Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH

Open Access

# Integrating request replication into FaaS platforms: an experimental evaluation

Yasmina Bouizem[1,2], Djawida Dib[1*], Nikos Parlavantzas[3] and Christine Morin[2]

**Abstract**

Function-as-a-Service (FaaS) is a popular programming model for building serverless applications, supported by all major cloud providers and many open-source software frameworks. One of the main challenges for FaaS providers is providing fault tolerance for the deployed applications, that is, providing the ability to mask failures of function invocations from clients. The basic fault tolerance approach in current FaaS platforms is automatically retrying function invocations. Although the retry approach is well suited for transient failures, it incurs delays in recovering from other types of failures, such as node crashes. This paper proposes the integration of a Request Replication mechanism in FaaS platforms and describes how this integration was implemented in Fission, a well-known, open-source platform. It provides a detailed experimental comparison of the proposed approach with the retry approach and an Active-Standby approach in terms of performance, availability, and resource consumption under different failure scenarios.

**Keywords** Cloud, Serverless, FaaS, Fault tolerance, High availability

## Introduction

Serverless computing is an increasingly popular model for developing and running cloud applications [1, 2]. With serverless computing, developers are isolated from the details of infrastructure management and are able to focus on the business logic of their applications. At the core of serverless computing is the Function-as-a-Service (FaaS) programming model in which the unit of computation is a function. Developers provide the function code, and the FaaS platform automatically manages resource provisioning and function execution. Several FaaS platforms are commercially available, such as Amazon Lambda [3], Google Functions [4], and Azure functions [5], or distributed in open source, such as Fission [6], OpenFaaS [7], Kubeless [8], and OpenWhisk [9].

*Correspondence:
Djawida Dib
d.djawida@gmail.com
[1] Department of Computer Science, University of Tlemcen, Tlemcen, Algeria
[2] Inria Centre at Rennes University, Rennes, France
[3] INSA Rennes, Rennes, France

A key challenge in running applications on FaaS platforms is ensuring fault-tolerance for the deployed functions. Fault tolerance for FaaS refers to the ability of the system to continue serving function requests despite infrastructure failures, such as hardware failures, virtualization software failures, and network failures. Fault tolerance is essential for ensuring high availability in FaaS deployments. High availability and built-in fault tolerance are promoted as essential features of commercial FaaS platforms (e.g., [3]). Most current FaaS platforms support a single fault tolerance approach that involves retrying function executions [4, 6, 10–13]. However, while the retry approach allows coping with transient failures such as temporary loss of network connectivity, it incurs delays in recovering from other kinds of failures such as node failures.

In the work described in the present paper, we propose to integrate an active replication [14] approach in FaaS frameworks in order to make failures transparent to the applications. The proposed approach consists in replicating function requests and is implemented in Fission, a popular open-source FaaS framework. This work extends our previous work [15] describing the integration

of an Active-Standby fault tolerance approach in FaaS platforms. Specifically, the current work proposes a new fault-tolerance approach, provides an extensive experimental evaluation of the two approaches along with the retry approach, and discusses the lessons learned from this evaluation.

The proposed approach along with the Active-Standby and retry approaches assume that functions are or can be converted to become idempotent, which means that the functions produce the same results when executed multiple times with the same input. This is the typical assumption made by current FaaS platforms. Converting non-idempotent code to become idempotent may be challenging, especially when the code interacts with external, non-idempotent services, but FaaS providers give guidelines for performing this conversion (e.g., [16, 17]).

Moreover, the proposed approach along with the Active-Standby approach maintain functions continuously running and thus using resources even if they receive no traffic. This resource cost is only paid for functions that require high availability. Current commercial FaaS platforms also include features that maintain functions continuously running, such as the provisioned concurrency feature in AWS Lambda [18]. The motivation for that AWS feature, however, is to reduce start-up latency rather than provide fault-tolerance.

This paper brings the following novel contributions:

- Study of the integration of an active replication fault tolerance approach (called Request Replication) in a FaaS environment;
- Implementation of the approach in the Fission FaaS platform (Fission Request Replication);
- Comparative evaluation according to several metrics of Fission Request Replication, Fission Vanilla (native retry approach), and a new version of Fission Active-Standby (enhanced implementation of the fault tolerance approach proposed in [15]), using a computational application both in normal functioning and in various failure scenarios, including instance and node failures and network delays;
- Insights on how to select a fault tolerance approach according to the application type and user requirements in terms of performance, resource consumption, and availability.

The remainder of the paper is organized as follows. Section "Related Work" discusses related work. Section "Fission FaaS Framework" presents Fission, a representative, open-source FaaS platform, which we used for implementing and evaluating our proposed fault tolerance approach. Section "Existing fault tolerance mechanisms in Fission" describes two existing fault tolerance approaches and their implementation in Fission; namely, the retry approach natively implemented in Fission, and the Active-Standby approach that we proposed in [15]. Section "Request Replication for FaaS" presents the Request Replication approach in the context of FaaS platforms and its implementation in Fission. Section "Experimental Setup" is devoted to the experimental setup, and Section "Experimental Results" analyses the experimental evaluation results. We expand on lessons learnt in Section "Lessons Learned" and conclude in Section "Conclusion and Future Work".

## Related work

A wide range of approaches have been applied to support fault tolerance in cloud systems [19]. In the following, we only consider work related to fault tolerance in serverless systems. The basic fault tolerance mechanism in current commercial and open-source FaaS platforms is automatically retrying invocations. All major commercial platforms, such as AWS Lambda [10, 11], Google Cloud Functions [4] and Microsoft Azure Functions [12], provide automatic retry functionality to handle failures and timeouts. For instance, AWS Lambda retries asynchronous invocations up to two times with a delay between such retries. Some open-source FaaS platforms also support the retry mechanism, including Fission and OpenFaaS, which retry asynchronous invocations with an exponential back-off [13]. Our work considers fault tolerance mechanisms beyond automatic retry.

Fault tolerance in serverless systems can also be realised through using additional services provided by cloud platforms. For instance, using Azure load-balancing and event ingestion services, developers can deploy functions in different regions to allow for disaster recovery. The functions can be deployed using an active-active or an active-passive configuration [20]. Using serverless orchestration services (such as Google Workflows [21], AWS Step Functions [22], or Azure Durable Functions [23]), 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 [24]. Similar capabilities are provided by open-source orchestration frameworks, such as Apache OpenWhisk Composer [25] or Faas-flow for OpenFaaS [26]. Our work focuses on fault tolerance mechanisms implemented within FaaS platforms without involving external services.

Recent research works investigate fault tolerance for stateful serverless applications, composed of multiple functions and interacting with storage services. Sreekanti

et al. [27] introduce a layer that lies between standard FaaS platforms and key-value databases to ensure atomic visibility of storage updates. The proposed system assures fault tolerance by enforcing the read atomic consistency guarantee. Zhang et al. [28] describe a library and runtime for building transactional, fault-tolerant workflows on existing serverless platforms. The system supports transactions within and across functions through applying a log-based fault tolerance approach. Jia et al. [29] propose 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. [30] present HydroCache, a distributed cache layer for FaaS systems, which provides transactional causal consistency for stateful functions. The system relies on Anna storage [31], 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. Node failures are detected using a heartbeat mechanism and unfinished functions of the failed node are re-scheduled on another node. Unlike those systems, our work focuses on ensuring fault tolerance for individual, idempotent functions rather than for stateful function compositions, and does not impose the use of additional APIs on FaaS developers.

Another recent work [32] introduces a programming model and associated implementation for supporting transactions across stateful FaaS functions. This work builds on Apache Flink StateFun [33], an open-source platform for stateful FaaS functions that uses a streaming dataflow engine. The platform deals with failures via checkpointing/snapshots to achieve exactly-once-processing guarantees. The StateFun programming model supports encapsulating state within function instances, which is not allowed in the typical FaaS model.

Zhang et al. [34] propose Kappa, a programming framework for building parallel serverless applications. The framework periodically checkpoints function results in order to enable failure recovery. Carver et al. [35] present Wukong, a framework for building parallel FaaS applications on top of AWS Lambda. In case of failure, the automatic retry mechanism of AWS Lambda is used to re-execute the failed function. Both these systems propose libraries built on top of an unmodified FaaS platform (AWS Lambda) while our mechanisms are integrated within the FaaS platform.

Karhula et al. [36] propose using 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. Although these mechanisms could be used as building blocks for a fault-tolerant FaaS system, this work does not provide a complete, practical implementation of such a system.

In summary, there are several solutions that propose fault tolerance mechanisms beyond the basic retry mechanism. However, all such solutions require the use of APIs and primitives outside those provided by the core FaaS model, which only supports invoking functions in response to events. For instance, these solutions require the use of load balancing services [20], workflow orchestration services [24], or specialized, state-aware programming models [27–30, 32]. To the best of our knowledge, our work is the first to integrate fault-tolerance mechanisms into the FaaS platform without deviating from the core FaaS model and thus without adding complexity for developers.

## Fission FaaS framework

In this section, we present Fission [6], the FaaS platform used in all the experiments presented in this paper. We selected Fission because it is representative of existing FaaS platforms. Indeed, as the vast majority of such platforms [37], Fission is built on the Kubernetes [38] container orchestrator, and is developed in Go [37]. Moreover, Fission is one of the most popular open-source FaaS platforms [37].

The core Fission components are: Function Pods, Router, and Executor (see Fig. 1). Function Pods contain function-specific containers to serve requests coming from users, called function calls.

The Router receives the function call (message 1 in Fig. 1) and checks if a corresponding function pod is running. If no corresponding function pod is running, the Router requests the creation of a new one from the Executor (message 2 in Fig. 1). There are two types of Executor: PoolManager and NewDeploy. The PoolManager Executor maintains a pool of warm generic pods in order to provide low cold start latencies [39] and start functions quickly (message 2.a in Fig. 1). After creating the function pod, the PoolManager Executor sends the Pod's IP address to the Router (message 3.a in Fig. 1). Then, the Router forwards the function call to the pod (message 4.a in Fig. 1). The PoolManager Executor does not allow using multiple pods per function, which limits its scaling during high traffic. The NewDeploy Executor allows creating multiple pods per function along with a Kubernetes service to load balance the requests between the function pods (message 2.b in Fig. 1). The NewDeploy Executor also uses a Horizontal Pod Autoscaler (HPA) to automatically adjust the number of pods to match the traffic. After creating the function pod(s), the NewDeploy Executor sends the IP address of the corresponding Kubernetes service to the Router (message 3.b in Fig. 1). The Router forwards the function call to the Kubernetes

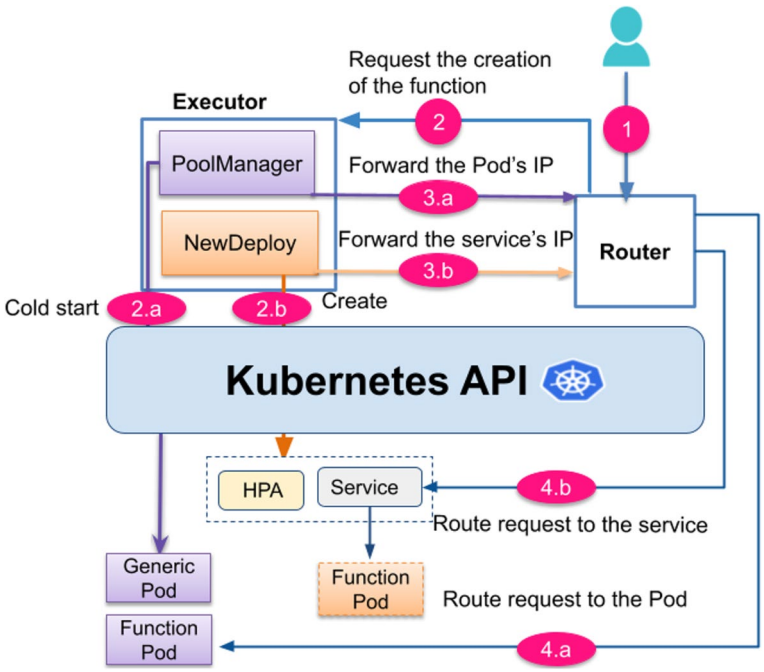


**Fig. 1** Simplified Fission Architecture

service which routes it to the corresponding pod(s) (message 4.b in Fig. 1). If a running function pod exists at the time of receiving a function call, the Router forwards it either to the corresponding Pod's IP address (message 4.a in Fig. 1) or to the corresponding Kubernetes service's IP address (message 4.b in Fig. 1), depending on the Executor used to create the running function pod.

## Existing fault tolerance mechanisms in Fission

We present in this section two fault tolerance mechanisms implemented in Fission: the native retry mechanism available in most FaaS platforms and the Active-Standby mechanism, an enhanced version of the one proposed in our previous work [15].

### Retry

Retry is the native fault tolerance approach in Fission and consists basically in restarting the entire submission process of a failed request. The retry mechanism used in Fission works as shown in Fig. 2 assuming a NewDeploy Executor. When a function call is received, the Router forwards it to the corresponding function pod, as described in Section "Fission FaaS Framework". The Router then sends the request to the Kubernetes service which forwards it to the function pod. If the function execution fails due to network timeout errors, the Router tries 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 [40]. If all the retries fail or the error is a network dial error, Fission assumes that the function pod does not exist anymore. Thus, the Router asks the Executor for a new service for the function. The Router tries to forward the function call to the new Kubernetes service and so on until the request is served. The Router relays to the user any errors beyond network timeout and network dial errors.

### Active-Standby

In the context of FaaS, the Active-Standby approach consists in creating two function instances. The first is active and serves incoming requests while the second is passive (on standby).

Each instance monitors the connectivity of the other instance using heartbeats. If no heartbeat is received from an instance within a given time interval, the instance is considered as unreachable. If the passive instance is unreachable, another passive instance is created. If the active instance is unreachable, the passive instance is activated and a new passive instance is created.

#### *Implementation in Fission*

To implement the Active-Standby approach in Fission, we use the NewDeploy executor because 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 [41] that indicates when the container is ready to receive requests.

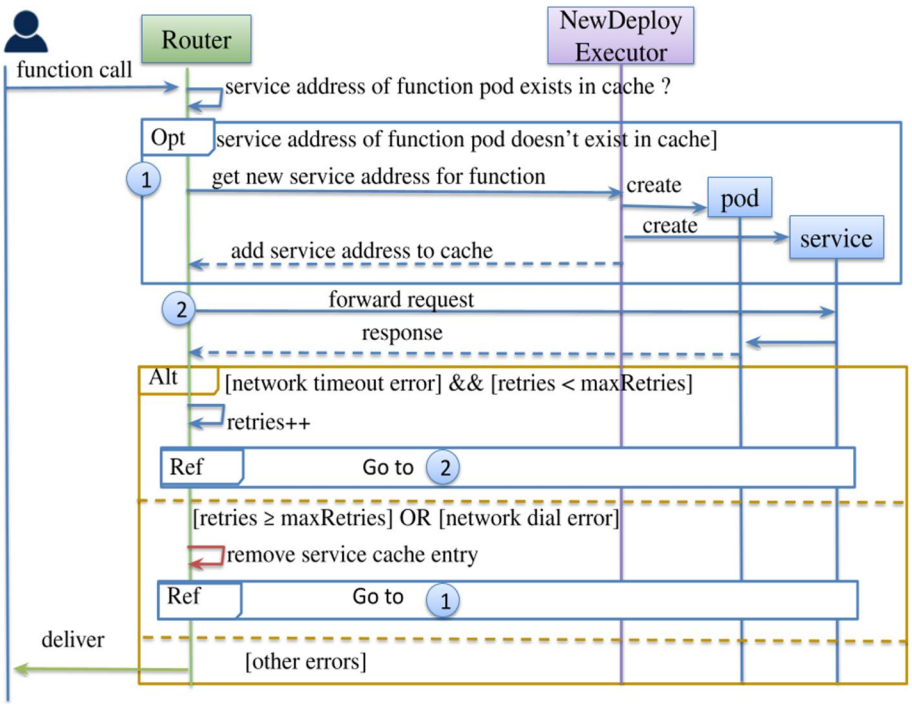


**Fig. 2** Fault tolerance protocol with the Retry mechanism

The active pod declares the ready state and can receive and serve traffic. The passive pod declares the not-ready state, and no traffic is forwarded to it.

We implemented a new Router, called Active-Standby Router (Router AS), which allows routing requests only to the Active pod, and use this instead of the default Fission Router. The Router AS does not support the retry mechanism. Once a request is received by the Router AS (message 1 in Fig. 3), it needs to be executed specifically by the active pod. In this case, the Router AS gets

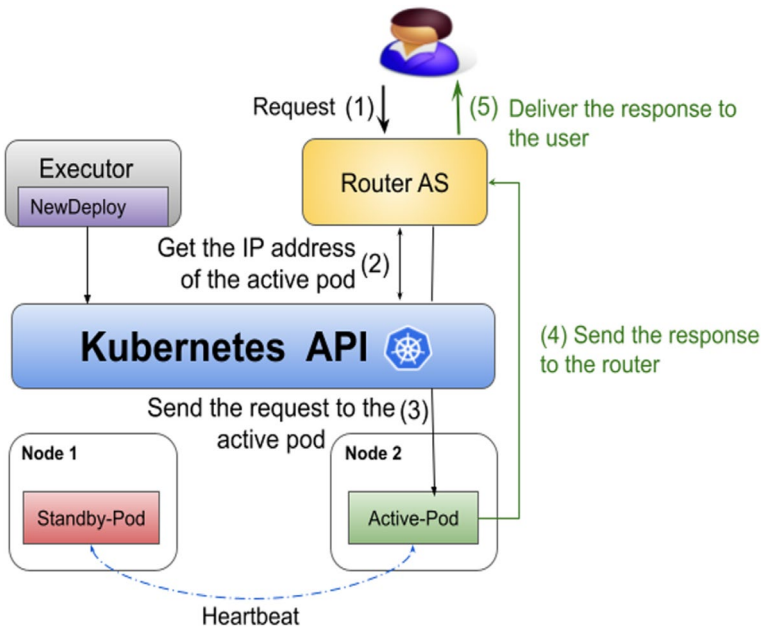


**Fig. 3** Overview of the Active-Standby mechanism in Fission

the IP address of the active pod from the Kubernetes API (message 2 in Fig. 3) and forwards the request directly to this address (message 3 in Fig. 3 ). The Router receives the response (message 4 in Fig. 3) and sends it to the user (message 5 in Fig. 3).

Figure 4 offers a more comprehensive view of the implementation of the Active-Standby mechanism in Fission. While the request is being processed, both active and standby pods send and receive heartbeats to and from each other for health checks. The heartbeats are sent every second (the minimum configurable value using Kubernetes Readiness probes). While the active pod is running, the passive pod fails the readiness probe and remains in the not-ready state. If the active pod fails, the passive pod passes the readiness probe and becomes active. A new pod is then created to take the place of the previously passive pod. Similarly, if the passive pod fails, a new pod is created to take its place.

## Request Replication for FaaS

In this section, we present the Request Replication fault tolerance approach and its implementation in Fission.

### Request Replication principle

Request Replication consists in having K replicas process a request at the same time. The number of replicas depends on the required 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 replica failures.

### 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 implemented one, called Router Request Replication (Router RR). This Router submits requests to all function pod replicas at the same time, and it does not support the retry mechanism. When the Router RR receives a request (message 1 in Fig. 5), it retrieves the IP addresses of the function pod replicas from the Kubernetes API (message 2 in Fig. 5). Then, it uses these IP addresses to replicate each received request on all function pod replicas in order to be processed in parallel (message 3 in Fig. 5). Then, the responses are sent to the router (message 4 in Fig. 5) and the first received response is sent to the user (message 5 in Fig. 5). To tolerate K failures using this approach, it is necessary to have a minimum of K+1 replicas, distributed across different nodes. The

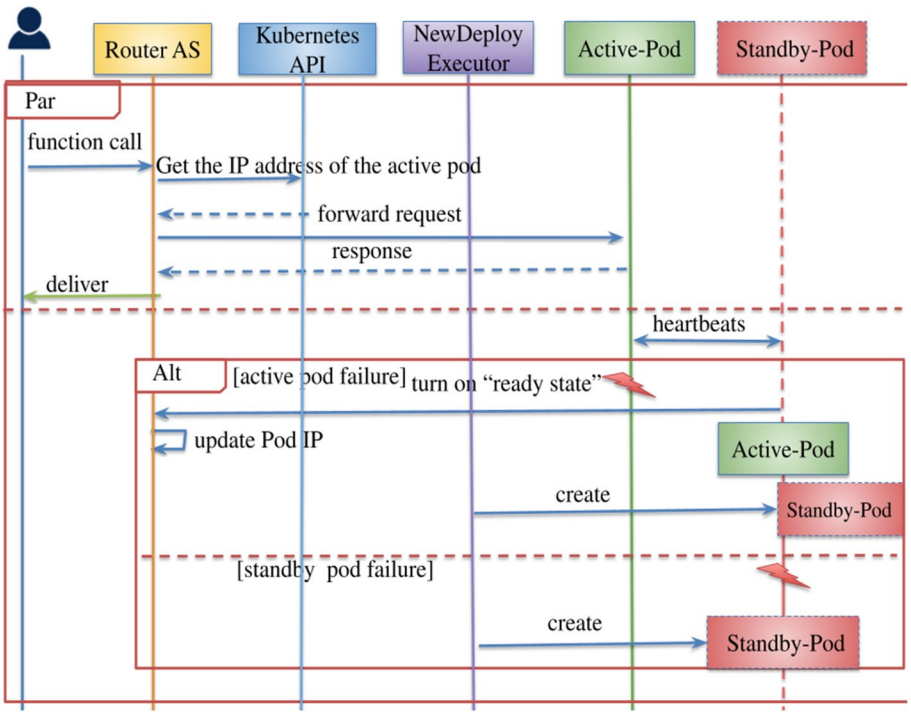


**Fig. 4** Fault tolerance protocol with the Active-Standby mechanism

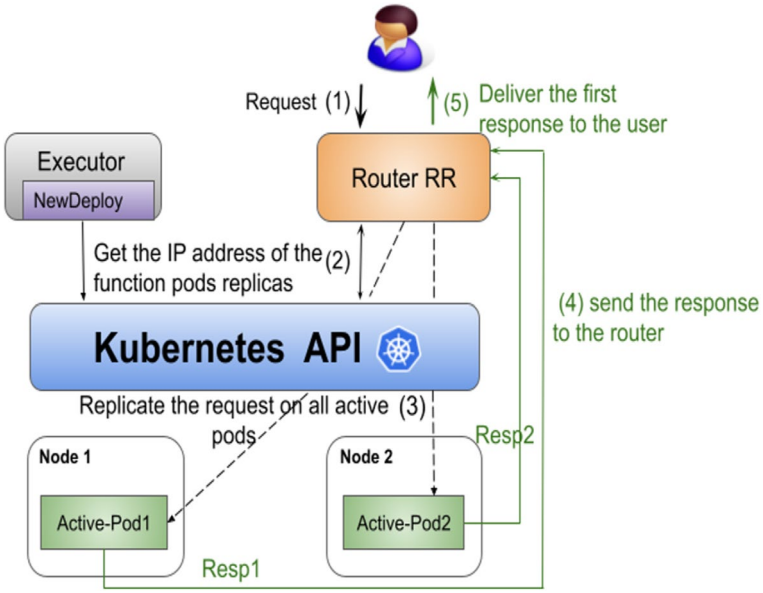


**Fig. 5** Overview of the Request Replication mechanism in Fission

pod anti-affinity mechanism provided by Kubernetes is used to guarantee this distribution. Figure 6 offers a more comprehensive view of the implementation of the Request Replication approach in Fission.

## Experimental setup

In this section we describe the experimental setup for evaluating the effectiveness of the proposed RR fault tolerance approach and comparing it with the retry and the AS approaches in the context of their implementation in Fission. In all experiments, the RR approach uses two replicas, which is sufficient to demonstrate the differences among the three approaches. Increasing the number K of replicas would be useful for services that require the highest level of availability (e.g., emergency response systems, medical services).

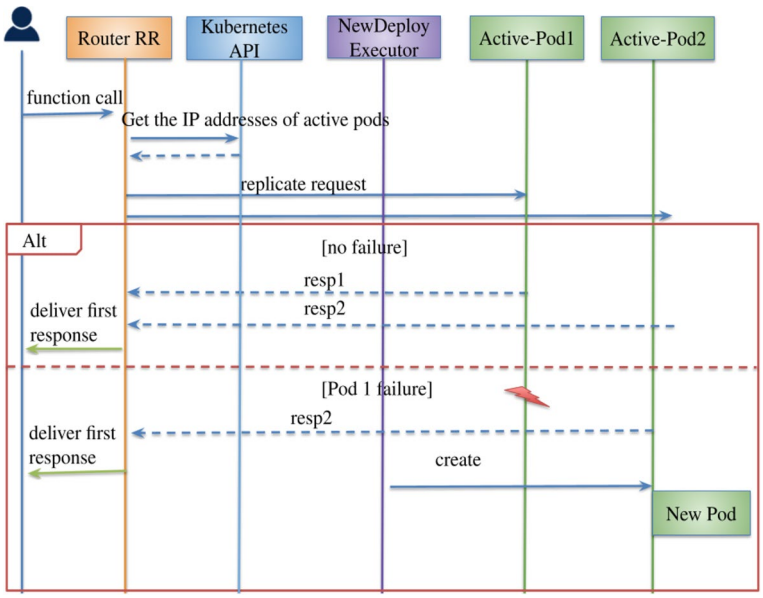


**Fig. 6** Fault tolerance protocol with the Request Replication mechanism

#### Environment

Our experiments were performed on Grid'5000 [42], a configurable testbed distributed over different sites in France with the goal to support experiment-driven research in parallel and distributed computing. Specifically, we used 5 nodes on the Lyon site (nova cluster), each node having 2 Intel Xeon E5-2620 v4 CPUs with 8 cores/CPU and 64 GB memory, to deploy Kubernetes [43] (version 1.19). In our cluster we have one node for the Kubernetes master and four worker nodes. One worker node contains the components of Fission AS (Active-Standby), Fission RR (Request Replication) or the original version of Fission (vanilla), depending on the experiment. The three other worker nodes are used to host the function pods. For each experiment we use either RR, AS or vanilla with version 1.10.0 (the latest stable release of Fission at the time we run our experiments). We set up 2 additional nodes; one used as client to invoke functions and one used to inject faults.

#### Applications

We used 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). Our function takes n=15 as input, computing the 15th term of the sequence. We chose this simple function because it demonstrates in the most clear and concise manner the differences between the three approaches. Indeed, since the function is compute-bound and does not rely on any data storage or external services, it becomes easier to isolate and analyze the effect of the FT approach on performance, resource consumption, and availability.

#### Workload

The workload is generated with Tsung [44], a high-performance benchmark framework. In our test, we generated 60000 requests during 10 minutes with 100 concurrent users created every second (corresponding to an input data rate of 100 requests/sec).

#### Failure scenarios

We defined three failure scenarios:

- Pod failure: where we inject a fault into a specified application to make it unavailable for a period of time.
- Node failure: where we kill the node hosting the application.
- Network delay: where we inject latency to see the impact of network delay on the deployed application. This scenario is executed to see how the three approaches react to network issues.

In the pod failure and network delay scenarios, we use the Chaos Mesh tool [45] to inject faults to pods. In the first scenario, the failure is simulated by killing the function pod at the 5th minute from the beginning of the workload execution. In the second scenario, the failure is simulated by killing the node hosting the function instance 5 minutes after the beginning of workload execution. In the third scenario, we inject latency at the 5th minute for a duration of 10 seconds. The injected latency values are 50ms, 100ms and 200ms. Note that the injected latency causes a delay for all responses coming from the function pod. 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 has been repeated at least 5 times for each version of Fission (i.e., vanilla, AS and RR). The averages of the measurements are shown in the illustrated results.

#### Metrics

We evaluate our solution using three categories of metrics:

- Performance: Performance is measured using throughput (i.e, number of requests per second) and response time (i.e., the time between sending a user request and receiving the system response).
- Availability: Availability is measured using recovery time, i.e., the time between the first reaction to a failure and the resumption of the service. We also measure the error rate, that is, the proportion of failed requests (having HTTP 5xx response code).
- Resource consumption: Resource consumption is measured using CPU and memory usage of the 5 nodes for the applied workload.

## Experimental results

This section presents the results obtained from the experimental comparison of our proposed RR approach with the existing AS and retry approaches.

#### Performance Results

##### *Results with no failures*

Figure 7 shows the response time for Fission AS, vanilla and Fission RR with no failures. In this figure we can notice that Fission RR is slightly faster than Fission AS and vanilla because once it receives the first response from one of the function replicas, it forwards it to the user. Vanilla is slightly slower. This can be explained by the use of the Kubernetes service to send the request to

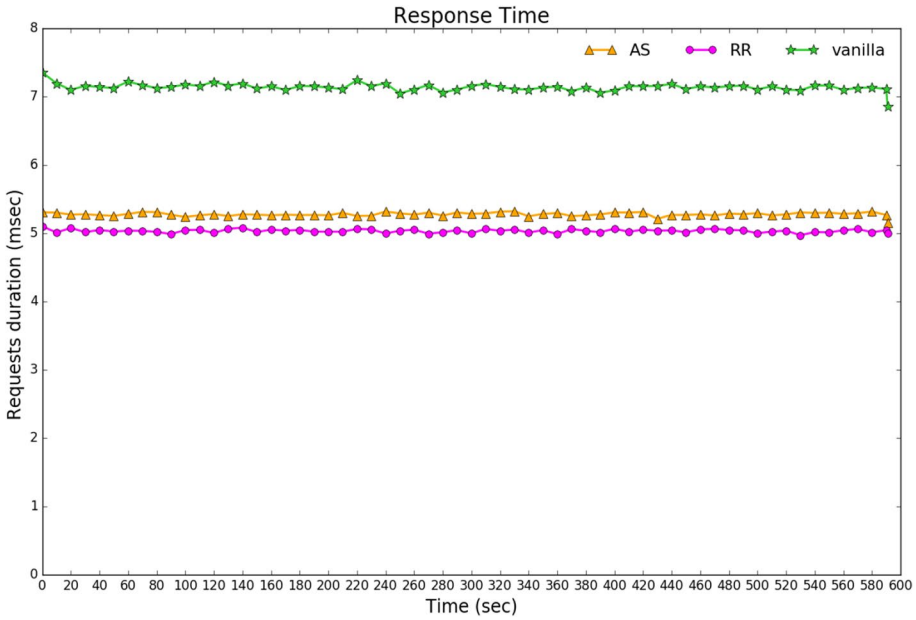


**Fig. 7** Response time of AS, vanilla and RR with no failure

the pod belonging to the function, which adds another hop compared to AS and RR. AS, vanilla and RR handle the same throughput with values around 100 requests/sec, which is the expected throughput. As a conclusion, we see that RR performs better than AS and vanilla in terms of response time when there are no failures.

***Results with failures***

1 **Pod Failure Scenario** Figures 8 and 9 illustrate the throughput and response time of AS, vanilla and RR with a pod failure. In Fig. 8, 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.

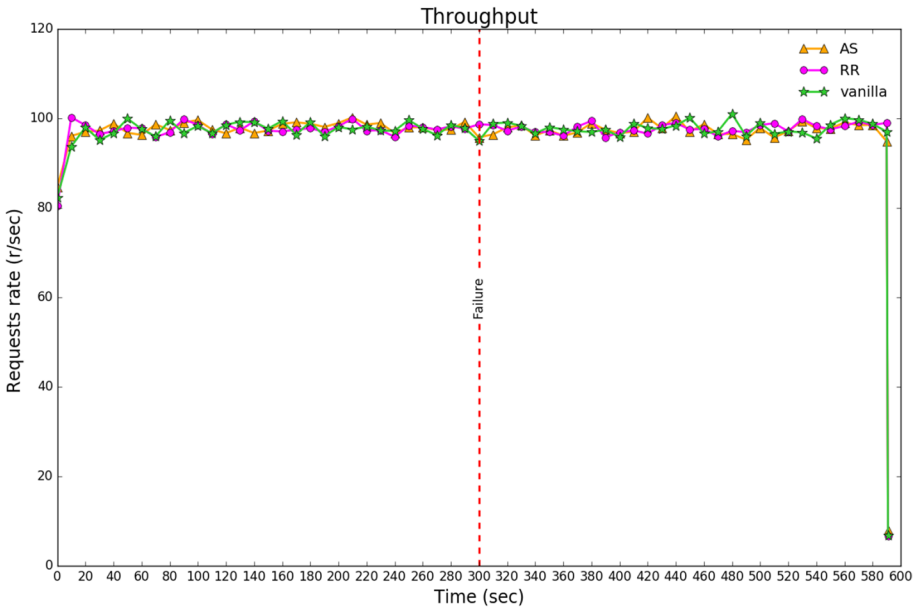


**Fig. 8** Throughput of AS, vanilla and RR with pod failure

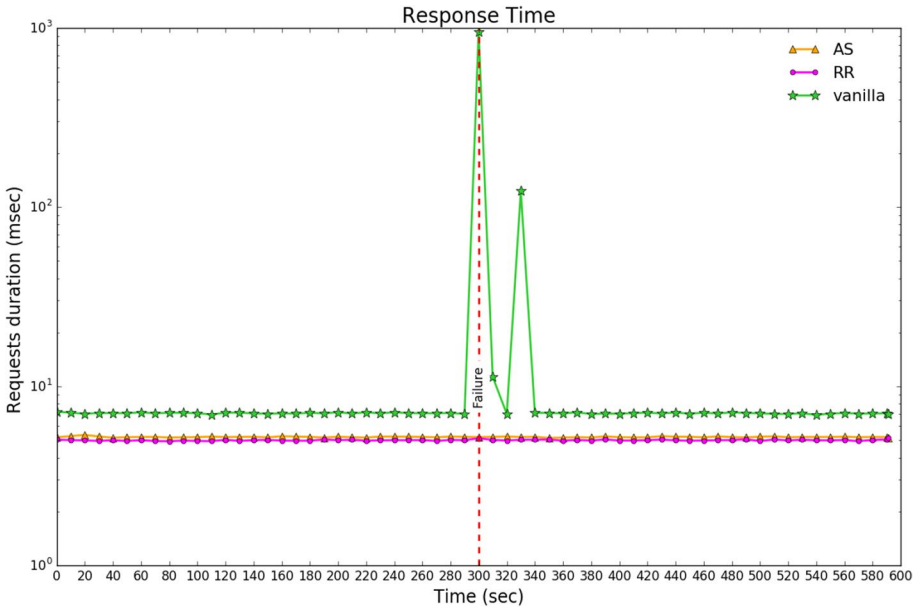


**Fig. 9** Response time of AS, vanilla and RR with pod failure

We also notice a degradation in the throughput of vanilla when the pod fails at 300s, as no pod is available to serve the requests. RR provides stable throughput despite the pod failure since all the traffic is executed by the healthy replica. In Fig. 9, we notice some spikes in the response time of vanilla during almost 30 seconds. This is attributed to 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 7ms. In contrast, RR and AS provide stable response times with values around 5ms.

2 **Node Failure Scenario** Figures 10 and 11 present performance results of AS, vanilla and RR with a node failure. Figure 10 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 active. In vanilla,

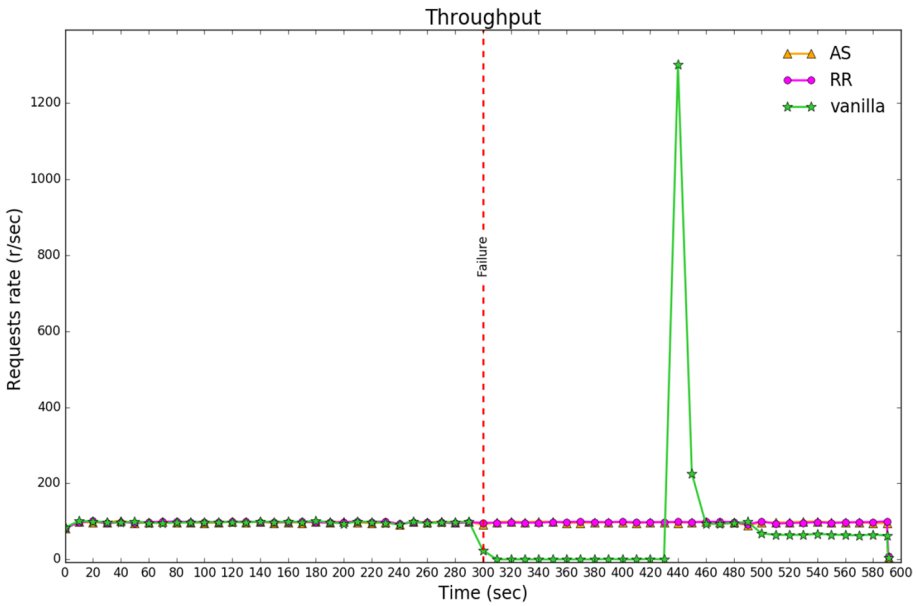


**Fig. 10** Throughput of AS, vanilla and RR with node failure

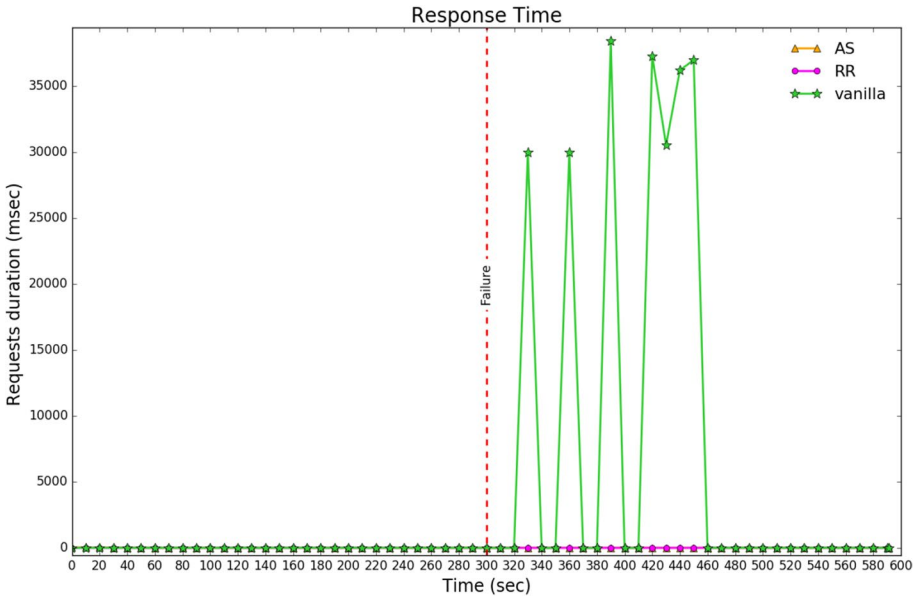


**Fig. 11** Response time of AS, vanilla and RR with node failure

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 healthy replica that continues to process the user's requests. Figure 11 shows spikes in the latency of vanilla. This is because the router retries many requests, where the wait time is increased exponentially after each attempt. We assume 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.

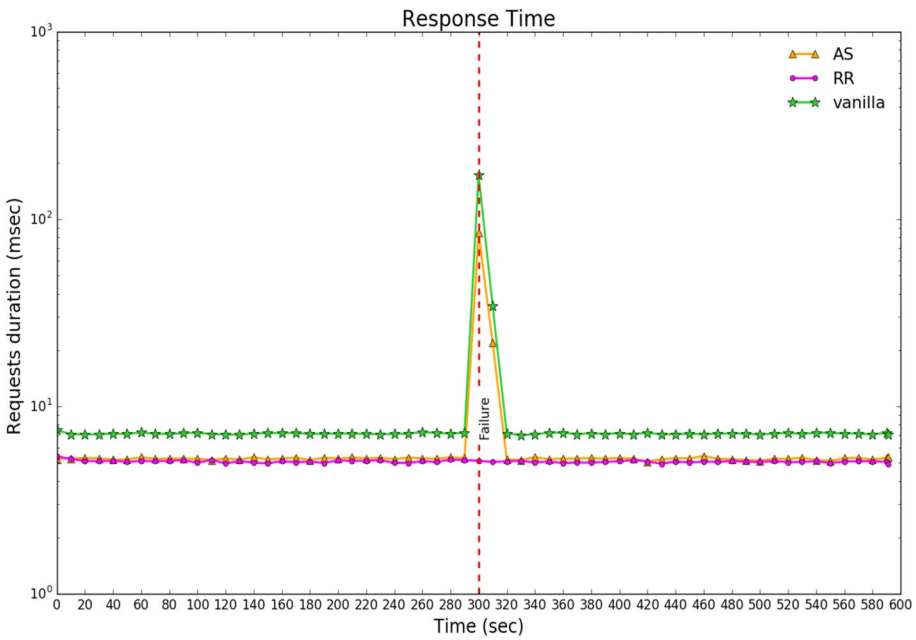


**Fig. 12** Response time with 50ms of latency

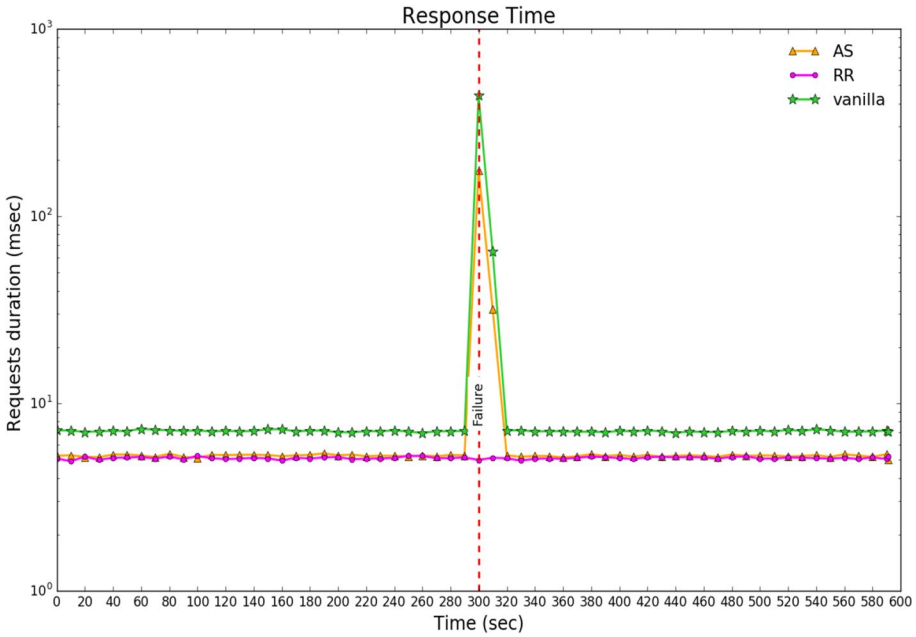


**Fig. 13** Response time with 100ms of latency

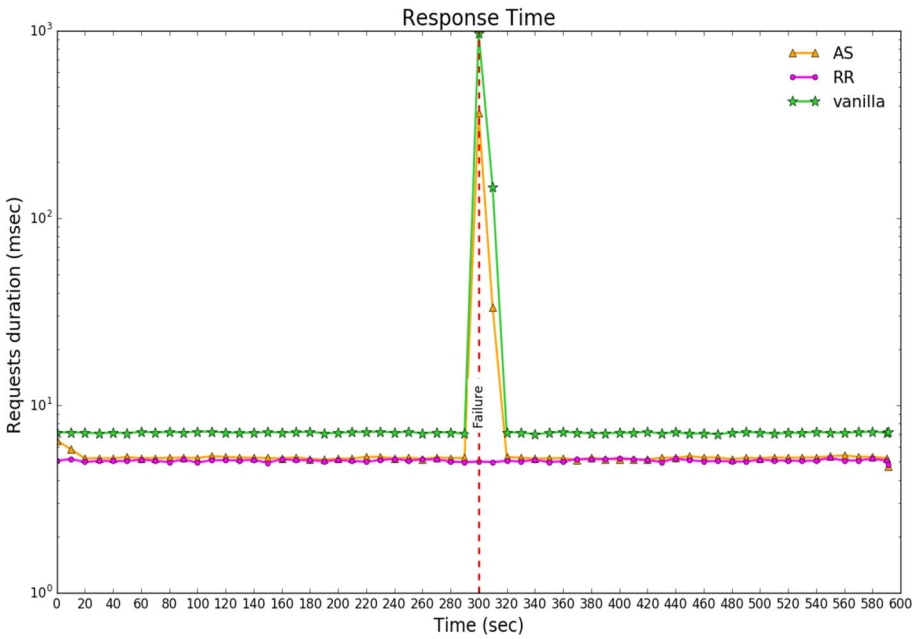


**Fig. 14** Response time with 200ms of latency

3 **Network Delay Scenario** We injected separately three latency values: 50ms, 100ms and 200ms. Figures 12, 13, and 14 show the response time of AS, vanilla and RR with the injected latency values. When we increase the latency value, 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 Figs. 13 and 14). 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 a single replica as this latency is masked by the fast response of the other replica.

## Availability results

### *Recovery time*

The recovery time is the time required for a service to recover from failures and become available again. This covers the time between failure detection and the time when the service becomes fully operational. It is measured for the three approaches as follows: For vanilla, after the failure, the pod becomes unhealthy (see Fig. 15). 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 Fig. 16). The reaction is the failover to the standby pod and the update of the router cache. Once the router cache is updated with the IP address of the active pod (Active-IP), the service becomes available.

For RR, no recovery is necessary as the failure of one of the replicas does not affect service availability (see Fig. 17). The service remains available because the second pod replica serves the requests.

1 **Pod Failure Scenario** As seen in Table 1, in the pod failure scenario, the measured recovery time for AS is significantly lower than for vanilla. The reason is that with AS, there is already a standby pod, and the service is recovered as soon as the standby detects the failure of the active pod. In contrast, recovery with vanilla depends on the repair of the failed pod. For RR, the second replica continues to serve requests. Therefore, for this approach, recovery time is zero.
2 **Node Failure Scenario** In the node failure scenario (see Table 1), the recovery time for vanilla is significantly higher than that for AS and RR. AS takes seconds to recover from a node crash, whereas vanilla takes more than 2 minutes. This is explained by the

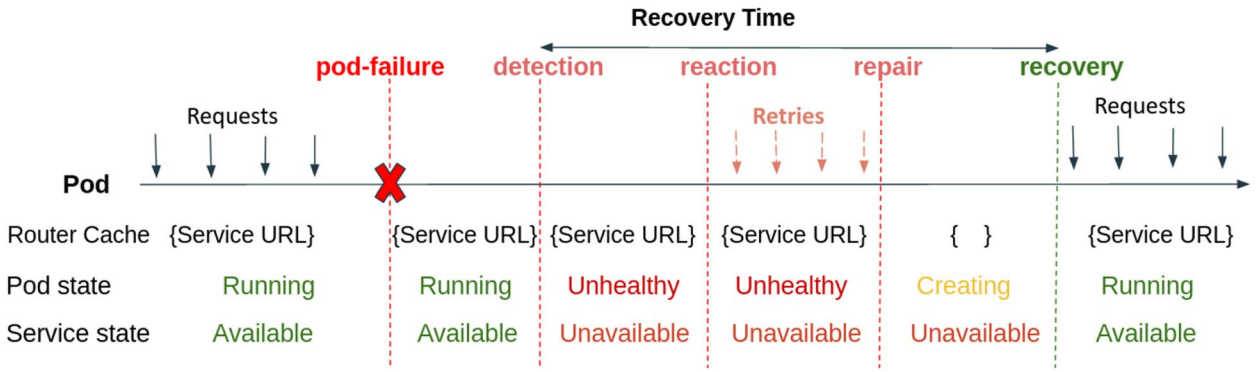


**Fig. 15** Recovery time in vanilla

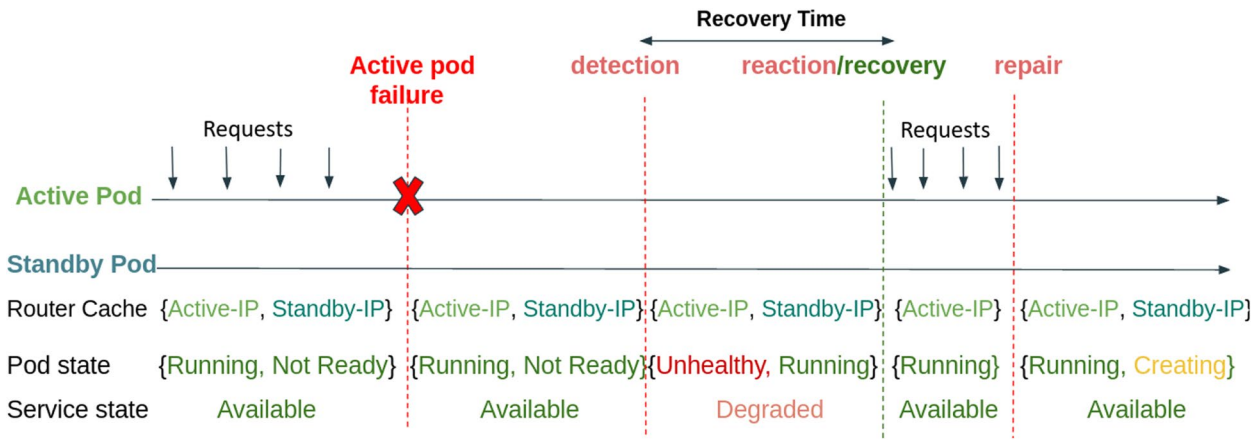


**Fig. 16** Recovery time in AS

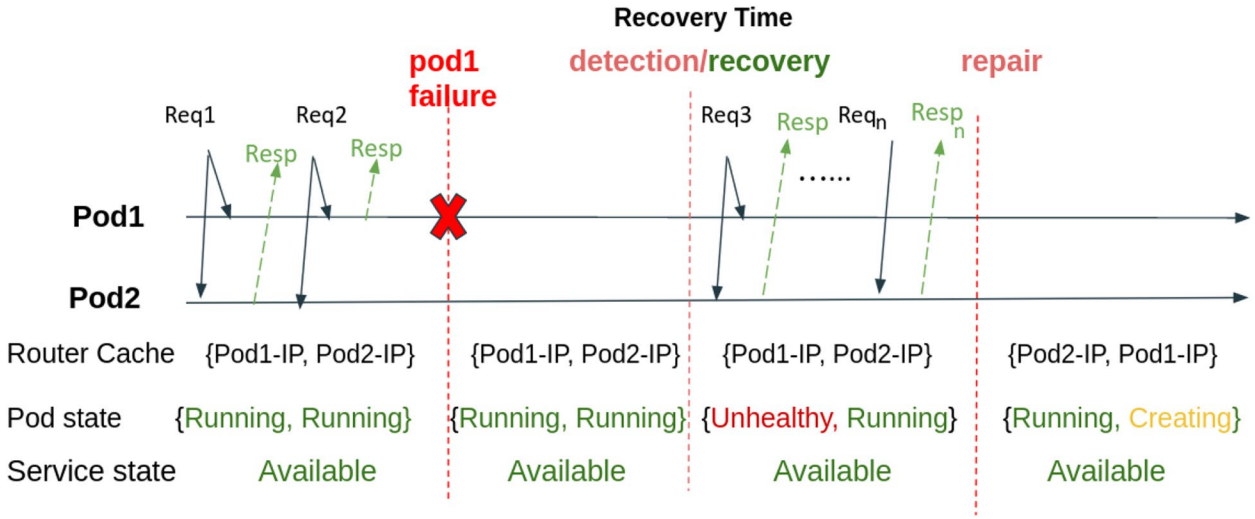


**Fig. 17** Recovery time in RR

**Table 1** Recovery Time with vanilla, AS, and RR in pod and node failure scenarios

| Failure scenario | Fission vanilla | Fission AS | Fission RR |
|---|---|---|---|
| Pod failure | 7s | 1.81s | 0s |
| Node failure | 2m19s | 2.80s | 0s |

**Table 2** Error rate for vanilla, AS, and RR in pod and node failure scenarios

| Failure scenario | Fission vanilla | Fission AS | Fission RR |
|---|---|---|---|
| Pod failure | 0.01% | 0% | 0% |
| Node failure | 1.26% | 0% | 0% |

behavior of each approach. With AS when the node hosting the active replica fails, the passive replica detects the failure of the active replica and starts processing the requests. RR recovery time is zero, since another active replica is running in parallel and continue processing the requests.

#### *Error rate*

In this section, we present the error rate of Fission vanilla, AS, and RR with pod and node failures. This metric is a useful measure to evaluate the user perception when using the approaches under different failure scenarios. To measure the rate of requests that fail, we count the number of requests that return an HTTP status with a response code of 5xx (it means the request cannot be fulfilled due to a server error). 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** As seen in Table 2 and Fig. 18, in the pod failure scenario, Vanilla has a 0.01% error rate (i.e., some HTTP requests failed with code 503 service unavailable) which indicates that the router is unable to handle the request due to a temporary overload. The error rate for AS and RR is 0% (i.e., all requests succeeded with a response code of 200) which means that all requests have been successfully served.

2 **Node Failure Scenario** In the node failure scenario (see Table 2 and Fig. 19), the error rate for vanilla is 1.26%. Once the requests are retried, some of them return a 502 status code, bad gateway, which means the router cannot reach the requested pod. For AS and RR, the error rate is 0%. The crash is better tolerated because of the presence of a replica. All requests are thus served with success and return the code 200.

#### Resource consumption analysis

We evaluate here the three fault-tolerance approaches in terms of CPU and memory usage. We chose to evaluate the approaches in terms of resource consumption rather than monetary cost because resource consumption is the main factor that determines cost. The cost may also depend on further factors that vary widely across deployments, such as the cost of infrastructure, servers, networks, energy as well as the billing model of the provider. Figures 20 and 21 show CPU and memory consumption,

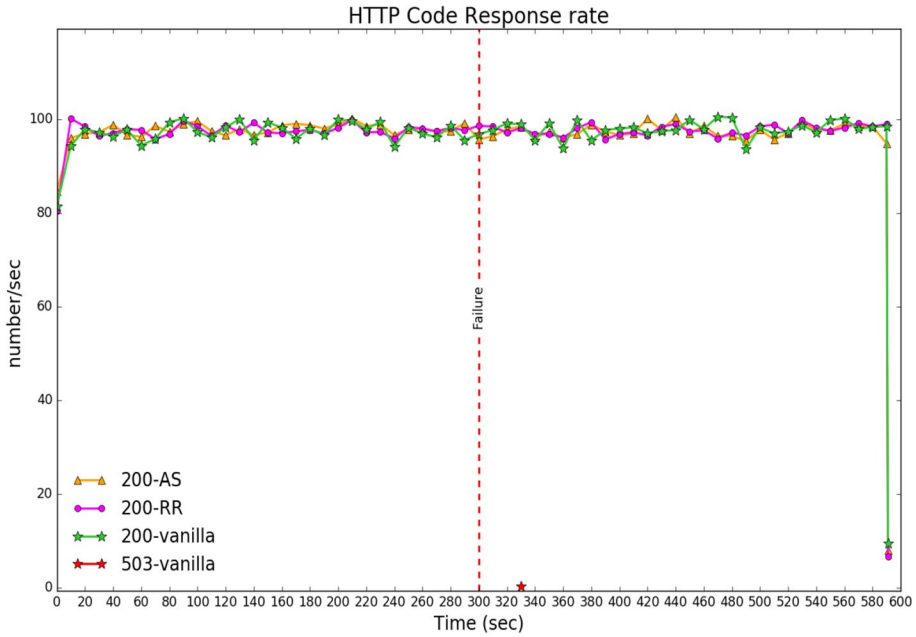


**Fig. 18** HTTP code response rate in vanilla, AS and RR with pod failure

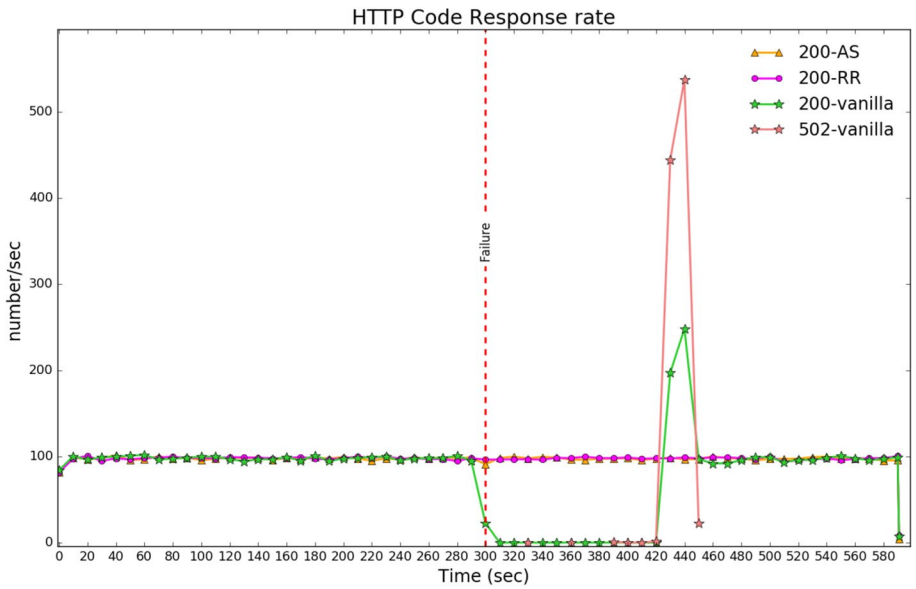


**Fig. 19** HTTP code response rate in vanilla, AS and RR with node failure

respectively, for vanilla, AS and RR approaches without and with failures (pod and node failures). This is 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, node failure), we observe that RR consumes more CPU and memory compared to vanilla and AS. In the case of no failures, for example, the overhead of using RR is 180% in CPU and 52% in memory consumption compared to vanilla. AS has an overhead of 141% in CPU consumption and 39% in memory consumption compared to vanilla. Theoretically the expected overhead of RR should be proportional to the number of replicas (around 100% compared to vanilla in the case of two replicas). However the actual overhead of AS/RR compared to vanilla is

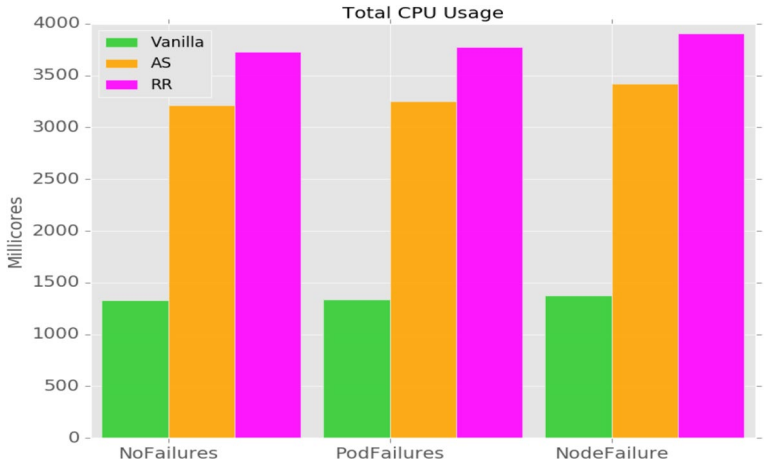


**Fig. 20** CPU consumption in AS, vanilla and RR without and with failures (pod and node failure)

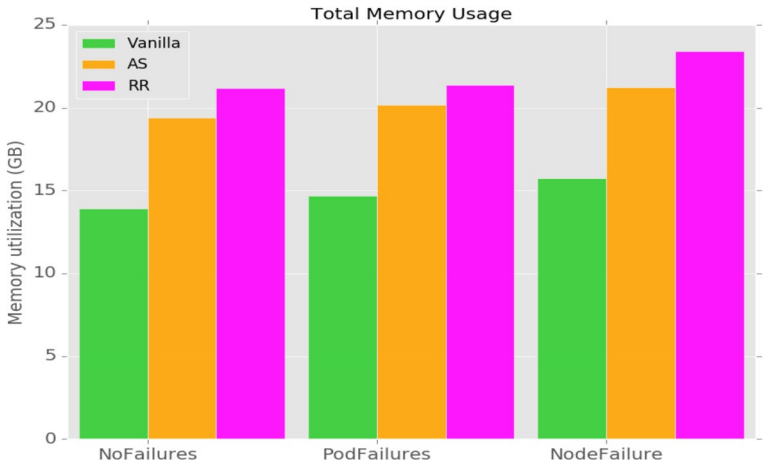


**Fig. 21** Memory consumption in AS, vanilla and RR without and with failures (pod and node failure)

higher. This is due to the implementation of their respective routers that regularly invoke the Kubernetes API to obtain the current IP addresses of pods. We chose this mechanism for its implementation simplicity. This overhead could be reduced by notifying the routers when changes in the pod addresses occur, as in the case of pod failures.

Figures 22, 23, and 24 show the average CPU consumption over time for the Kubernetes master node, the Fission worker node, and the 3 worker nodes for all approaches with a pod failure.

The CPU consumption of AS and RR is similar and vanilla shows the lowest CPU utilization. We notice that on average, the coordinator nodes (i.e., the two nodes hosting respectively the services that manage the Kubernetes cluster and those that manage the Fission functions) need more resources compared to the worker nodes. Especially for AS and RR, their coordinator nodes are experiencing high CPU usage compared to vanilla. As previously mentioned, this is due to the CPU consumption of the routers in the Fission worker node that regularly call the Kubernetes API server to get updates on the IPs of the function pods.

When looking at the CPU consumption of Worker 2 and Worker 3 in AS (see Fig. 23), we notice that after the failure, the CPU usage of Worker 3 starts to grow while the CPU usage of Worker 2 goes down, which reflects the behavior of the failover to the standby pod.

In RR, when pod 1 fails, another one is created in the same node (Worker 2) and we observe a very short peak in the CPU at the 6th minute, as shown in Fig. 24.

## 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

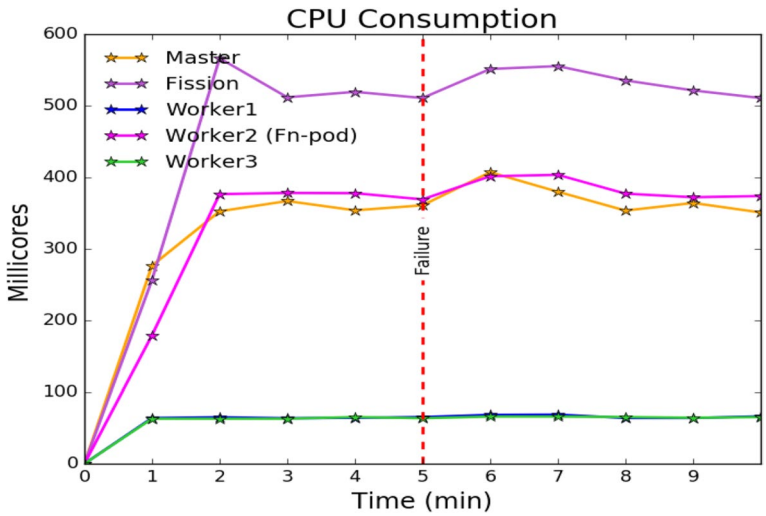


**Fig. 22** CPU consumption per node in vanilla with pod failure

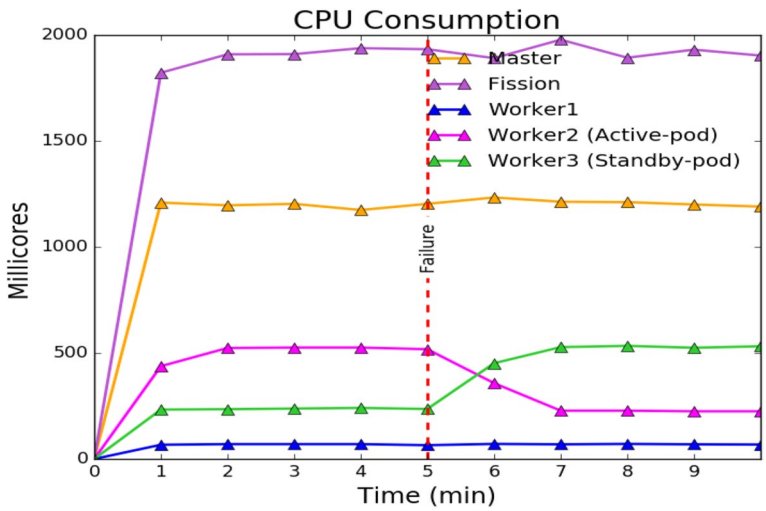


**Fig. 23** CPU consumption per node in AS with pod failure

different properties and is most effective under different conditions. The retry approach is well suited for transient failures that last a short time. This approach consumes less resources and is thus more energy-efficient than the Active-Standby and the Request Replication approaches. The Active-Standby approach offers better availability for long-lasting failures, compared to the retry approach, but at the cost of higher resource consumption. For instance, in our experiments, the Active-Standby mechanism consumes more than two times the CPU consumed by the retry mechanism. The Request Replication approach offers the best availability whatever the duration of the failure. Indeed, when the failure 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 approach incurs the highest resource consumption. In general, we observe that availability and resource consumption in the three approaches are inversely related.

Based on the previous discussion, we can see that choosing the appropriate fault tolerance approach depends 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 the emphasis is on minimum resource usage (e.g., for resource-constrained environments, such as edge environments) with limited need for availability, the preferred

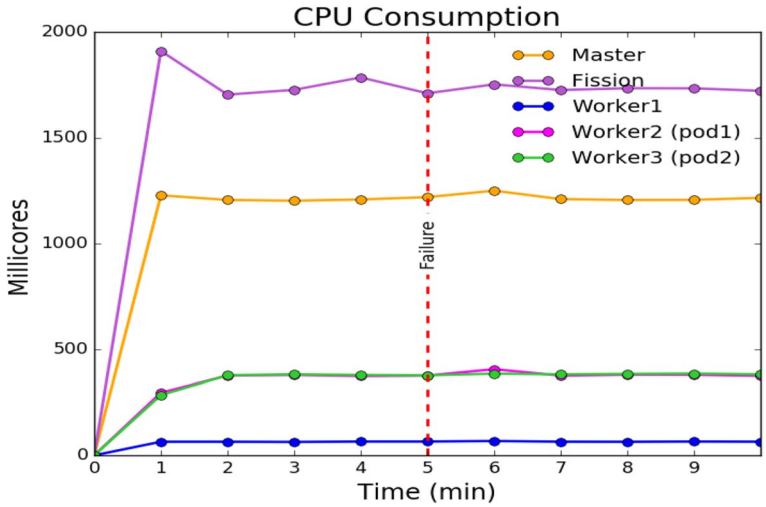


**Fig. 24** CPU consumption per node in RR with pod failure

approach is retry. AS can be used when the requirement is for high availability as well as moderate resource usage.

In the presented experiments, the three approaches were tested with a stateless application that does not retain state between requests (neither within the instance nor in external storage). In future efforts, we will investigate the behavior of these approaches with other application types, such as stateful FaaS applications. With these applications, state is typically maintained in external storage services, such as NoSQL databases [28]. Using RR for such applications seems challenging. The reason is that concurrent accesses generate a high 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 AS or retry. Integrating caching into the stateful functions could mitigate this problem [46].

Given the trade-offs between the different fault tolerance approaches, we believe that a FaaS platform should simultaneously support multiple mechanisms, such as retry, AS and RR, and use one or another according to specific criteria. These criteria may include performance, availability, and resource consumption requirements, application types, fault models, and operating conditions, such as network latencies.

## Conclusion and future work

This paper proposes the integration of an active replication fault tolerance approach (RR) in FaaS platforms. The RR mechanism was experimentally compared with a passive replication mechanism (AS) and the basic retry mechanism in terms of different metrics, and under different failure scenarios.

The obtained results highlight the differences among the three approaches. Notably, they show that the retry approach is not sufficient for providing high availability. The reason is that the default behavior of retry results in significant recovery time in the case of node failures. The retry approach is better suited for transient failures as seen in the network delay scenario. With AS, the recovery time is decreased because the service becomes available as soon as the standby replica detects the failure of the active replica. With RR, the service remains available as long as at least one replica continues to respond to users as recovery does not depend on replacing the faulty replica.

In our future work, we plan to investigate additional fault tolerance approaches proposed in the literature, such as checkpointing. This approach periodically saves a snapshot of an application's state, known as a checkpoint, to be used for restoring the application in the case of failures. In the context of FaaS, some research [34, 36] has already proposed the usage of checkpointing to restart functions from where they timed out, which is useful for functions that execute long-running computations. Checkpointing could be implemented with an adaptive checkpoint interval that uses the timings of already occurred failures to estimate the occurrences of the next ones [47].

Furthermore, we plan to design a fault-tolerant system for FaaS that simultaneously supports multiple mechanisms (e.g., retry, replication, checkpointing) and uses one or another based on the type of FaaS application (e.g., stateful or stateless) and operating conditions (e.g., fault rates, network latencies) while meeting user's requirements (e.g., performance, availability, resource consumption).

**Abbreviations**

| | |
|---|---|
| AS | Active-Standby |
| AWS | Amazon Web Service |
| CPU | Central Processing Unit |
| FaaS | Function as a Service |
| HTTP | HyperText Transfer Protocol |
| IP | Internet Protocol |
| NoSQL | non Structured Query Language |
| RR | Request Replication |

**Acknowledgements**
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).

**Authors' contributions**
Yasmina Bouizem co-wrote the paper, developed the code, performed the experiments, and participated in results analysis. Christine Morin, Nikos Parlavantzas, and Djawida Dib co-supervised Yasmina Bouizem's PhD thesis. They co-wrote the paper. They provided critical feed-back and helped shape the research, analysis, and paper. The author(s) read and approved the final manuscript.

**Authors' information**
**Yasmina Bouizem** received her Master degree in Networking and Distributed Systems from Department of Computer Science, Tlemcen University, Algeria in 2015. She received her Ph.D. degree in Computer Science from the University of Rennes I. Her research interests are specialized in Fault Tolerance, Serverless computing, Cloud computing and distributed systems.
**Djawida Dib** has been an associate professor at the Computer Science Department of the University of Tlemcen since 2015. Her research interests include cloud computing, distributed systems and solving tradeoffs in such systems between cost, performance, failure resiliency and energy efficiency. She received her PhD and MSc degrees in Computer Science from the University of Rennes I.
**Nikos Parlavantzas** has been an associate professor at INSA Rennes and member of the Myriads research team at IRISA/Inria Rennes-Bretagne Atlantique since 2009. Before joining INSA Rennes, he worked as a researcher at Inria and Lancaster University. He has participated in several European projects, such as PaaSage, S-Cube, CoreGRID, and Grid4All. His research interests include cloud computing, autonomic systems, and adaptable middleware. He has a Ph.D. and M.Sc. from Lancaster University, UK, and a Diploma in Computer Engineering and Informatics from the University of Patras, Greece.
**Christine Morin** is a senior scientist at Inria. Her research interests include distributed systems, dependable computing, autonomic computing, green computing, and cloud computing. She published more than 150 papers. She led the XtreemOS and Contrail European projects respectively in Grid computing and Cloud computing. She co-founded the Kerlabs startup for providing commercial support on the Kerrighed cluster operating system resulting from her research activities on the design and implementation of single system image operating systems. She received a PhD in Computer Science from the University of Rennes 1 and a Computer Engineering degree from INSA Rennes, France.

**Funding**
Yasmina Bouizem's PhD work was partially funded by Inria and by a fellowship from the University of Tlemcen. She also received mobility grants from PROFAS B+, an Algerian-French scholarship program offered by the Algerian Ministry of Higher Education and Scientific Research and Campus France, and from Rennes Metropole (France).

**Availability of data and materials**
The source code and related resources are available in our Github repository: https://github.com/YasFaaS/FaaS-FT.

## Declarations

**Competing interests**
The authors declare no competing interests.

Received: 18 September 2022 Accepted: 9 May 2023
Published online: 22 June 2023

**References**

1. Castro P, Ishakian V, Muthusamy V, Slominski A (2019) The rise of serverless computing. Commun ACM 62(12):44–54
2. Jonas E, Schleier-Smith J, Sreekanti V, Tsai CC, Khandelwal A, Pu Q, Shankar V, Carreira J, Krauth K, Yadwadkar N, et al (2019) Cloud programming simplified: A berkeley view on serverless computing. arXiv preprint arXiv:1902.03383
3. Amazon Web Services (2020) Aws lambda features. https://aws.amazon.com/lambda/features/. Accessed 07 July 2021
4. Google cloud functions (2019) Retrying background functions. https://cloud.google.com/functions/docs/bestpractices/retries. Accessed 07 July 2021
5. Azure Functions (2020) Azure functions. https://azure.microsoft.com/fr-fr/services/functions/. Accessed 07 July 2021
6. Fission (2019) Fission. https://docs.fission.io/docs/. Accessed 07 July 2021
7. OpenFaaS (2019) Openfaas. https://www.openfaas.com. Accessed 07 July 2021
8. Kubeless (2021) Kubeless. https://kubeless.io. Accessed 07 July 2021
9. Apache OpenWhisk (2021) Apache openwhisk. https://openwhisk.apache.org. Accessed 07 July 2021
10. Amazon Web Services (2020) Error handling and automatic retries in aws lambda. https://docs.aws.amazon.com/lambda/latest/dg/invocation-retries.html. Accessed 07 July 2021
11. 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/. Accessed 07 July 2021
12. Cloud design patterns (2020) Retry pattern. https://docs.microsoft.com/en-us/azure/architecture/patterns/retry. Accessed 07 July 2021
13. OpenFaaS (2019) Timeouts - asynchronous invocations. https://docs.openfaas.com/deployment/troubleshooting/#timeouts-asynchronous-invocations. Accessed 07 July 2021
14. Felber P, Narasimhan P (2004) Experiences, strategies, and challenges in building fault-tolerant corba systems. IEEE Trans Comput 53(5):497–511
15. Bouizem Y, Dib D, Parlavantzas N, Morin C (2020) Active-Standby for High-Availability in FaaS. In: Sixth International Workshop on Serverless Computing (WoSC6) 2020, Delft, Netherlands. https://doi.org/10.1145/3429880.3430097. https://hal.inria.fr/hal-03043479
16. Amazon Web Services (2022) How do i make my lambda function idempotent? https://aws.amazon.com/premiumsupport/knowledge-center/lambda-function-idempotent/. Accessed 29 Apr 2022
17. Google cloud functions (2022) Retrying event-driven functions. https://cloud.google.com/functions/docs/bestpractices/retries. Accessed 30 Apr 2022
18. Amazon Web Services (2022) Configuring provisioned concurrency. https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html. Accessed 09 Dec 2022
19. Mukwevho MA, Celik T (2021) Toward a smart cloud: A review of fault-tolerance methods in cloud systems. IEEE Trans Serv Comput 14(2):589–605. https://doi.org/10.1109/TSC.2018.2816644
20. AWS Admin (2020) Azure functions geo-disaster recovery. https://docs.microsoft.com/en-us/azure/azure-functions/functions-geo-disaster-recovery. Accessed 07 July 2021
21. Google Cloud Workflows (2021) Google cloud workflows. https://cloud.google.com/workflows. Accessed 07 July 2021
22. AWS Step Functions (2021) Aws step functions. https://aws.amazon.com/step-functions. Accessed 07 July 2021
23. Azure Durable Functions (2021) Azure durable functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable. Accessed 07 July 2021
24. Resume AWS Step Functions (2021) Resume aws step functions from any state. https://aws.amazon.com/blogs/compute/resume-aws-step-functions-from-any-state/. Accessed 07 July 2021
25. Apache OpenWhisk Composer (2021) Apache openwhisk composer. https://github.com/apache/openwhisk-composer. Accessed 07 July 2021

26. Faas-flow (2021) Faas-flow. https://github.com/s8sg/faas-flow. Accessed 07 July 2021
27. Sreekanti V, Wu C, Chhatrapati S, Gonzalez JE, Hellerstein JM, Faleiro JM (2020) A fault-tolerance shim for serverless computing. In: Proceedings of the Fifteenth European Conference on Computer Systems, Association for Computing Machinery, New York, NY, USA, EuroSys '20. https://doi.org/10.1145/3342195.3387535
28. Zhang H, Cardoza A, Chen PB, Angel S, Liu V (2020) Fault-tolerant and transactional stateful serverless workflows. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), USENIX Association, Banff, Alberta. https://www.usenix.org/conference/osdi20/presentation/zhang-haoran. Accessed 21 June 2023
29. Jia Z, Witchel E (2021) Boki: Stateful serverless computing with shared logs. In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM. Association for Computing Machinery, New York, pp 691–707
30. Wu C, Sreekanti V, Hellerstein JM (2020) Transactional causal consistency for serverless computing. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. Association for Computing Machinery, New York, pp 83–97
31. Wu C, Faleiro J, Lin Y, Hellerstein J (2019) Anna: A kvs for any scale. IEEE Trans Knowl Data Eng 33(2):344–58
32. de Heus M, Psarakis K, Fragkoulis M, Katsifodimos A (2021) Distributed transactions on serverless stateful functions. In: Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems. Association for Computing Machinery, New York, pp 31–42
33. Apache Flink StateFu (2021) Apache flink statefu. https://flink.apache.org/stateful-functions.html. Accessed 14 Nov 2021
34. Zhang W, Fang V, Panda A, Shenker S (2020) Kappa: A programming framework for serverless computing. In: Proceedings of the 11th ACM Symposium on Cloud Computing. Association for Computing Machinery, New York, pp 328–343
35. Carver B, Zhang J, Wang A, Anwar A, Wu Y Panruo andCheng (2020) Wukong: A scalable and locality-enhanced framework for serverless parallel computing. In: Proceedings of the 11th ACM Symposium on Cloud Computing. Association for Computing Machinery, New York, pp 1–15
36. Karhula P, Janak J, Schulzrinne H (2019) Checkpointing and migration of iot edge functions. In: Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking, Association for Computing Machinery, New York, NY, USA, EdgeSys '19, p 60–65. https://doi.org/10.1145/3301418.3313947
37. Yussupov V, Soldani J, Breitenbücher U, Brogi A, Leymann F (2021) Faasten your decisions: A classification framework and technology review of function-as-a-service platforms. J Syst Softw 175:110906. https://doi.org/10.1016/j.jss.2021.110906
38. Hightower K, Burns B, Beda J (2017) Kubernetes: up and running: dive into the future of infrastructure. O'Reilly Media, Inc
39. Shilkov M (2019) What is a cold start? https://mikhail.io/serverless/coldstarts/define/. Accessed 07 July 2021
40. Fission Router (2020) Fission router. https://godoc.org/github.com/fission/fission/pkg/router. Accessed 07 July 2021
41. Kubernetes Readiness Probe (2021) Configure liveness, readiness and startup probes. https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/. Accessed 07 July 2021
42. Grid5000 (2020) Grid5000. https://www.grid5000.fr/w/Grid5000:Home. Accessed 07 July 2021
43. Kubernetes (2021) Kubernetes. https://kubernetes.io/. Accessed 07 July 2021
44. Tsung (2021) Tsung. http://tsung.erlang-projects.org/user_manual/. Accessed 07 July 2021
45. chaos (2021) Chaos mesh. https://chaos-mesh.org/. Accessed 07 July 2021
46. Sreekanti V, Wu C, Lin XC, Schleier-Smith J, Gonzalez JE, Hellerstein JM, Tumanov A (2020) Cloudburst: Stateful functions-as-a-service. Proc VLDB Endow 13(12):2438–2452. https://doi.org/10.14778/3407790.3407836
47. bin Bandan MI, Bhattacharjee S, Pradhan DK, Mathew J, (2015) Adaptive checkpoint interval algorithm considering task deadline and lifetime reliability for real-time system. Procedia Comput Sci 70:821–828


## Publisher's Note