Banff, Alberta: USENIX Association, Nov. 2020, URL: `https://www.usenix.org/conference/osdi20/presentation/zhang-haoran`.

[201] 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*, 2019, pp. 61–66.

[202] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker, « Kappa: A programming framework for serverless computing », *in*: *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 328–343.

[203] Junlong Zhou, Mingyue Zhang, Jin Sun, Tian Wang, Xiumin Zhou, and Shiyan Hu, « Drheft: Deadline-constrained reliability-aware heft algorithm for real-time heterogeneous mpsoc systems », *in*: *IEEE Transactions on Reliability* (2020).

Résumé en français

Introduction

Il y a encore quelques années les organisations devaient posséder et gérer leurs propres serveurs physiques pour exécuter leurs logiciels, ce qui entraînait des coûts en termes d'investissement et de maintenance. Le *cloud computing* a introduit de nouveaux services et de nouvelles solutions pour l'accès aux ressources informatiques. Récemment, un nouveau modèle de service de cloud a émergé, appelé Fonction en tant que Service (FaaS), qui permet aux développeurs de créer et d'exécuter des applications sans avoir à gérer des serveurs. Les développeurs se concentrent sur l'écriture du code de l'application tandis que le fournisseur de plateforme FaaS est responsable de la gestion et de la configuration de l'infrastructure. Dans ce modèle, les applications se présentent sous la forme de fonctions. Les fonctions sont des fragments de code qui exécutent une tâche spécifique. Dans le modèle FaaS, les fonctions ont une courte durée de vie et s'exécutent dans un conteneur sans état qui est déclenché par des événements. Le passage à l'échelle des fonctions est géré automatiquement et les clients ne sont facturés que pour les ressources utilisées pendant l'exécution des fonctions.

Les principaux fournisseurs de cloud offrent une plateforme FaaS. Par exemple, Amazon et Google fournissent respectivement *Lambda* et *Google Functions*. De plus, plusieurs environnements open source pour le déploiement d'applications FaaS ont émergé, tels que Fission, OpenFaaS et Knative. Le modèle FaaS attire un nombre croissant d'utilisateurs du cloud qui l'ont adopté pour différentes classes d'applications, à savoir le traitement de vidéos, le calcul scientifique, l'internet des objets et les applications d'apprentissage automatique.

Bien qu'un environnement FaaS offre de nombreux avantages, il présente un certain nombre de limites, notamment du point de vue de la tolérance aux fautes. Étant donné que la tolérance aux fautes est gérée par le fournisseur de service FaaS, ce dernier doit assurer une haute disponibilité pour les fonctions déployées. La plupart des environnements FaaS fournissent un mécanisme basique de tolérance aux fautes qui consiste à relancer l'exécution des fonctions en l'absence de réponse. Toutefois, si le mécanisme de relance

permet de faire face aux délais induits par le réseau, il entraîne de longs délais de recouvrement lorsqu'il s'agit de traiter d'autres types de fautes. Ainsi, le mécanisme de relance ne permet pas de tolérer des fautes permanentes. Par exemple, si l'exécution d'une fonction échoue en raison de la défaillance d'un nœud, la requête sera relancée inutilement plusieurs fois.

D'autres approches de tolérance aux fautes présentent des propriétés de disponibilité, de performance et de consommation des ressources différentes, ce qui les rend appropriées pour différents scénarios de défaillance. Par exemple, les approches de réplication qui utilisent des instances de fonction déployées sur plusieurs nœuds sont appropriées pour gérer les pannes franches des nœuds. En particulier, l'approche de réplication active favorise les performances et convient aux applications sensibles à la latence. L'approche de réplication passive favorise une consommation de ressources réduite et convient aux environnements dont les ressources sont limitées [68]. Par conséquent, notre objectif principal dans cette thèse est de permettre aux plateformes FaaS de supporter des approches de tolérance aux fautes supplémentaires afin d'assurer l'exécution fiable des fonctions FaaS en présence de différents types de fautes, tout en maintenant les performances et en réparant le système après une faute sans affecter de manière significative l'expérience utilisateur.

Tolérance aux fautes dans les environnements FaaS

Dans les environnements FaaS, des défaillances peuvent se produire en raison de différents types de dysfonctionnement tels que des délais de communication ou des défaillances matérielles. Par exemple, en cas de panne franche d'un nœud de calcul, l'utilisateur peut subir une instabilité des performances jusqu'à la réparation du système. Les défaillances doivent être prises en compte dans les systèmes FaaS afin de garantir la haute disponibilité des fonctions. Le mécanisme couramment utilisé pour gérer les défaillances des fonctions dans les plateformes FaaS est le mécanisme de relance. Les principales plateformes commerciales, telles que AWS Lambda [9], Google Cloud Functions [81] et Microsoft Azure Functions [51], fournissent toutes une fonctionnalité de relance automatique pour gérer les défaillances et les délais. Par exemple, AWS Lambda réessaye des invocations asynchrones jusqu'à deux fois avec un délai entre les tentatives. Certaines plateformes FaaS open source offrent également le mécanisme de relance comme Fission et OpenFaaS, qui relancent les invocations asynchrones avec un délai exponentiel [75, 142].

La tolérance aux fautes dans les systèmes FaaS peut également être réalisée en utilisant des services supplémentaires fournis par les plateformes de cloud computing. Par exemple, en utilisant les services d'équilibrage de charge et d'ingestion d'événements d'Azure, les développeurs peuvent déployer des fonctions dans différentes régions selon un modèle actif-actif ou actif-passif, ce qui offre une protection contre les scénarios de catastrophe [26]. À l'aide des services d'orchestration FaaS (tels que Google Workflows [82], AWS Step Functions [27] ou Azure Durable Functions [32]), les développeurs peuvent définir des workflows qui coordonnent les fonctions, relancent automatiquement les invocations qui ont échoué ou dont le délai est dépassé et exécutent un code personnalisé pour gérer différents types d'erreurs. Par exemple, à l'aide d'AWS Step Functions, les développeurs peuvent reprendre les workflows ayant échoué à partir de l'état dans lequel ils ont échoué [161]. Des capacités similaires sont fournies par des environnements d'orchestration open source, tels que Apache OpenWhisk Composer [20] ou Faas-flow pour OpenFaaS [67]. Notre travail se concentre sur les mécanismes de tolérance aux fautes mis en œuvre au sein des plateformes FaaS sans impliquer de services externes.

Des recherches récentes étudient la tolérance aux fautes pour les applications FaaS avec état, composées de plusieurs fonctions et interagissant avec des services de stockage. Sreekanti et al. [173] proposent d'insérer une couche entre la plateforme FaaS et un système de stockage clé-valeur pour assurer la visibilité atomique des mises à jour dans le système de stockage. Le système proposé assure la tolérance aux fautes en appliquant la garantie de cohérence atomique en lecture. Zhang et al. [200] décrivent une bibliothèque et un environnement d'exécution (runtime) pour créer des workflows transactionnels et tolérants aux fautes sur les plateformes FaaS existantes. Le système prend en charge les transactions au sein et entre des fonctions en appliquant une approche de tolérance aux fautes fondée sur des journaux (logs). Jia et al. [101] proposent Boki, un runtime FaaS qui offre une API pour les applications à état. L'API permet aux applications de gérer leur état et utilise un mécanisme fondé sur des logs pour assurer la tolérance aux fautes. Wu et al. [192] présentent HydroCache, une couche de cache distribuée pour les systèmes FaaS, qui fournit une cohérence causale transactionnelle pour les fonctions à état. Le système s'appuie sur le système de stockage Anna [191], un backend d'état clé-valeur qui supporte la tolérance aux fautes. Pour garantir la tolérance aux fautes, les transactions sont relancées avec la même version de clé en cas de défaillance du nœud de stockage ou de délais de transmission dans le réseau. Les défaillances des nœuds sont détectées à l'aide d'un mécanisme de *heartbeat* et les fonctions non terminées du nœud défaillant sont reprogrammées sur un

autre nœud. Contrairement à ces systèmes, notre travail se concentre sur la garantie de la tolérance aux fautes pour les fonctions individuelles et idempotentes plutôt que pour les compositions de fonctions avec état, et n'impose pas l'utilisation d'API supplémentaires aux développeurs d'applications FaaS.

Un autre travail récent [89] introduit un modèle de programmation et une mise en œuvre associée pour la prise en charge des transactions entre les fonctions FaaS avec état. Ce travail s'appuie sur Apache Flink StateFun [15], une plateforme open source pour les fonctions FaaS avec état qui utilise un moteur de flux de données en continu. La plateforme gère les défaillances via des points de reprise ou *snapshots* pour obtenir la garantie de l'unicité de traitement. Le modèle de programmation StateFun prend en charge l'encapsulation de l'état dans les instances de fonction, ce qui n'est pas permis dans le modèle FaaS classique.

Zhang et al. [202] proposent Kappa, un environnement de programmation pour la construction d'applications FaaS parallèles. Cet environnement vérifie périodiquement les résultats des fonctions afin de permettre la reprise après défaillance. Carver et al. [45] présentent Wukong, un environnement pour construire des applications FaaS parallèles au-dessus de AWS Lambda. En cas de défaillance, le mécanisme de relance automatique d'AWS Lambda est utilisé pour ré-exécuter la fonction qui a échoué. Ces deux systèmes proposent des bibliothèques construites au-dessus d'une plateforme FaaS non modifiée (AWS Lambda) alors que les mécanismes que nous proposons dans cette thèse sont intégrés dans la plateforme FaaS elle-même.

Karhula et al. [105] proposent d'utiliser Docker et CRIU (*Checkpoint/Restore in Userspace*) pour la sauvegarde de points de reprise et le recouvrement des fonctions de longue durée qui s'exécutent sur des dispositifs IoT ainsi que pour la migration des fonctions entre différents dispositifs IoT. Bien que ces mécanismes puissent être utilisés comme blocs de base pour la construction d'un système FaaS tolérant aux fautes, ce travail n'a pas abouti à une mise en œuvre complète d'un tel système.

En résumé, il existe plusieurs solutions qui proposent des mécanismes de tolérance aux fautes au-delà du mécanisme basique de relance de l'exécution des requêtes. Cependant, toutes ces solutions nécessitent l'utilisation d'API et de primitives en dehors de celles fournies par le modèle FaaS de base, qui ne prend en charge l'appel de fonctions qu'en réponse à des événements. Par exemple, ces solutions nécessitent l'utilisation de services d'équilibrage de charge [26], de services d'orchestration de workflow [161] ou de modèles de programmation spécialisés pour des fonctions à état [173, 200, 101, 192, 89]. À notre

connaissance, notre travail est le premier à intégrer des mécanismes de tolérance aux fautes dans une plateforme FaaS sans dévier du modèle FaaS originel et donc sans ajouter de complexité pour les développeurs.

Contributions principales

Comparaison de plateformes FaaS diffusées en open source

Dans notre thèse, nous avons commencé par une étude de trois plateformes FaaS open source, à savoir Fission, Kubeless et OpenFaaS, afin de sélectionner la plateforme la plus appropriée pour intégrer de nouveaux mécanismes de tolérance aux fautes. Notre sélection est fondée sur un ensemble défini de critères, notamment l'extensibilité et la popularité de la plateforme, la qualité de la documentation et la taille et l'activité de la communauté open source. Nous avons évalué les performances des trois environnements FaaS open source qui s'appuient tous sur la plateforme d'orchestration de conteneurs Kubernetes. Parmi les environnements FaaS évalués, nous avons sélectionné Fission car il est très représentatif des plateformes FaaS existantes et offre de bonnes performances. En outre, Fission met en œuvre le mécanisme de relance pour la tolérance aux fautes, consistant essentiellement à redémarrer l'ensemble du processus de soumission d'une requête ayant échoué. Nous avons ainsi pu comparer les approches de tolérance aux fautes que nous avons intégrées à Fission avec l'approche classiquement mise en œuvre dans les plateformes FaaS.

Proposition d'approches de tolérance aux fautes pour environnements FaaS et leur intégration dans Fission

Dans les environnements FaaS, la ré-exécution des fonctions en cas de défaillance est une stratégie courante pour tolérer les fautes. Le mécanisme de relance est bien adapté aux fautes transitoires. Cependant, il n'est pas efficace pour gérer d'autres types de fautes, comme les fautes permanentes, un type de faute fréquent dans les infrastructures sous-jacentes. Cela motive le besoin d'autres approches de tolérance aux fautes pour assurer la haute disponibilité des fonctions FaaS en cas de défaillance.

Dans notre travail, nous avons proposé l'intégration dans une plateforme FaaS de deux approches de tolérance aux fautes à base de réplication des calculs afin de rendre les fautes transparentes pour les applications. La première approche, appelée Active-Standby (AS) est fondée sur l'utilisation de la réplication passive. Elle consiste à créer deux répliques

(*replica*) d'une fonction, l'une active et l'autre en veille prenant le relais en cas de défaillance de la réplique active. La seconde approche, appelée Réplication de Requête (RR), est fondée sur la réplication active. Elle consiste à créer deux répliques actives d'une fonction et à envoyer toutes les requêtes aux deux répliques, chacune exécutant les requêtes reçues. Les deux approches ont été intégrées dans Fission en utilisant l'exécuteur *NewDeploy* pour créer les répliques de fonction et de nouveaux routeurs ont été implémentés et utilisés à la place du routeur mis en œuvre nativement dans Fission. Dans l'approche AS, le routeur transmet tous les appels de fonction reçus spécifiquement au pod actif. Dans l'approche RR, le routeur réplique chaque requête reçue sur toutes les répliques de fonction, afin de la traiter en parallèle. Il envoie ensuite la première réponse reçue à l'utilisateur et ignore la seconde.

Évaluation des approches de tolérance aux fautes proposées

Après l'intégration des deux approches de tolérance aux fautes à base de répliquetion des calculs dans Fission, nous les avons comparées expérimentalement avec le mécanisme basique de relance implémenté nativement dans Fission (vanilla). Les expériences ont été menées sur la plateforme d'expérimentation de systèmes distribués Grid'5000. Nous avons utilisé plusieurs métriques telles que la performance, la disponibilité et la consommation de ressources, à la fois en fonctionnement normal et dans divers scénarios de défaillance. Nous avons considéré les scénarios suivants : défaillance d'un pod Kubernetes, défaillance d'un nœud de l'infrastructure et différents délais de transmission des messages dans le réseau. Les résultats obtenus montrent que l'approche RR est plus performante que les autres dans les trois scénarios de défaillance indiquant son efficacité. Cette approche est capable d'assurer une haute disponibilité des fonctions, par rapport aux approches AS et vanilla (mécanisme de relance). Cependant, l'approche RR nécessite une quantité importante de ressources supplémentaires pour atteindre la tolérance aux fautes des applications FaaS. Dans le cas où il n'y a pas de défaillance, la surcharge de l'utilisation de RR est de 180% en termes de consommation de CPU et de 52% en termes de consommation de mémoire par rapport à vanilla dans nos expériences. Cela est dû aux ressources supplémentaires allouées à la deuxième réplique. Dans l'approche vanilla, une seule réplique exécute les requêtes. L'approche AS a un surcoût de 141% en consommation CPU et de 39% en consommation de mémoire par rapport à l'approche vanilla. Il faut noter que dans l'approche AS, la réplique en veille est chaude, c'est-à-dire qu'elle est chargée en mémoire. Contrairement à la réplique active de la stratégie RR, la réplique en veille de l'approche AS ne traite pas

les requêtes, mais elle transmet des *heartbeats* réguliers, ce qui consomme des ressources.

Leçons tirées de l'étude

D'après notre comparaison expérimentale des trois stratégies de tolérance aux fautes (mécanisme basique de relance, approche Active-Standby, réplication des requêtes), nous constatons que les approches étudiées ont des propriétés différentes et qu'elles sont donc intéressantes dans des conditions différentes. Le mécanisme de relance est bien adapté aux fautes transitoires qui durent peu de temps, notamment les délais de transmission des réseaux. Cette approche consomme moins de ressources que l'approche Active-Standby et que la stratégie de réplication des requêtes. L'approche Active-Standby offre une meilleure disponibilité en présence de défaillances de longue durée par rapport à l'approche de relance mais au prix d'une consommation de ressources plus élevée. Par exemple, dans nos expériences, l'approche Active-Standby consomme plus de deux fois le CPU consommé par l'approche de relance. L'approche de réplication des requêtes offre la meilleure disponibilité pour tout type de faute. En effet, lorsque la défaillance n'affecte pas toutes les répliques, il n'y a quasiment aucun impact sur la disponibilité globale des fonctions. Cette approche offre également des performances de meilleur niveau et d'une plus grande stabilité que les autres stratégies. Enfin, la stratégie de réplication des requêtes engendre la plus grande consommation de ressources. En général, nous observons que la disponibilité et la consommation de ressources dans les trois approches sont inversement liées.

Chacune des approches de tolérance aux fautes a ses propres avantages et inconvénients et peut être utilisée pour des scénarios spécifiques de tolérance aux fautes. Le choix de l'approche de tolérance aux fautes appropriée dans le contexte des plateformes FaaS dépend des exigences qui doivent être respectées. Si l'accent est mis sur les bonnes performances (par exemple, pour les applications sensibles à la latence), l'approche privilégiée est la réplication de requêtes (RR). Si l'accent est mis sur la réduction des ressources, par exemple pour les environnements limités en ressources, comme à la bordure du réseau (*edge computing*), et le besoin de haute disponibilité, l'approche privilégiée est ActiveStandy (AS). Le mécanisme de relance peut être utilisé lorsqu'il s'agit d'économiser des ressources et que le besoin de disponibilité est limité.

Conclusion et perspectives

Bilan des travaux effectués

Dans cette thèse, nous avons identifié le défi de la tolérance aux fautes dans les environnements FaaS. Nous avons remarqué que le mécanisme de relance est couramment utilisé pour offrir de la tolérance aux fautes dans les plateformes FaaS. Cependant, ce mécanisme n'est pas efficace pour faire face aux fautes permanentes, par exemple la panne franche d'un nœud de l'infrastructure de calcul. Dans cette thèse, nous avons étudié d'autres approches de tolérance aux fautes, à savoir des approches fondées sur la réplication des calculs. Nous avons intégré la réplication active (réplication de requêtes) et la réplication passive (Active-Standby) dans la plateforme Fission et nous avons comparé expérimentalement ces deux approches avec le mécanisme de relance mis en œuvre nativement dans Fission.

La comparaison expérimentale des trois approches a mis en évidence les forces et les faiblesses de chacune d'entre elles et leur comportement face aux défaillances. Les résultats expérimentaux ont notamment montré que le mécanisme de relance n'est pas suffisant pour assurer une haute disponibilité des fonctions en cas de défaillances de longue durée telles que les pannes franches de nœuds. En effet, le comportement par défaut du mécanisme de relance entraîne un temps de recouvrement élevé en cas de défaillance d'un nœud. Avec l'approche Active-Standby (AS), le temps de recouvrement est réduit car le service devient disponible peu après que la réplique en veille ait détecté la défaillance de la réplique active. Avec la réplication des requêtes (RR), le service reste disponible tant qu'au moins une réplique continue à répondre aux utilisateurs et le recouvrement ne dépend pas du remplacement de la réplique défectueuse. Les deux approches de réplication utilisent plus de ressources que le mécanisme de relance.

Perspectives

Le travail effectué dans cette thèse a ouvert de nouvelles perspectives. Par exemple, dans nos expériences, les trois approches ont été testées avec une application qui calcule la suite de Fibonacci. Néanmoins, pour renforcer notre étude, il est important de tester les approches présentées en utilisant des applications sans état de différents domaines (i.e. IoT, traitement vidéo, apprentissage automatique).

Une autre direction pour les travaux futurs est liée à la façon d'adapter les tech-

niques de tolérance aux fautes des plateformes FaaS au contexte du *edge computing*. Dans cet environnement, les ressources sont limitées et le taux de fautes permanentes des périphériques est élevé. Par conséquent, le mécanisme de relance n'est pas bien adaptée car la plupart des défaillances sont permanentes. L'approche de réplication des requêtes (RR) semble également inappropriée car les ressources à la périphérie du réseau sont limitées et coûteuses. Des recherches complémentaires sont donc nécessaires pour fournir une solution efficace afin d'améliorer la tolérance aux fautes des fonctions déployées à la périphérie du réseau dans le contexte du *edge computing*.

Dans notre travail, nous avons considéré des applications sans état et idempotentes, où la même entrée donne toujours la même sortie. Pour satisfaire d'autres cas d'utilisation, les architectures et les offres FaaS ont évolué récemment pour prendre en compte l'état des applications. Par conséquent, il serait souhaitable de concevoir des approches de tolérance aux fautes pour les applications FaaS avec état. Avec les applications à état, un état est généralement maintenu dans des services de stockage externes, tels que des bases de données NoSQL [200]. L'utilisation de l'approche de réplication des requêtes pour de telles applications semble difficile. En effet, les accès concurrents augmentent la charge sur le service de stockage et introduisent un surcoût important pour maintenir la cohérence. Cela peut entraîner des performances réduites dans le cas d'un fonctionnement normal (sans fautes) par rapport à l'utilisation de l'approche Active-Standby ou du mécanisme de relance. L'intégration de caches dans les fonctions à état pourrait atténuer ce problème [174].

Avec cette thèse axée sur différentes approches de tolérance aux fautes, une extension naturelle serait de concevoir un système de tolérance aux fautes pour FaaS qui offre simultanément plusieurs approches, telles que le mécanisme de relance, l'approche Active-Standby et la stratégie de réplication des requêtes, et qui utilise l'une ou l'autre en fonction de facteurs spécifiques tout en répondant aux besoins des utilisateurs (i.e., performance, disponibilité, consommation de ressources). Ces facteurs peuvent inclure les types d'application (i.e., avec ou sans état) et les conditions de fonctionnement (i.e., taux de défaillance, latence réseau).

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Tolérance aux fautes dans les environnements FaaS

Mot clés : Fonction en tant que service, Cloud, Tolérance aux fautes, Haute disponibilité

Résumé : Fonction en tant que service (FaaS) est un modèle de programmation émergent pour construire des applications cloud dans lesquelles la gestion de l'infrastructure est abstraite pour le développeur. L'un des principaux défis des systèmes FaaS est de fournir la tolérance aux fautes des fonctions déployées. Le mécanisme de tolérance aux fautes de base dans les plates-formes FaaS actuelles consiste à réessayer automatiquement les invocations de fonctions. Bien que ce mécanisme soit bien adapté aux fautes transitoires, il entraîne des délais dans le recouvrement en présence d'autres types de fautes comme les fautes permanentes. Notre objectif est de fournir la haute disponibilité des applications FaaS quel que soit le type de fautes. Dans cette thèse, nous proposons l'intégration dans les plates-formes FaaS d'approches de tolérance aux fautes fondées sur des schémas de réplication passive et active. Nous décrivons comment nous avons réalisé cette intégration dans Fission, un environnement open source bien connu. Nous analysons les résultats d'une évaluation expérimentale approfondie comparant les mécanismes proposés avec le mécanisme de relance des invocations de fonction en termes de performance, de disponibilité et de consommation de ressources, à la fois en fonctionnement normal et sous différents scénarios de défaillance.

Title: Fault Tolerance in FaaS Environments

Keywords: Function as a Service, Cloud, Fault Tolerance, High Availability

Abstract: Function as a Service (FaaS) is an emerging programming model for building cloud applications where the infrastructure management is abstracted away from the developer. One of the main challenges of FaaS systems is providing fault tolerance for the deployed functions. The basic fault tolerance mechanism in current FaaS platforms is automatically retrying function invocations. Although the retry mechanism is well-suited for transient faults, it incurs delays in recovering from other types of faults such as permanent faults. Our objective is to provide high availability for FaaS applications regardless of the type of faults. In this thesis, we propose the integration of fault tolerance approaches based on passive and active replication schemes in FaaS platforms. We describe how we performed this integration in Fission, a well-known, open source framework. Furthermore, we provide a detailed experimental comparison of the proposed mechanisms with the retry mechanism in terms of performance, availability, and resource consumption, both in normal functioning and under different failure scenarios.