Active-Standby for High-Availability in FaaS

Yasmina Bouizem, Djawida Dib, Nikos Parlavantzas, Christine Morin

Active-Standby for High-Availability in FaaS

Yasmina Bouizem
Univ Tlemcen, LRIT
Univ Rennes, Inria

Djawida Dib
Univ Tlemcen, LRIT

Nikos Parlavantzas
INSA Rennes, IRISA

Christine Morin
Univ Rennes, Inria

Abstract

Serverless computing is becoming more and more attractive for cloud solution architects and developers. This new computing paradigm relies on Function-as-a-Service (FaaS) platforms that enable deploying functions without being concerned with the underlying infrastructure. An important challenge in designing FaaS platforms is ensuring the availability of deployed functions. Existing FaaS platforms address this challenge principally through retrying function executions. In this paper, we propose and implement an alternative fault-tolerance approach based on active-standby failover. Results from an experimental evaluation show that our approach increases availability and performance compared to the retry-based approach.

CCS Concepts: • **Computer systems organization** → **Availability**.

Keywords: FaaS, fault tolerance, availability

ACM Reference Format:
Yasmina Bouizem, Djawida Dib, Nikos Parlavantzas, and Christine Morin. 2020. Active-Standby for High-Availability in FaaS. In *Workshop on Serverless Computing (WoSC'20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3429880.3430097

1 Introduction

Serverless computing is a new computing paradigm for developing distributed cloud-based systems. This paradigm is principally supported by Function-as-a-Service (FaaS) platforms, which allow developers to write and deploy functions without being concerned with provisioning, configuring, and managing servers. Developers can thus concentrate on the logic and business value of their applications while the cloud provider takes full responsibility for managing the underlying infrastructure.

One of the main challenges for FaaS providers is ensuring high availability for the deployed functions. Indeed, high availability and built-in fault tolerance are touted as main features of commercial FaaS platforms(e.g.,[2]). All current FaaS platforms support a basic form of fault-tolerance through retrying function executions. Our work is the first to propose applying an alternative fault-tolerance mechanism in FaaS, namely active-standby failover. Specifically, this paper makes two contributions. First, it describes a High-Availability (HA) approach for FaaS based on active standby [7] and its implementation in an open-source FaaS platform, namely Fission. Second, it provides a detailed comparison of this approach with the retry-based approach using experiments on the Grid'5000 testbed [11].

The paper is organized as follows. Section 2 reviews related work on fault-tolerance mechanisms in FaaS environments. Section 3 describes the retry mechanism used in existing FaaS environments. Section 4 presents our failover solution. Section 5 describes our experimental setup and Section 6 analyses the results of our evaluation. We conclude and discuss future work in Section 7.

2 Background

The main mechanism for supporting fault tolerance in current FaaS platforms is retrying invocations. The major commercial platforms, AWS Lambda [3, 4], Google Cloud Functions [10] and Microsoft Azure Functions [8], automatically retry invocations after failures or timeouts. Open-source platforms also apply the retry mechanism. For instance, OpenFaaS retries asynchronous invocations based on a timeout [16]. Fission retries function invocations using a router component [9]. Azure Functions includes support for deploying functions in different regions in an active-active or active-passive pattern, which provides protection against disaster scenarios [5]. [18] proposes inserting a layer between commodity FaaS platforms and key-value stores to ensure atomic visibility of storage updates. The proposed model also relies on the retry mechanism and adds support for read atomic isolation.

In this paper, we propose applying the active-standby high availability (HA) mechanism [6] in FaaS platforms and compare it to the typical retry mechanism and, in particular, its implementation in Fission.

WoSC'20, December 7–11, 2020, Delft, Netherlands

ACM ISBN 978-1-4503-8204-5/20/12…$15.00
https://doi.org/10.1145/3429880.3430097

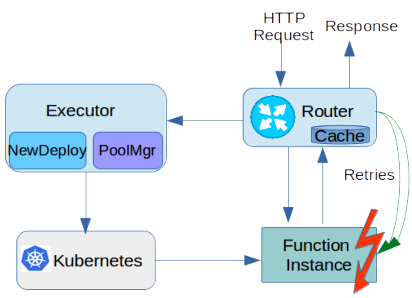


Figure 1. Overview of the Retry mechanism in Fission

3 The Used FaaS Framework

To implement our approach, we selected a popular open source framework, namely Fission, due to its ease of deployment and flexibility. Fission uses the typical, retry-based approach that we intend to compare with our approach. We assume that functions are idempotent in both approaches. In the next subsections we describe the architecture and the fault-tolerance mechanism used in Fission.

3.1 Fission Architecture

Fission [9] is based on Kubernetes' core abstractions, such as deployments, pods and services. Deployments are declarative objects that describe a deployed application. Pods are collections of application containers running in the same execution environment. Services are collections of policies for accessing specific pods with load balancing, naming and discovery [13]. Fission has two main components: an Executor and a Router. The Executor creates and controls the lifecycle of function pods. There are two types of Executors: PoolManager and NewDeploy. PoolManager maintains a pool of generic warm containers to reduce the cold start time [19] of functions. This executor type does not support auto-scaling. NewDeploy is based on creating Kubernetes deployments, services and a Horizontal Pod Autoscaler, which enables autoscaling function pods. The Router routes a function call to the corresponding function pod and retries in case of failures. Figure 1 illustrates the architecture of Fission and its retry mechanism to tolerate faults.

3.2 Existing Retry Mechanism in Fission

The retry mechanism used in Fission works as follows (see Figure 2). First, the Router receives a function call and 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 pod. If the request fails, the Router retries to forward the request to the function service up to a configurable maximum number of retries with an exponential back-off before giving up [1]. If all the retries fail or if the received response is a network dial error, Fission assumes that the pod doesn't exist anymore. Thus, the Router removes the service cache entry and requests the Executor to get a new service for the function. Then, it retries to forward the request to the new function service and so on.

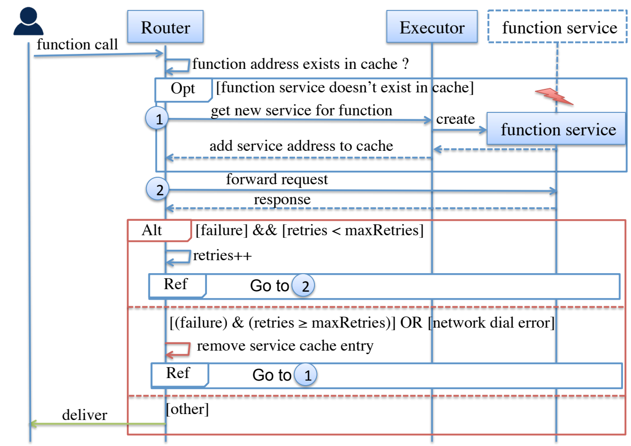


Figure 2. Fault tolerance protocol with the Retry mechanism

4 Proposed Active-Standby Approach

In this section, we present the Active-Standby approach we propose and its implementation in Fission.

4.1 Active-Standby for FaaS

The Active-Standby solution consists in creating two function instances. The first instance acts as the active instance and serves all requests during normal usage. The second instance is passive or on standby. The active and the standby instances are connected by a heartbeat mechanism that continually checks their connectivity and status. If the heartbeat of one instance isn't received within a configured amount of time, an action is triggered depending on the instance's type. In case of an unreachable passive instance, another passive instance is created. In case of an unreachable active instance, the standby instance is activated to serve incoming requests and another passive instance is created.

4.2 Implementation in Fission

The implementation of our approach in Fission required us to use specific components from both Fission and Kubernetes. First, we use the NewDeploy executor type because this one supports creating replicas of function pods. Second, we use the Kubernetes Readiness Probe [15] to specify the state of pods. 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. Third, instead of using the Router to get the function address, we use the Kubernetes DNS server "CoreDNS" to get the IP address of the active pod. Finally, in case of failures the pods are recreated and the NewDeploy Executor ensures that the two replicas of the function are running.

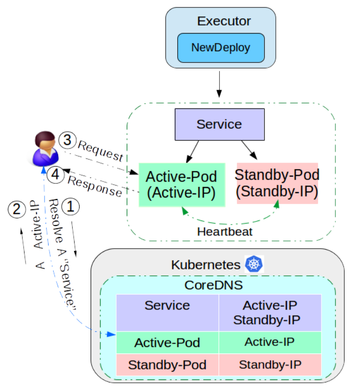


Figure 3. Overview of the Active-Standby mechanism in Fission

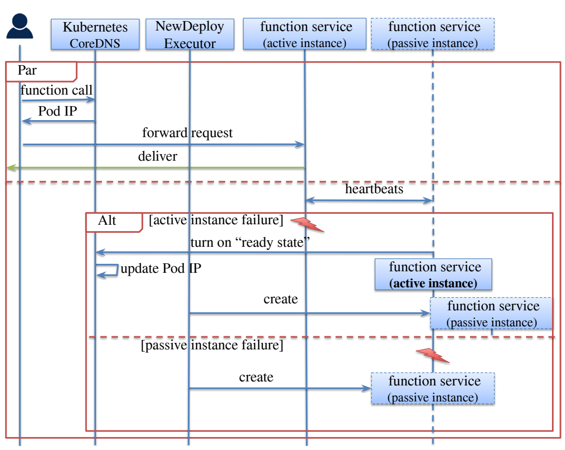


Figure 4. Fault tolerance protocol with the Active-Standby mechanism

The implemented Active-Standby mechanism in Fission works as follows (see Figure 4). The Kubernetes CoreDNS receives the function call and returns the IP address of the active pod. The user forwards her request directly to the active pod. In parallel, both active and standby pods send and receive regular heartbeats to and from each other for health checks. The heartbeats are configured using Kubernetes Readiness probes. The probes are performed each 1 second (the minimum configurable value). When the active pod is running, the passive pod fails the readiness probe and stays running in a not-ready state. If the active pod fails, the passive pod succeeds the readiness probe and becomes active. Then, another pod is created to replace the passive pod. The same action happens if the passive pod crashes for any reason.

5 Experimental Setup

This section describes the experimental setup for evaluating the effectiveness of our active-standby approach and comparing it with the retry mechanism used in Fission.

5.1 Test Environment

We performed our experiments on the Grid'5000 [11] testbed. We used 5 nodes on the Lyon site to deploy Kubernetes [14] (version 1.11), Fission AS (Active-Standby) and the original version of Fission (vanilla version 1.5.0), each node having 2 CPUs Intel Xeon E5-2620 v4, 8 cores/CPU, and 64 GB memory. We setup 2 additional nodes, one to invoke functions and another one to inject faults.

5.2 Test Scenarios

We defined two sets of failure scenarios. In the first set, an application failure is due to a pod failure whereas in the second set it is due to a node failure. In the first scenario with pod failures, we use PowerfulSeal tool [17] to inject faults to pods. The failure is simulated by killing the function pod at a random time between 30 s and 60 s from the beginning of the workload execution. In the second scenario with node failures, 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. Each scenario has been repeated at least 5 times with the deployed applications in Fission AS and vanilla. The averages of the measurements are shown in all Figures, Table 1 and Table 2.

5.3 Applications

We used two HTTP-Triggered functions. The first one is Fibonacci, a CPU-intensive function that computes a Fibonacci 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 [12].

5.4 Workload

The workload is generated with Tsung [20], a high-performance benchmark framework. In our test, we generated 3000 requests during 5 minutes.

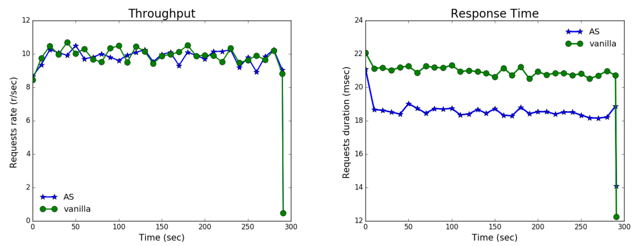
5.5 Metrics

We evaluate our solution using the following metrics:

- Performance: The performance is measured using throughput and response time values. 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.
- 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. We also capture the HTTP status code returned in the response.
- Resource consumption: The resource consumption is measured as the amount of CPU and memory consumed by nodes during the execution of the workload.

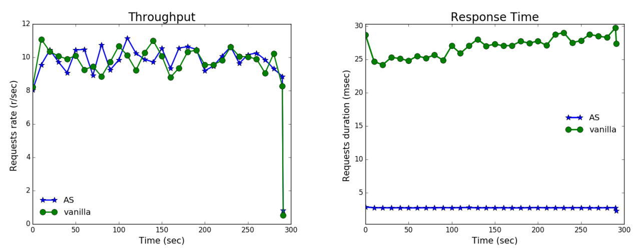
6 Experimental Results and Discussion

We performed three different sets of experiments: (1) Experiments without failures; (2) Experiments with pod failures; (3) Experiments with node failures.



(a) Throughput **(b)** Response Time

Figure 5. Fibonacci without failures



(a) Throughput **(b)** Response Time

Figure 6. Guestbook application without failures

6.1 Experiments without failures

Figure 5 and Figure 6 present the throughput and average response time of the Fibonacci and Guestbook applications deployed with both Fission AS and Fission vanilla without failures. From Figure 5(a) and Figure 6(a), we can observe that the throughput for the two functions in both versions of Fission is quite similar. AS and vanilla are both capable of processing in average 11 requests per second. Figures 5(b) and 6(b) 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 around 2ms and 16 ms respectively. The higher response time obtained with Fission vanilla is explained by its use of the Router component to route the request to function instances. AS performs much better than vanilla in this scenario and provides fastest response times.

6.2 Experiments with pod failures

Figure 7 and 8 show the throughput, average response times, and HTTP code response rates of the Fibonacci and Guestbook applications with Fission AS and vanilla, with pod failures. From Figures 7(a), 7(b), 8(a) and 8(b) we see that AS and vanilla react to the failure differently. For instance, 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. This leads to a waste of time and resources as it is essentially re-executing a request that is likely to fail at the end. All failed requests return errors, which is represented by codes 503 and 502 in Figure 7(c) and Figure 8(c). However in AS, just after the failure is detected, the traffic is forwarded to the standby instance, which explains seeing only successful responses represented by code 200 in Figure 7(d) and Figure 8(d). The recovery time (RT) is shown in Table 1. We can see that the RT of Fibonacci and Guestbook functions under AS is 1.814s and 1.528s, respectively, whereas under vanilla is 2.840s and 3.614s, respectively. These results show that our approach enables faster recovery than the retry mechanism used in vanilla.

Table 1. Recovery Time with AS and vanilla in pod failures

	Fission Vanilla	Fission AS
Finonacci Function	2.840s	1.814s
Guestbook application	3.614s	1.528s

6.3 Experiments with node failure

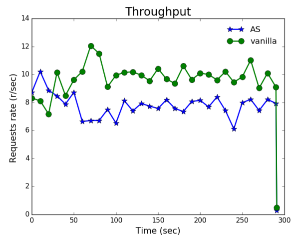
Figure 9 and 10 show the throughput, average response times, and HTTP code response rates of Fibonacci and Guestbook applications with Fission AS and vanilla, with node failures. In Figure 9(a) and Figure 10(a), we notice peaks in the throughput for both functions in vanilla. This can be explained as 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 9(b) and 10(b). However, in Fission AS the response rate is almost constant as the requests are just redirected to the standby instance. The recovery time is shown in Table 2. We can see that RT of Fibonacci and Guestbook functions under AS is 6.384s and 6.194s, respectively, whereas under vanilla is 3min7s and 2min39s, respectively. We clearly see that AS performs better than vanilla in terms of availability. Another observation is that vanilla tolerates better short, transient failures than long-lasting ones, such as node crashes.

6.4 Resource Consumption Analysis

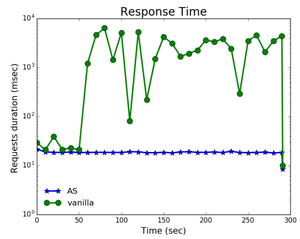
Figure 11 shows resource consumption (CPU, memory) of Fibonacci and Guestbook for AS and vanilla without and with failures (pod and node failures). We measured the overall CPU and memory usage of the 5 nodes during the execution of the workload in the scenarios without and with pod failures; we took measures of only 4 nodes in the node failure scenario (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 explained by the creation of two instances of each deployed application in Fission AS.

7 Conclusion and Future Work

In this work, we proposed an Active-Standby failover approach for FaaS platforms. We implemented this approach

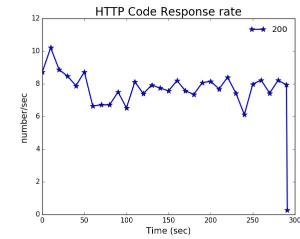


(a) Throughput



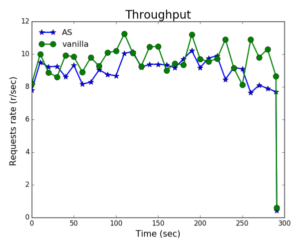
(b) Response time

(c) HTTP code response rate in vanilla

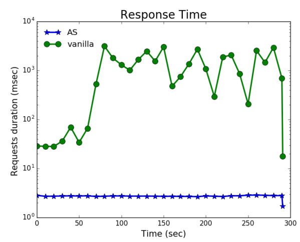


(d) HTTP code response rate in AS

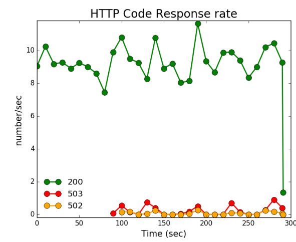
Figure 7. Fibonacci with pod failures



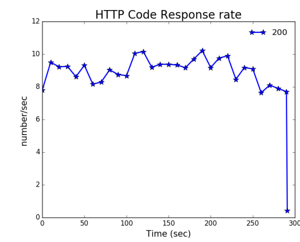
(a) Throughput



(b) Response time

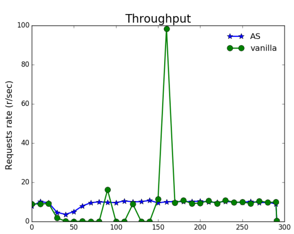


(c) HTTP code response rate in vanilla

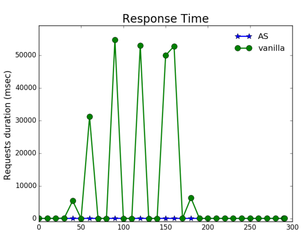


(d) HTTP code response rate in AS

Figure 8. Guestbook application with pod failures

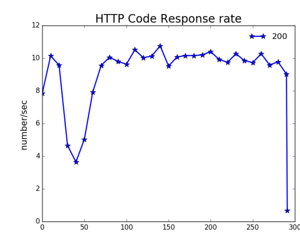


(a) Throughput



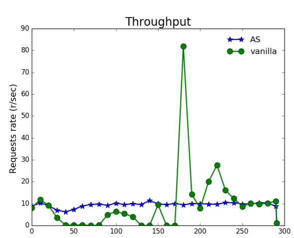
(b) Response time

(c) HTTP code response rate in vanilla

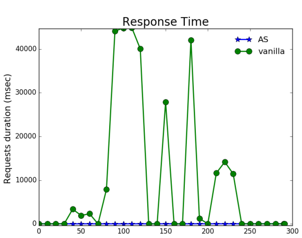


(d) HTTP code response rate in AS

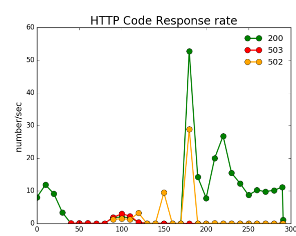
Figure 9. Fibonacci with node failures



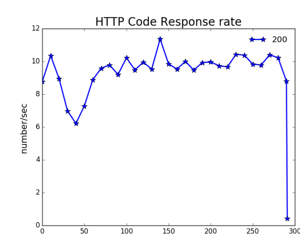
(a) Throughput



(b) Response time

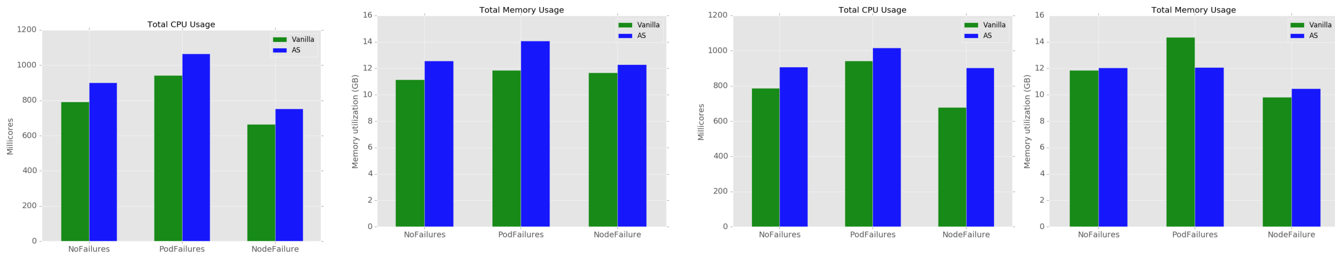


(c) HTTP code response rate in vanilla



(d) HTTP code response rate in AS

Figure 10. Guestbook application with node failures



(a) CPU consumption of Fibonacci

(b) Memory consumption of Fibonacci

(c) CPU consumption of Guestbook

(d) Memory consumption of Guestbook

Figure 11. CPU and memory consumption for Fibonacci and Guestbook application in Fission AS and vanilla without and with failures (pod and node failure)

Table 2. Recovery Time with AS and vanilla in node failures

	Fission Vanilla	Fission AS
Finonacci Function	3min7s	6.384s
Guestbook application	2min39s	6.194s

in an open source FaaS framework, called Fission, and we performed experiments to compare our approach to the retry-based approach implemented in the default (vanilla) version of Fission. The experiments showed that AS outperforms vanilla in terms of response time and availability while incurring a limited overhead in resource consumption.

In future work, we will investigate additional fault-tolerance techniques applicable in the FaaS context, such as checkpoint/restart, logging, or replication. We will implement and evaluate these techniques using real-life FaaS workloads. Our longer term goal is to design a smart, fault-tolerant system for FaaS that uses these techniques to automatically make the right trade-off among availability, performance and energy consumption.

8 Acknowledgments

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr)

References

[1] 2020. https://godoc.org/github.com/fission/fission/pkg/router. [Online; accessed 28-september-2020].

[2] Amazon Web Services 2020. AWS Lambda Features. https://aws.amazon.com/lambda/features/. [Online; accessed 18-september-2020].

[3] Amazon Web Services 2020. Error Handling and Automatic Retries in AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/invocation-retries.html. [Online; accessed 28-september-2020].

[4] AWS Admin 2019. 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 28-september-2020].

[5] AWS Admin 2020. Azure Functions geo-disaster recovery. https://docs.microsoft.com/en-us/azure/azure-functions/functions-geo-disaster-recovery. [Online; accessed 12-sept-2020].

[6] Tony Bourke. 2001. *Server load balancing.* " O'Reilly Media, Inc.".

[7] Logan Vadivelu Carol Hernandez, Eduardo Patrocinio and Marc Rodier. 2020. Architecting highly available cloud solutions. https://www.ibm.com/garage/method/practices/run/cloud-platform-for-ha. [Online; accessed 28-september-2020].

[8] Cloud design patterns 2020. Retry pattern. https://docs.microsoft.com/en-us/azure/architecture/patterns/retry. [Online; accessed 28-september-2020].

[9] Fission 2019. Fission. https://docs.fission.io/docs/. [Online; accessed 28-september-2020].

[10] Google cloud functions 2019. Retrying Background Functions. https://cloud.google.com/functions/docs/bestpractices/retries. [Online; accessed 28-september-2020].

[11] Grid5000 2020. Grid5000. https://www.grid5000.fr/w/Grid5000:Home. [Online; accessed 28-september-2020].

[12] Guestbook App [n.d.]. Kubernetes. https://github.com/fission/fission/tree/master/examples/python/guestbook. [Online; accessed 28-september-2020].

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

[14] Kubernetes [n.d.]. Kubernetes. https://kubernetes.io/. [Online; accessed 28-september-2020].

[15] Kubernetes Readiness Probe [n.d.]. Configure Liveness, Readiness and Startup Probes. https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/. [Online; accessed 28-september-2020].

[16] OpenFaaS 2019. Timeouts - Asynchronous invocations. https://docs.openfaas.com/deployment/troubleshooting/#timeouts-asynchronous-invocations. [Online; accessed 28-september-2020].

[17] Powerfulseal [n.d.]. Powerfulseal. https://github.com/bloomberg/powerfulseal. [Online; accessed 28-september-2020].

[18] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. 2020. A Fault-Tolerance Shim for Serverless Computing. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/3342195.3387535

[19] Colby Tresness. 2020. Understanding serverless cold start. https://azure.microsoft.com/en-us/blog/understanding-serverless-cold-start/?ref=msdn. [Online; accessed 28-september-2020].

[20] Tsung [n.d.]. Tsung. http://tsung.erlang-projects.org/user_manual/. [Online; accessed 28-september-2020].