Canary: Fault-Tolerant FaaS for Stateful Time-Sensitive Applications

Moiz Arif
Department of Computer Science
Rochester Institute of Technology
Rochester, NY, USA
ma3890@cs.rit.edu

Kevin Assogba
Department of Computer Science
Rochester Institute of Technology
Rochester, NY, USA
kta7930@cs.rit.edu

M. Mustafa Rafique
Department of Computer Science
Rochester Institute of Technology
Rochester, NY, USA
mrafique@cs.rit.edu

***Abstract*—Function-as-a-Service (FaaS) platforms have recently gained rapid popularity. Many stateful applications have been migrated to FaaS platforms due to their ease of deployment, scalability, and minimal management overhead. However, failures in FaaS have not been thoroughly investigated, thus making these desirable platforms unreliable for guaranteeing function execution and ensuring performance requirements. In this paper, we propose *Canary*, a highly resilient and fault-tolerant framework for FaaS that mitigates the impact of failures and reduces the overhead of function restart. *Canary* utilizes replicated container runtimes and application-level checkpoints to reduce application recovery time over FaaS platforms. Our evaluations using representative stateful FaaS applications show that *Canary* reduces the application recovery time and dollar cost by up to 83% and 12%, respectively over the default retry-based strategy. Moreover, it improves application availability with an additional average execution time and cost overhead of 14% and 8%, respectively, as compared to the ideal failure-free execution.**

***Index Terms*—Data-intensive Computing, Serverless Computing, Deep Learning, Data Parallelism, OpenWhisk, TensorFlow**

I. Introduction

Function-as-a-Service (FaaS) is a computing paradigm that allows building, executing, and managing applications as functions without having to deploy and manage the underlying infrastructure. FaaS is a popular way to implement serverless computing which abstracts away infrastructure management such as managing or provisioning servers and resource allocation from developers, thus enabling them to focus on application development and delivering business value. FaaS follows an event-driven execution model that runs functions specifying the server-side application logic in stateless containers. FaaS platforms have pre-built runtime [1] images to support the execution of functions written in various programming languages. A *runtime* is a container image that contains the application's source code or executables, libraries, and the required software packages for deploying and executing application functions. Runtime images alleviate users to install the required libraries and software packages for executing their applications.

Modern applications are composed of complex workflows where different components depend on and interact with other components to work as a single unit by relying on the timely completion of each sub-component. The functioning of each component requires information about the executions of previous components. This information is passed on as the application states throughout the application workflow. Similarly, iterative and stateful applications maintain application information and states for executing the current iteration or to service the next user requests. FaaS platforms support such stateful workloads by enabling application functions to store data and state information on ephemeral storage. The overall execution workflow is divided into several loosely-coupled independent small functions that must reliably interact with each other. For stateful application, each function starts its execution using triggers that are invoked after the successful completion of the previous function. For example, a MapReduce [2] workload launches mappers that process the input data and produce intermediate data. The reducers are launched after successful mapper execution and consume mappers output to produce the final result. Similarly, deep learning (DL) workloads consist of various execution stages [3] including data pre-processing, model training, weight aggregation, and inference. Each stage is managed by a set of stateful functions that work with the state and data produced by previous functions in FaaS platforms [4]–[6].

An increasing number of High-Performance Computing (HPC) applications [7], [8], such as DL jobs [9], [10], leverage FaaS platforms to increase parallelism and scalability. However, existing public [11]–[14] and on-premise [15], [16] FaaS offerings have resource limitations for function execution, such as memory allocation and network timeouts, that negatively impact the performance of function executions. Failures in FaaS platforms are common [17] and are broadly classified as hardware, software, runtime, network, and application level failures, e.g., function failures [18], permission or resource limitation failures [19], [20], runtime failures [21], invocation failures [22], and bugs in the application code [23]. FaaS platforms sustain these failures by using retry-based strategies to restart the failed functions until their execution is successful. The retry-based approach significantly increases the application execution time and is particularly impractical for time-sensitive workloads with strict performance requirements. Similarly, a failure during stateful function execution results in loss of computation, inconsistent application state, and potential loss of critical data. These

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE

challenges only exacerbate in large-scale HPC systems where 60% and 20% of failures are attributed to hardware and software, respectively [24], [25]. Therefore, it is crucial to reduce the recovery time of these serverless applications for meeting their performance requirements and service level agreements (SLAs).

Modern data centers consist of heterogeneous computing resources with unique performance profiles [26]. In such setups, the performance of applications depend on the availability of resources on the underlying servers. Particularly, the resource heterogeneity results in unpredictable application performance [27] and function recovery time after failures [28]. Older hardware is more prone to failure as compared to relatively newer hardware [29], [30]. Similarly, slower computing devices and storage systems can significantly increase application recovery time as compared to the faster data center resources. Moreover, in case of failures, the function recovery time on heterogeneous resources is non-deterministic and results in variations that affect application performance. Therefore, FaaS platforms must incorporate resource heterogeneity while providing a low recovery time for failed functions.

In this paper, we develop a framework, called *Canary*, that enables FaaS platforms to tolerate failures over heterogeneous resources and minimize the impact of failures on application performance. We achieve this by proposing checkpointing mechanisms and runtime replication for faster function recovery from a failure. *Canary* is composed of four key components, namely, *Core Module*, *Checkpointing Module*, *Replication Module*, and a *Runtime Manager Module*. The *Core Module* orchestrates the detection and recovery of the failed functions and handles coordination between all the components. The *Checkpointing Module* handles checkpointing of data and application states, and the *Replication Module* handles the replication of runtime and data. The *Runtime Manager Module* manages the life cycle of the deployed runtimes and their replicated runtimes that are deployed in the cluster. *Canary* tracks the current state of each function execution, adjusts the checkpointing frequency, and uses state information to restore stateful functions from their most recent checkpoints. Our proposed *Canary* framework makes FaaS platforms reliable and fault-tolerant, and reduces the overall recovery time of functions.

Specifically, we make the following contributions:

- We propose *Canary* which improves the reliability of FaaS platforms and provides fault tolerance to applications with minimal application-level changes.
- We provide a prototype implementation of *Canary* and integrate it with a popular open-source FaaS platform, Apache OpenWhisk [15], to demonstrate its effectiveness.
- We conduct a thorough performance evaluation of *Canary* and compare it with the state-of-the-art reliability approach in FaaS platforms. Our evaluation shows that *Canary* reduces the application recovery time and dollar cost by up to 83% and 12%, respectively, over the default retry-based strategy, and improves application availability with an additional average execution time and cost overhead of 14% and 8%, respectively, as compared to the ideal failure-free execution.

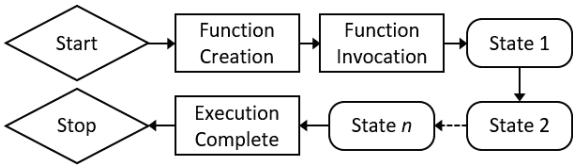

Fig. 1: Execution flow of a function in FaaS.

The rest of this paper is organized as follows. In Section II, we discuss the motivation and background of various failure types and the current fault tolerance mechanism of FaaS platforms. In Section III, we formulate the problem of providing fault tolerance in FaaS platforms. In Section IV, we explain the design of our proposed framework *Canary* and explain the functionality of each module. In Section V, we present the implementation details of *Canary* along with a detailed comparison of our approach with alternate approaches. In Section VI, we summarize the existing efforts along with their limitations that are related to *Canary*. Finally, in Section VII, we conclude the paper.

II. Background and Motivation

A. FaaS Execution and Failure Types

The execution workflow of a function is shown in Figure 1. Serverless functions are small and modular pieces of code that typically perform a single function in response to an event. In a FaaS platform, functions are created by providing application code, its runtime, memory allocation, trigger, and a unique name. Triggers invoke functions that subsequently execute the provided application code in the specified runtime. A function can consume input data and process the data in a single or multiple phases called states. These states are referred to as the current state of function variables and data structures. Functions produce intermediate data and final data that is stored in a storage system ready to be consumed by other functions. Once the processing is complete, a function can invoke other functions which work on the data produced by the previous functions.

The FaaS platform can experience failure at each phase of the function execution. These failures are mainly classified under four categories, i.e., request, concurrency, function, and runtime. Request failures are related to the resources requested for a function that exceed the limits associated with a particular account. Concurrency failures are related to the number of concurrent executions requested by the application and the maximum concurrent invocations allowed by the FaaS platform. Function failures that stem from the application code and runtime failures are related to the preparation and setting up of the runtime required for the function's execution. Function and runtime failures are important from a FaaS perspective since they cause stateful applications to lose data, computation, and money on public FaaS platforms. These failures must be either handled proactively or their recovery time must be minimal to meet application SLAs.

B. Reliability and Fault Tolerance in FaaS Platforms

Failures occur at various levels such as hardware [29]–[31], platform [32], [33], software stack [34], [35], and application [36], [37]. However, the reliability of FaaS platforms are based on best effort approaches. Function failures result in requests to be dropped or executed multiple times, thus requiring applications to implement stronger reliability guarantees. Current approaches for providing stronger semantics, such as *exactly once* guarantees, can be integrated into FaaS platforms at a high cost for both latency and resource consumption. Different types of failures, e.g., server, network links, and software processes can result in a loss of data and inconsistent stream processing across the data center. Therefore, providing fault tolerance and reliability to stateful applications in FaaS platforms is critical yet a largely unexplored area.

The adoption of serverless computing continues to rise with more than 200% [38] increase in the average weekly invocations over the past years. Meanwhile, the number of failures in function execution has increased from 1% for highly maintained runtimes to about 25% for deprecated runtimes [38]. Therefore, there is a need to mitigate FaaS failures and reduce the recovery time of failed functions to improve application reliability and response time. FaaS platforms deployed on HPC infrastructure are directly impacted by HPC failures [39]. The state-of-the-art fault tolerance techniques, e.g., checkpointing [40], [41] and replication [42], [43], to mitigate HPC failures cannot be directly applied to FaaS platforms [44], [45] because of their unique characteristics, e.g., the massive scale and short lifespan of invoked functions.

C. Stateless and Stateful Functions

Stateless functions execute in isolation with no prior information or inputs from previous function executions. On the other hand, stateful functions are frequently accessed and referred to over time. They have information on previous function execution, and the current execution may be affected by the status of previous executions. If a stateful function is interrupted, terminates prematurely, then the context and execution progress of an application is lost, making it impossible to return and resume from the previous state. Moreover, modern data center applications, e.g., iterative applications, are stateful and depend on application data and results from previous executions. The challenge of maintaining states exacerbates for FaaS platforms, which use containers that are designed to be stateless, portable, and flexible, for executing the given function. Because of the popularity and ease of use of FaaS computing, existing applications are being migrated [4] to FaaS platforms, while new stateful applications are being developed using FaaS platforms.

To support stateful functions, one approach is to make the FaaS execution model stateful by default. However, this violates the basic design concepts of short-running and lightweight functions because persisting data to provide stateful capabilities would significantly increase function execution time. Nevertheless, migration of stateful applications to FaaS computing is inevitable, therefore, FaaS platforms must adapt to support both stateless and stateful applications. Due to the ephemeral nature of data in FaaS computing, the impact of a function failure would be significant as all progress of the failed function will be lost. Typically, stateful applications rely on independent fault tolerance approaches, e.g., replication to external storage, for reliably maintaining their states, which may not work in FaaS computing because of the limited resource allocation, both in time and space, for each function. Simple retry-based approaches do not address failure challenges for stateful functions because the failed application would experience loss of computation and inconsistencies in critical data. To the best of our knowledge, there is no end-to-end fault tolerance mechanism that adapts to both stateless and stateful FaaS applications. In this paper, we propose an end-to-end framework, called *Canary*, that enables FaaS platforms to provide fault tolerance for stateless and stateful functions.

III. Problem Formulation

The execution of a workload on FaaS platforms is marked by distinct events including job launch, container launch, container initialization, execution startup, state updates, function completion, and job completion. With a given function failure rate on the FaaS platform, one or more containers are subject to fail, thus increasing the total execution time, T_{tot}, of applications. To minimize the total execution time, we include checkpoint-restore and runtime replication mechanisms. *Canary* reduces the overhead of a failure by loading the latest checkpoint onto a replicated runtime. By restoring from the latest checkpoint, *Canary* eliminates the container launch and initialization overhead, and the duration between the function start and the latest checkpoint. To simplify the target optimization problem, we define the total execution time of a function f in Equation 1 as the sum of the function launch time lch_f, initialization time ini_f, workload execution time $exec_f$, and the remainder of the execution fin_f from the last state update to function completion. To model the impact of checkpoint-restore and replication, we define two binary variables α_f and α_i where α_f equals 0 if a replicated container is used by a function and 1 otherwise. Similarly, α_i is set to 0 for all the states that were collected during the failed execution and 1 otherwise.

$$\min \quad \sum_{f=1}^{F} \alpha_f \left(lch_f + ini_f\right) + exec_f + fin_f \tag{1}$$

We incorporate the overhead of checkpoint-restore by including the time taken for checkpoint collection and restoration in the workload execution time as defined in Equation 2. The workload execution time is comprised of the duration $st_{ij}(i < j)$ between updates of states i and j, the overhead ckp_i of checkpointing state i, and the additional time t_{res} to restore the latest checkpoint. Thus, the optimization problem can be formulated using the following equations:

$$\text{such that,} \quad exec_f = t_{res} + \sum_{i=1}^{S} \alpha_i \left(ckp_i + st_{ij}\right), \forall i < j \tag{2}$$

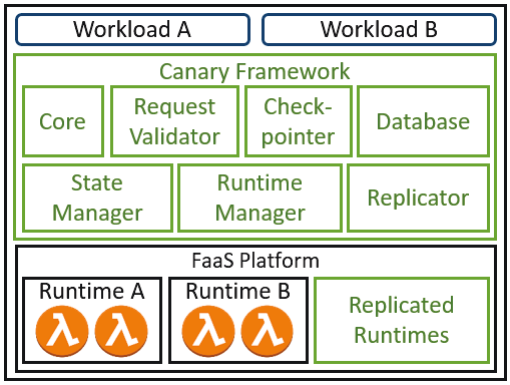

Fig. 2: High-level architecture of *Canary*.

The saved checkpoints denote various function states throughout their execution and enable the restoration of a previous state whenever a function fails. Consequently, we compute the total execution time of an application by including the execution time of functions that used replicated runtime based on Equation 3.

$$\alpha_f \in \{0,1\}, \quad \alpha_i \in \{0,1\}, \quad \forall f \in \mathcal{F}, \forall i \in \mathcal{S} \tag{3}$$

IV. System Design

We now present the objectives and details of *Canary* design.

A. Design Objectives

The main objective of the *Canary* framework is to improve the fault tolerance and reliability of FaaS platforms and provide execution and SLA-level guarantees for stateful functions. The key objectives of the proposed framework are as follows:

1) Make FaaS platforms tolerate faults in function execution and invocation. This will enable them to execute functions *exactly once* and reduce the total recovery time for the failed functions.
2) Enable faster recovery of serverless applications from faults by using intelligent and dynamic checkpointing and replication techniques.
3) Replicate container runtime to ensure faster function execution after a failure by restoring the saved state and data in the replicated runtimes.

B. High-level System Overview

Canary ensures that functions execute *exactly once* on FaaS platforms for achieving minimal application execution time. *Canary* achieves this by proposing a modular architecture, which is shown in Figure 2. *Canary* consists of a *Core Module* that handles end-to-end job execution, failure recovery, and coordination between various components. The *Request Validator Module* is used by the *Core Module* to avoid failures regarding the submitted job request. The *Checkpointing Module* handles the state and critical data checkpointing. The *Runtime Manager Module* keeps track of all the function runtimes deployed and the runtime replicas for the jobs created by the *Replication Module*. Finally, the database stores the function states and the checkpoints.

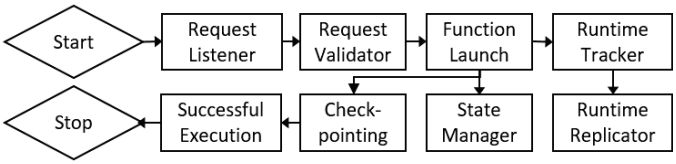

Fig. 3: High-level orchestration of *Canary* modules.

C. System Design

In this section, we describe the details of the new software modules added in *Canary* to reduce the total execution and recovery time with checkpointing and replication techniques.

1) Core Module: The *Core Module* in *Canary* orchestrates execution between its various components and modules, as shown in Figure 3. It exposes a lightweight listener interface that receives incoming user requests and forwards them to the *Request Validator Module* for validation. It generates a set of unique IDs for the submitted jobs functions, checkpoints, and replicas used to identify functions, corresponding applications, location of functions, identification of the failed functions, and the associated checkpoints.

The *Core Module* handles the creation and maintenance of the required database tables. The five main tables created in the database are *worker_info*, *job_info*, *function_info*, *checkpoint_info*, and *replication_info*. The *worker_info* table stores information about the platform, including the number of nodes in the cluster. It also stores worker-specific information, i.e., assigned roles and system specifications. The *job_info* table stores information about the submitted job, its unique ID, the number of functions launched for each job, and other critical information required by the *Core Module*. The *function_info* table stores information about all the functions launched for the submitted jobs, their unique IDs, the job ID to which they belong, runtime for each function, and the worker on which the function is deployed. The *checkpoint_info* table stores information about the checkpoints of each function, its unique ID, job ID, function ID, and the state information related to the checkpoint. Finally, the *replication_info* table stores information about the replicated runtimes deployed on the FaaS platform. It also includes the runtime information, job ID, and the worker information where the replicated runtime is deployed.

The *Core Module* forwards the validated requests for scheduling and creates database entries based on the type of job, its runtime, number of scheduled functions, checkpointing frequency, and the replication factor. The *Core Module* forwards the information to the *Checkpointing Module*, which stores the checkpoint metadata in the database. It coordinates between different components of *Canary*'s runtime, which includes scheduling of function runtimes, usage details, and the entire life cycle, through the *Runtime Manager Module*. It also keeps track of all scheduled functions and their current states. Upon function failure, the *Core Module* detects the failure, identifies the function runtime, gathers checkpoint information, and initiates the recovery process. The recovery

Algorithm 1: State and Critical Data Checkpointing.

Input: Func. ID f_{id}, Job ID j_{id}, State st, Checkpoint $ckpt$

begin
  for *each* st **do**
    if *user* $ckpt$ **then**
      get $ckpt_{data}$, $ckpt_{name}$, $ckpt_{loc}$
      if $ckpt_{data} > db_{limit}$ **then**
        $ckpt_{data} \rightarrow disk$
        $ckpt \leftarrow \{ckpt_{name}, ckpt_{loc}\}$
      else
        $ckpt \leftarrow \{ckpt_{data}\}$
      end
    else
      $ckpt \leftarrow \{st, data_{cric}\}$
    end
    if $ckpt_{cur} > ckpt_{thresh}$ **then**
      remove $ckpt_{oldest}$ from db
    end
    push $\{j_{id}, f_{id}, ckpt_{id}, ckpt\}$ to db
  end
end

process restores the function from its latest checkpoint available on the runtime associated with the failed function.

2) Request Validation Module: The main function of the *Request Validator Module* is to prevent request failures before *Canary* starts processing the request. It accepts requests from the *Core Module* and uses the job information and the resources requested from the FaaS platform for validating the job request. The *Request Validator Module* verifies if the requested resources are within the resource limits of the FaaS platform, and the user has not reached the associated maximum concurrent function limit. For example, if invoking a new function would result in a concurrency failure because the requested functions, if launched, will exceed the maximum limit, the *Request Validator Module* notifies the *Core Module* which queues the job until there is enough limit available to launch new functions.

3) Runtime Manager Module: The *Runtime Manager Module* keeps track of all runtimes used by the running functions in the cluster. It works alongside the *Replication Module* to replicate the runtimes being used in the cluster. It maintains information about the used runtimes and their corresponding replicated runtimes and enables the *Core Module* to map the failed functions to the replicated runtimes in the event of a function failure. Moreover, the *Runtime Manager Module* stores the location information of replicated runtimes that are deployed in the cluster.

4) Checkpointing Module: Stateful functions produce data that must be stored and persisted during and after the function execution along with the state information. Functions that belong to the same application require the state information and data from the previous functions for their successful execution. *Canary* supports fault-tolerant stateful function by maintaining the state information of all functions along with the application data.

a) State Management: The *Checkpointing Module* exposes its core functionality via an API that interacts with other modules to monitor and record the state of running functions. Application states can be defined in the application code that will be used by the *Checkpointing Module* for checkpointing. With minimum modification to the function code, application states are registered by calling the *Canary* APIs. The specified states are stored throughout application execution and are used to recover a failed function.

The *Checkpointing Module* exposes the functionality to define critical data within the application code that should be replicated and persisted after the successful function execution. This functionality is critical when an application must store its critical data structures along with the function state. This data is added to the state information. We show *Canary*'s approach of checkpointing application states and critical data in Algorithm 1. The location for storing critical datasets is determined by the total size of the dataset. Checkpoints in *Canary* are primarily maintained in an in-memory key-value (KV) data store. We use Apache Ignite [46] as the KV store for storing the state information. However, in-memory databases limit the size of data stored per key. The *Checkpointing Module* transfers the checkpoint data to a faster storage tier available in the system such as persistent memory, Ramdisk, or to a shared storage accessible to all cluster nodes. The storage hierarchy is determined at the deployment phase of the FaaS platform and can be overwritten by a custom storage endpoint, such as an S3 bucket. The *Checkpointing Module* executes in a linear time to checkpoint the state of each function in a given job. Algorithm 1 yields $\mathcal{O}(S)$ complexity, where S denotes the number of states within a function.

b) Checkpoints: Canary records a series of state checkpoints throughout the function execution and stores the latest *n* checkpoints in an in-memory data store. The initial value of *n* is set to 3, which is dynamically adjusted throughout the execution based on the application data to be checkpointed and the frequency of states produced during function execution. An application state is comprised of current values of its critical data structures that are registered with the *Checkpointing Module*. The critical data remains available in the persistent storage or a KV store and is used to restore the corresponding failed function. For enabling quick lookup, application states are stored in a KV store where the key corresponds to the function ID and the value corresponds to its states. When a new function is assigned the task of a failed one, the *Checkpointing Module* issues a query to the KV store to retrieve the state of the given function ID. When the size of the checkpoint and the data exceed the database limits, the data is then stored in a fast storage tier or external storage, e.g., an in-memory storage or a distributed persistent memory, and the location of the checkpoint is pushed to the database along with the state information. For a DL workload, the checkpoint also includes a copy of pre-processed data, model weights, and other data required to resume the training process from the failed epoch.

By default, *Canary* implements an implicit checkpointing strategy, which has coarse-grained control over checkpoint intervals, location of stored checkpoints, and restoring function

Algorithm 2: Runtime Replication at Job Submission

Input: Act. funcs. ($func_{act}$), Act. repl. (rep_{act}), Repl. loc. (rep_{loc}), Sched. funcs. ($func_{sch}$), Sched. runt. (run_{sch})
Output: Req. repl. (rep_{req}), Repl. loc. (rep_{loc}), Repl. thresh. rep_{th}

```
begin
    compute func_tot given (func_act, func_sch)
    for each run_sch do
        compute rep_req
        if rep_req ≥ 1 then
            compute cur_rep_factor given (func_act, rep_act)
            compute new_rep_factor given (func_tot, rep_act)
            if cur_rep_factor < new_rep_factor then
                determine rep_loc
                launch rep_req at rep_loc
            end
        end
    end
end
```

state from the stored checkpoint in the event of a failure. *Canary* also supports explicit checkpointing where the application can specify its state and data for creating checkpoints, thus reducing the checkpoint size and the associated overhead while increasing the programming complexity. In both of these approaches, checkpoints are first stored in either the KV-store or written in-memory and then flushed asynchronously to the shared storage that is available to all nodes in the cluster.

c) State Recovery: The *Checkpointing Module* handles the recovery of failed functions by restoring the function state and data to a new function. Checkpointing provides the record of previous states of the function to avoid restarting the function from the beginning. The *Core Module* detects failed functions in the cluster and handles the end-to-end recovery process. It identifies the execution runtime required by the failed function, the latest checkpoint available, and the location of the checkpoint data. The *Core Module* ensures that the best possible replicated runtime is selected to minimize the recovery time. Once the replicated runtime is located, the function is deployed on it along with the checkpointed function state and data. The *Core Module* notifies the *Runtime Manager Module* about the runtime utilized during the recovery process. Once the function state has been recovered, the function resumes normal execution and continues execution from its previous state. In the event of multiple function failures, the default retry-based strategy concurrently restarts all the failed functions which leads to resource contention and further increases the recovery time.

5) Replication Module: To ensure that *Core Module* quickly recovers failed functions, the *Replication Module* replicates the runtimes used for launching functions of the scheduled jobs. The runtimes used at any given point are replicated throughout the cluster to enable faster recovery by reducing initialization and cold-start latencies by providing warm function runtime. Instead of creating a replica of each running function's runtime, the *Runtime Manager Module* detects the runtime of an invoked function and verifies whether a corresponding replica is active. The *Runtime Manager Module* only triggers the replication when a function is created with a runtime that is not already replicated in the cluster. Once a replica is assigned to a failed function, the *Runtime Manager Module* creates a new replica if an active function is deployed with the same runtime to replace the existing replica. Therefore, throughout the execution of a function, an active replicated runtime is available to use for failure recovery.

Algorithm 2 explains the runtime replication workflow in *Canary*. Once a new job is submitted to the FaaS platform, the *Core Module* determines the number of functions $func_{sch}$ to launch for the job and the function runtimes run_{sch} to schedule. The replication module uses a linear-time method to compute the total number of functions $func_{total}$, including active functions $func_{act}$, and iterate through the scheduled runtimes for replication. For each run_{sch}, the replication module computes the required number of replicas rep_{req} for a given job. The current replication factor cur_rep_{factor} is the ratio of $func_{act}$ and rep_{act} and the new replication factor new_rep_{factor} includes the $func_{total}$ and rep_{act}. The replication factors determine if enough runtime replicas are available for all running functions. The runtime replication module keeps the current and new replication factors consistent and if the cur_rep_{factor} is less than the new_rep_{factor}, a new runtime replica is launched at the replica location rep_{loc} which is determined to avoid a single point of failure for the submitted job as well as for the FaaS platform. The rep_{loc} is crucial to recover failed functions as it provides enough replicas to the *Core Module* to select a suitable replica to ensure minimal recovery time on heterogeneous resources.

a) Runtime Replication Factor: The runtime replication factor maps running functions to the replicated runtimes. A higher factor value shows that the number of replicated runtimes for each runtime is higher. This provides redundancy and allows faster recovery for large function failures but results in higher operating costs. A lower value of the replication factor means that less number of replicated runtimes are launched. This results in lower cost, but, in the event of large function failures, the initialization time of *Canary* for launching new functions becomes the same as the default retry-based strategy. The *Replication Module* dynamically adjusts the replication factor to achieve an optimal operating point which results in less frequent restarts and lower operating costs.

b) Replica Placement: The *Replication Module* handles the placement of runtime replicas in the cluster. The replica placement follows a set of rules that determines the ideal location for a replica based on the location of the running functions. The first replica is placed on any worker that hosts the job function. Further replicas are placed away from the worker hosting the first replica to avoid a single point of failure for the replicated runtimes. The placement decisions are locality aware and take into account the location of worker nodes in the data center.

V. Performance Evaluation

In this section, we present the evaluation of the proposed *Canary* framework. We explain our evaluation methodology and testbed, workloads used in evaluating *Canary*, and the performance metrics that are used to analyze and compare the effectiveness of *Canary* with alternative approaches.

A. Implementation

We implemented *Canary* using approximately 1300 lines of python code to demonstrate its effectiveness using the open-source Apache OpenWhisk [15] serverless platform. However, *Canary* can be integrated with other serverless platforms, e.g., AWS Lambda [12], Azure Functions [14], and Google Cloud Function [13]. Our hand-tuned implementation automates launching functions by integrating with the OpenWhisk controller. In our implementation, we launch one container per function and leave consolidating multiple functions in a single container to reduce the cold start latency for future work.

B. Evaluation Methodology

We compare *Canary* with the baseline that represents an ideal scenario where there are no function failures during workload execution. We also compare the performance of *Canary* with a more realistic scenario where the FaaS platform experiences failures and relaunches the failed functions to restart their execution from the start. In our evaluation, we measure and compare the recovery time of the failed functions and the total application or job execution time. We run each experiment 10 times and report the average for each experiment. Overall, we observe a negligible variance, i.e., less than 5% between different executions of the same experiment. We define error rate as the percentage of the failed functions for a given workload. We simulate failures by randomly killing containers that host functions based on the defined error rate, and vary the error rate from 1% to 50% to analyze *Canary* in a variety of execution environments.

C. Evaluation Setup

1) Evaluation Testbed: Our testbed consists of a cluster of 16 bare-metal servers from the Chameleon Cloud testbed [47] that are connected using 10G Ethernet. Each server has two Intel Xeon Gold 6126/6240R/6242 processors, contains 192 GB of main memory, and runs Ubuntu 20.04 LTS server operating system. We deploy OpenWhisk [15] on a Kubernetes [48] cluster along with Docker [49], OpenWhisk CLI (wsk), and CouchDB [50]. We deploy Apache Ignite [46] to store data in the highly scalable distributed cluster using replicated caching mode which ensures that the data is available in the entire cluster. We also enable Ignite native persistence to provide data persistence. The underlying storage for storing large files is shared over NFS [51] across the cluster nodes. We also enable the option to use Intel Optane persistent memory [52] in AppDirect mode [53] or Ramdisk [54] for storing large files and to avoid I/O bottlenecks.

2) Workloads: To evaluate *Canary*, we use five classes of application workloads: deep learning (DL), web service, Spark [55] data mining, data compression, and graph search. These applications are developed using Python, Node.js, and Java programming languages and use their corresponding execution runtimes. We used these workloads as these are the most widely used function runtimes in FaaS based on the current FaaS adoption trends [38], and include representative serverless HPC applications from the SeBS [56] benchmark. The DL workload is a TensorFlow [57] application that trains ResNet50 [58] model on the MNIST [59] dataset over 50 epochs. Checkpoint data for a DL application include weights and biases collected after every successful completion of an epoch. Web service workload is composed of responding to 50 requests from a web front-end to a database, i.e., PostgreSQL [60]. Each request is composed of five queries and checkpoints include queries and responses after each request. Spark data mining workload entails extracting, transforming, loading, and analyzing the given dataset to get meaningful insights, where each part of the computation is implemented as serverless functions. Specifically, it computes the diversity index at the local and national levels over the US census data [61]. A checkpoint is collected when the output for each location is computed and aggregated with the existing results. Data compression workload is a modified version of the SeBS *311.compression* benchmark that performs zip compression [62] on 50 input files ($\sim$1 GB each). The input and output files are stored in the local storage instead of S3. Each function processes multiple input files and a checkpoint is performed after compressing an input file. Finally, the Graph search workload is based on the SeBS *501.graph-bfs* benchmark which performs Breadth-First Search (BFS) using igraph [63] in a binary tree with 50 million vertices. Each function is checkpointed after 1 million vertices have been traversed. Depending upon the experiment, these workloads require invoking one or more functions where each function is invoked in a separate container.

3) Performance Metrics: In our evaluation, we consider the total execution time, i.e., the time required to complete the submitted application including the time consumed in recovering from failures to study the effectiveness of the studied approaches. We also measure the failure recovery time and perform a cost-benefit analysis of *Canary* in terms of dollar cost incurred to quickly recover from failures by leveraging additional resources, e.g., for function runtime replication.

D. Performance Results

1) Impact of Runtime Replication on Recovery Time: We run the workloads as functions on OpenWhisk and report the impact of replicated runtimes on the failure recovery time for the given workload runtimes. Figure 4 shows the impact of replicated runtimes on the workload execution time with varying failure rates for 100 invocations of Python, Node.js, and Java container runtimes. We observe that the replicated runtimes reduce the recovery time by up to 81% as compared to the default retry-based recovery strategy. Moreover, we

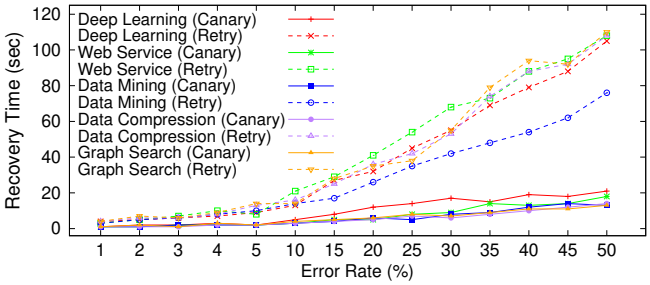

Fig. 4: Impact of replicated runtimes on recovery time for 100 function invocations.

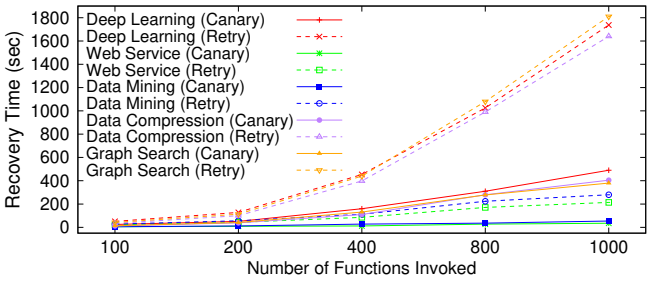

Fig. 5: Impact of replicated runtimes on recovery time with a failure rate of 15%.

observe that as the failure rate increases, the recovery time of the default retry-based strategy increases almost linearly due to the increasing number of failed functions. However, due to the replicated runtimes, *Canary* keeps the recovery time fairly constant and stays close to the ideal scenario where there are no function failures. Similarly, the replica placement also incorporates resource heterogeneity to mitigate the impact of variation in recovery time on application performance. As more functions fail, the replicated runtimes are utilized effectively and *Canary* dynamically increases the replication factor to cope with the failures and reduces function initialization time. Overall, *Canary* reduces the recovery time by 76%, 81%, 78%, 79%, and 80% on average as compared to the default retry-based approach for DL, web service, Spark data mining, data compression, and graph search workloads, respectively.

We also run experiments to observe the performance of replicated runtimes for a large number of function invocations in a cluster setup with a fixed failure rate of 15%. The functions fail at random intervals during function execution. The results are shown in Figure 5. We observe that the runtime replication strategy performs better than the default retry-based strategy by up to 82%. The recovery time of *Canary* remains close to the ideal scenario where there are no failed functions. The additional time as compared to the ideal scenario is due to the time required to migrate the function to the replicated runtime and includes cases where the platform has to wait for the replicated runtimes to be ready where large numbers of functions fail simultaneously and there are not

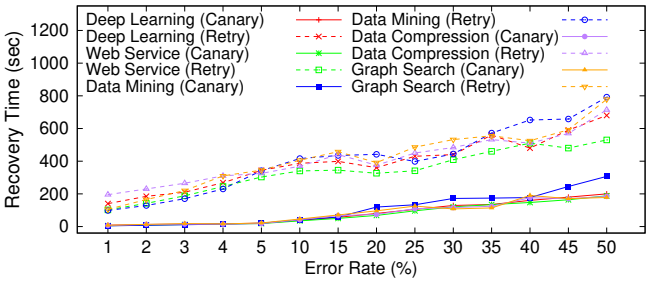

Fig. 6: Impact of checkpoints on recovery time for 100 function invocations.

enough replicated runtime to host the failed functions. *Canary* strategically places the replicated runtimes based on the job, locations of the functions, types of runtime containers used by the failed functions, and the current resource availability in the cluster. Overall, we observe that for this experiment, *Canary* reduces the recovery time by 63%, 82%, 80%, 70%, and 71% on average as compared to the default retry-based approach for DL, web service, Spark data mining, data compression, and graph search workloads, respectively.

2) Impact of Checkpointing on Recovery Time: We study the impact of checkpoints in recovering from function failures by increasing the failure rate for a fixed number of function invocations. To simulate failures at the given failure rate, the functions are killed at random times during the job execution. The result is shown in Figure 6. We observe that the recovery time depends on the function failure rate and the time at which the failure occurs during the function execution. The recovery time for the retry-based strategy is large when a failure occurs close to the function completion. Moreover, we observe that the total execution time of a failed job remains close to the ideal execution scenario of failure-free execution specifically when the failure rate is low. *Canary* reduces the recovery time of the failed function by up to 83% as compared to the default retry-based recovery strategy. Overall, we observe that for this experiment *Canary* reduces the recovery time by 82%, 81%, 79%, 83%, and 82% on average as compared to the default retry-based approach for DL, web service, Spark data mining, data compression, and graph search workloads, respectively. *Canary* ensures that the function is recovered from the latest checkpoint, thus reducing the recovery time and keeping it consistent regardless of when the failure occurs during the function execution.

3) Impact of Using Canary *on the Workload's Makespan:* We study the impact of *Canary* on the total execution time of the studied workloads. Figure 7 shows the result of the total execution time of *Canary* with the default retry-based approach for various failure rates using the studied DL workload. The replicated runtimes provide a quick way to restore the latest checkpoints of the failed functions. We observe that *Canary* is more effective than the retry-based approach as the failure rate increases and when a failure occurs towards the end of workload execution. We also compare the execution

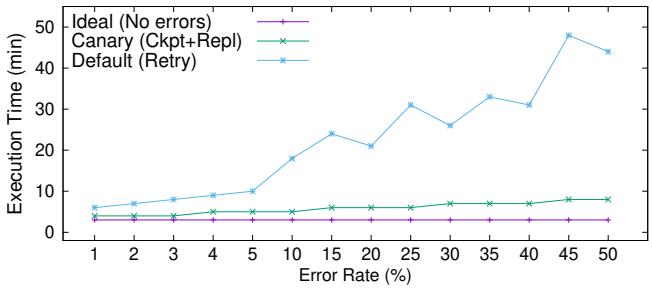

Fig. 7: Execution makespan of 100 function invocations for the DL workload with replication and checkpointing.

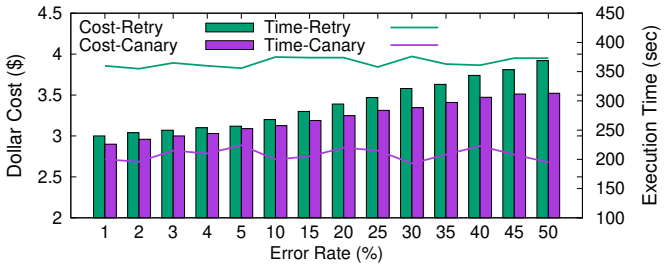

Fig. 8: Impact of failure on cost and time of training ResNet50 on CIFAR10 over 50 epochs.

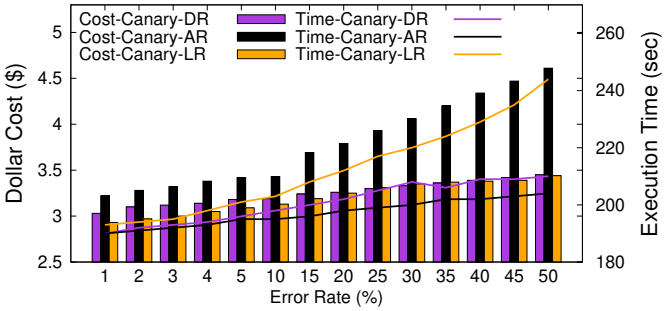

Fig. 9: Impact of replication on cost and time of training ResNet50 on CIFAR10 over 50 epochs with aggressive replication (AR), lenient replication (LR), and dynamic replication (DR).

time with the ideal scenario where there is no function failure. The retry-based strategy diverges from the ideal execution time as the failure rate increases, however, we observe that the execution time using *Canary* is comparable to the ideal execution time. Overall, *Canary* increases the execution time by 14% on average as compared to the ideal scenario without any failure. The overhead associated with *Canary* is because of the worst case scenario where the function fails right before a checkpoint is taken and recovers from the previous saved checkpoint. The retry-based recovery strategy performs the worse because of the loss of the entire computation of the failed function and restarting of the execution from the start upon failure. Our evaluation shows that *Canary* reduces the total execution time by up to 83% with a failure rate of 50% over the default retry-based recovery strategy. We observe similar performance trends in terms of the execution time for the web service and Spark data mining applications.

4) Impact of Using Canary *on Dollar Cost:* We perform a dollar cost analysis of using *Canary* by calculating the total cost of the launched functions and the replicated runtimes. We consider the pricing model of $0.000017 per second of execution, per GB of memory allocated from IBM Cloud Functions [11] as it is based on Apache OpenWhisk which we use for prototyping *Canary*. However, the pricing model of AWS Lambda [64] is comparable, i.e., ∼$0.0000167 per second of execution, per GB of memory allocated. We correlate the cost with the total job execution time. For our analysis, we consider the total execution time as the time from the first launched function to the completion of the last function. Moreover, the cost of concurrent functions is aggregated to represent the overall dollar value of a workload's execution.

Figure 8 shows the cost and execution time of *Canary* as compared to the retry-based recovery strategy. We observe that as the failure rate increases, the total cost for both *Canary* and the default strategy increases proportionally. Moreover, the difference between the cost of the retry-based strategy and *Canary* becomes larger with the increase in error rate. *Canary* has a lower cost by up to 12% as compared to the default retry-based strategy due to the replicated runtimes and checkpointing strategy. Overall, *Canary* improves application availability at an average additional cost of 8% as compared to the ideal scenario without any failure. We also observed that the cost of the retry-based strategy is much higher as compared to *Canary* for high failure rates. For the retry-based recovery strategy, functions failing close to the end of their execution incur much higher costs as they have to redo the entire execution from the beginning. In the case of *Canary*, the function is recovered from the latest available checkpoint and completes the remaining execution. We observe that the execution time for *Canary* is 43% less on average as compared to the retry-based technique demonstrating the benefits of *Canary* at a reduced cost even with the additional overhead of replicating the runtimes. The overhead of function checkpointing and data replication in *Canary* results in an additional cost that depends on the error rate, checkpoint location, network congestion, and the number of replicated runtimes.

We also evaluate the impact of replication on the cost and execution time of functions. We evaluate three replication strategies: dynamic, aggressive, and lenient replication. Dynamic replication (DR) is the default strategy of *Canary* in which the replication factor is dynamic and adjusted based on the failure rate. Aggressive replication (AR) uses a higher replication factor for each running job. Lenient replication (LR) maintains one active replica throughout the execution of each job. The results of this experiment are shown in Figure 9. We observe that when *Canary* increases the replication factor, the associated cost also increases because of concurrently running the additional runtimes with the application functions. LR strategy results in slightly lower cost as compared to the DR strategy, however, the job execution

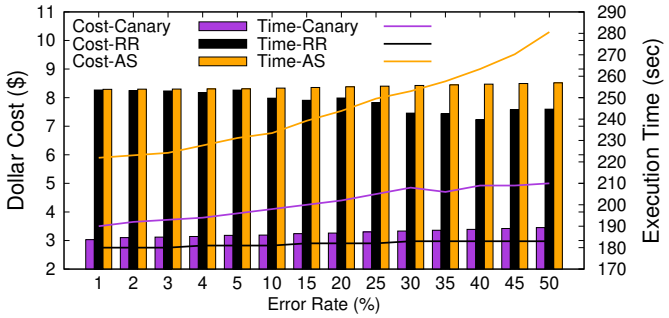

Fig. 10: Comparison of *Canary* with active-standby (AS) and request replication (RR).

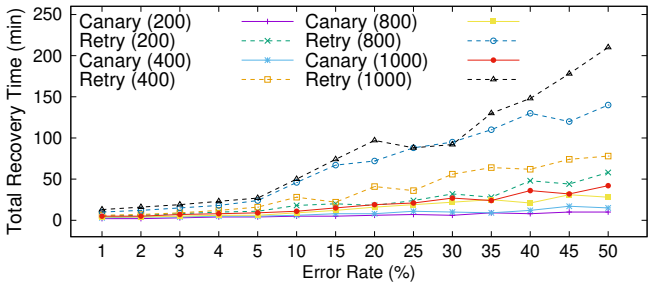

Fig. 11: Impact of *Canary* on recovery time with a cluster size of 16 and increased function invocations.

time with LR increases at a higher rate with the increase in the failure rate. Moreover, we observe a slower increase in the execution time using AR as compared to LR and DR. This trend shows that dynamic replication used by *Canary* scales better and provides better reliability as compared to the LR approach. AR yields a higher overall cost but has the lowest execution time. As more functions fail, the number of deployed replicas under the AR approach increasingly matches the number of failed functions. This trend results in fewer unused instances and a lower cost per replica. Overall, the DR approach outperforms AR and LR approaches by 25% and 2% on-average dollar cost savings, respectively.

5) Comparison of Canary *with State-of-the-Art Fault Tolerance Techniques:* We compare the performance of *Canary* with state-of-the-art fault tolerance techniques i.e., request replication (RR) [65] and active-standby (AS) [66]. RR launches multiple replicated functions for each given function based on the given replication factor. Similarly, AS creates two function instances; one for serving all requests and the other as standby. In our evaluation, we launch one replica per request. The incoming requests are forwarded to all functions and the first successful response is accepted and the rest are discarded. Figure 10 shows the comparison of *Canary* with RR and AS approaches. We observe that both RR and AS result in higher costs than *Canary* by up to 2.7× and 2.8×, respectively, because of launching additional functions as replicas or standby. As the error rate increases, the probability of active, standby, and replicas functions being killed at the same time increases, thus increasing the overall execution time and cost as failed functions must be restarted from the beginning. Similarly, we observed that the standby function stays dormant while replicas process incoming requests, hence, consuming additional system resources, resulting in resource contention, and increasing the resource requirements for functions execution. We observe that as the error rate increases, the execution time of *Canary* increases by 5% on average as compared to RR due to the checkpoint restore approach of *Canary*. The execution time of AS increases with the error rate because an increased number of failed functions are redeployed on the standby instances. Overall, the execution time of AS is up to 34% higher than *Canary*. This is because stateful functions depend on previous states for correct operation and functions are restarted as there is no checkpoint in the AS technique.

6) Impact of Scaling on Canary *Performance:* We increase the size of the computing cluster and the number of submitted jobs to observe the performance of *Canary*. We concurrently launch several hundred, i.e., 200, 400, 800, and 1000, functions and randomly kill the running functions that belong to various FaaS jobs and observe the total execution time of the submitted jobs. We also increase the failure rate proportional to the number of functions launched. Figure 11 shows the results of this experiment. We observe that as the number of functions increases, the total recovery time of the submitted batch of FaaS jobs remains fairly constant as compared to the default retry-based approach. The recovery time of *Canary* stays close to zero which matches the optimal failure-free scenario. However, with increased failure rates *Canary* experiences a slight increase in the recovery time due to recovery overhead. Our experiments include cases with node-level failures that lead to total loss of computation for the jobs scheduled on the failed node. For large function invocations, the retry-based strategy shows an interesting behavior, i.e., the recovery time depends on the time at which the function fails and if there is any node-level failure. For node-level failures, a large number of functions are restarted at the same time, hence, the recovery time for these functions overlap and is equal to the longest recovery time of any single failed function. Node-level failures in *Canary* are treated differently due to the availability of checkpoints in the shared storage system that is accessible throughout the cluster. Overall, in this experiment, we observe that *Canary* reduces the average recovery time by up to 80% as compared to the retry-based approach.

Next, we study the scalability of the studied approaches by increasing the cluster size from 1 to 16 nodes. In this experiment, we use a failure rate of 15% and a fixed number of function invocations, i.e., 5000. Figure 12 shows the results. We observe that as the cluster size increases, the total execution time of batch jobs decreases for all three execution scenarios. The performance of *Canary* is close to the ideal case, with an average increase of 2.75% in the execution time, when increasing the number of nodes from 1 to 16. However, *Canary* reduces the overall execution time by up to 17% as compared to the retry-based approach. Overall, we observe the scalability of 1.2×, 1.18×, and 1.10× for the ideal,

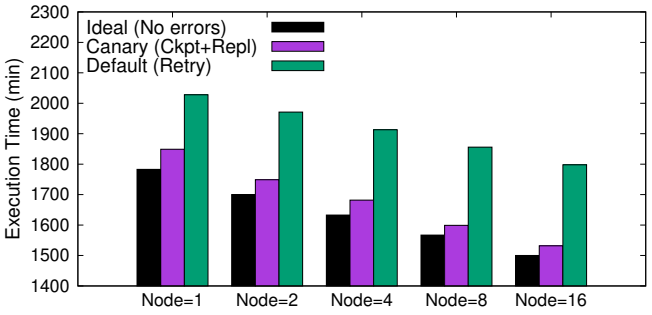

Fig. 12: Impact of *Canary* on recovery time with 5000 function invocation with a failure rate of 15%.

Canary, and the default retry-based approach, respectively, when increasing the number of nodes from 1 to 16.

E. Discussion

Existing FaaS platforms implement a retry-based recovery strategy for all failed functions which may not guarantee successful function execution. Moreover, using a retry-based approach re-executes the failed functions multiple times leading to significantly higher execution time and associated cost as compared to the proposed *Canary* framework. The performance of retry-based failure recovery is worse when functions fail frequently and towards the end of a function's execution. In this section, we discuss the implications of the replication and checkpointing techniques on the performance of *Canary* and analyze its potential benefits for both cloud and FaaS service providers.

1) Replicating Function Runtimes: Our evaluation shows that replicating execution runtime and critical application data significantly improves the recovery time of serverless functions. Replication provides warm containers to resume the execution of the failed functions. However, the benefits of replication are influenced by the number of replicas maintained per job. Our analysis of dynamic, aggressive, and lenient replication strategies shows that *Canary* achieves better performance by determining the number of replicas based on the error rate. Consequently, *Canary* yields a trade-off between time and cost of execution. Lenient replication incurs less computation overhead for the same cost as compared to the dynamic replication approach when the failure rate is low. This performance gap reduces as the number of failed functions increases because a workload with the lenient function replication strategy spends more time on starting the new replicas of function runtimes. The aggressive replication strategy spends slightly higher time for a significantly higher cost as compared to dynamic replication. Therefore, dynamic replication with *Canary* performs better than the aggressive and lenient replication strategies.

2) Checkpointing Function States and Data: The retry-based fault tolerance strategy forces re-executing a function from its first instruction. The use of checkpoints to store function states and critical data significantly reduces the job execution time in case of function failures. The frequency of checkpoints adds overhead to FaaS computing, however, it addresses the challenges of unpredictable system failures, such as network failures. The ideal scenario is to checkpoint function states and data right before a failure or state completion, and restart the failed function using the latest checkpoint. However, it is challenging to accurately predict these events. *Canary* maintains up-to-date checkpoints after successful completion of function states to ensure quick recovery of failed functions.

3) Benefits of Canary *for FaaS Platforms:* Fault tolerance and resiliency are key features for measuring the quality of services provided by the cloud platforms. *Canary* integrates replication and checkpointing techniques to ensure a reduced execution time. The traditional retry-based approach employed by the existing FaaS platforms leads to a higher job execution time, which negatively impacts time-sensitive applications and may violate their SLAs. Moreover, longer function execution requires occupying the same resources for the same job for a longer period of time. *Canary* addresses these issues and alleviates the challenges of resource scheduling of incoming jobs by significantly reducing the impact of failures thus freeing up expensive data center resources.

4) Benefits of Canary *for FaaS Users:* FaaS offers an attractive computing model for reducing the cost of using cloud resources without negatively impacting application performance. However, the cost benefits of deploying application on FaaS platforms is undermined by unexpected failures. The default retry-based strategies used at large by cloud providers significantly increase FaaS costs. *Canary* alleviates the burden of extending the expected duration of a job and enables FaaS users to reduce the function completion time as compared to the retry-based approach. Specifically, *Canary* improves the reliability of time-sensitive applications by reducing their failure recovery time.

VI. Related Work

In this section, we provide an overview of the existing efforts that are closely related to the contributions of this paper.

A. Stateful FaaS

Modern FaaS applications are composed of several closely connected components launched as functions. These components communicate and share states through an additional storage layer that requires fine-grained state management at a low cost [22]. A well-orchestrated state-sharing technique is required to avoid issues with non-atomic updates, concurrency control, duplication, etc. [67]. Existing research has addressed the design of such a data layer for stateful FaaS in three main directions, i.e., function composition, external storage, and low latency shared memory layer. State sharing between FaaS functions is achieved through function composition when two consecutive functions are executed such that the output of the first function is the input of the second function. This sharing technique is solely applicable when the output size remains within the quotas of the FaaS platform. When a large data transfer is required between functions, FaaS applications rely on external storage systems, such as AWS S3 [68], Google

Cloud Storage [69], and IBM Cloud Object Storage [70]. Despite their high data access latency, these solutions are used to facilitate the design of stateful applications and ensure data persistence in FaaS. In-memory KV stores, such as Redis [71] and MemcacheDB [72], are used to provide low latency and high bandwidth, but enabling persistent data storage in KV stores significantly increases access latency and reduces bandwidth. To facilitate function auto-scaling, FaaS platforms, such as Cloudburst [73], maintain states in auto-scaling and fault tolerant KV stores, e.g., Anna [74]. Using a distributed shared memory storage system provides a trade-off between latency and data size while improving state management [22], [75]. However, concerns regarding memory address space isolation [76] are not addressed in such approaches. Similarly, Faaslets [76] employs a WebAssembly software-fault isolation tool to provide isolation while sharing memory regions between serverless functions.

B. Fault Tolerance Approaches in Cloud Computing

Failures are addressed in cloud computing using approaches such as replication [77], checkpoint [78], checksum [79], self-healing [80], retry [81], safety-bag checks [82], task resubmission [83], etc. Fault tolerance techniques are grouped into two categories, i.e., proactive and reactive. Proactive fault tolerance [84] involves preemption and migration [85], self-healing [80], periodically reboot with a clean copy [86], or load balance when the resource utilization threshold is reached [87]. Reactive fault tolerance [88] includes techniques, such as, retries [81], task resubmission to the same or a different node [83], and reconfiguration [82]. Proactive and reactive fault tolerance techniques are often used in combination with each other to improve system reliability [89]. *Canary* combines both of these techniques by maintaining runtime replicas and preemptively saving checkpoints during FaaS job execution, and completing the remainder of the workload execution in a replica in case of failures.

C. Fault Tolerance in Stateful FaaS

The most widely-studied failures in serverless computing are the failures that are related to resource limitations on FaaS platforms [90]. Existing research addresses hardware and network issues, however, hardware failures are mainly explored from the perspective of cloud providers [30]. Serverless platforms have built-in fault tolerance techniques, such as checkpointing [91], retries [17], and object replication [92]. However, function failure can still occur due to memory, timeout, network, concurrency, and user quotas. These failures are more detrimental to stateful applications due to the cost associated with the loss of computation. Existing efforts provide fault tolerance and reliability to stateful applications by integrating object storage, KV stores, or optimizing file systems [93] to facilitate function retries or resubmission. Moreover, log-based techniques that monitor execution logs have been explored for fault tolerance and data consistency [94]. Monitoring logs facilitate the detection of function states and the coordination of chained and stateful FaaS applications. To further optimize state management for data consistency guarantees, transaction processing techniques are used to control read and write operations on intermediate data [95]. These techniques involve data staging and committing after transaction validation. Nevertheless, transaction processing techniques add an overhead to the target system [96]. Similarly, as more FaaS applications depend on network-based services, node failures cause requests to be re-executed multiple times [97]. Two strategies, i.e., request replication and active standby have been proposed [66] to improve fault tolerance. Request replication involves having multiple replicas to execute the same request and returning results to the client once results are returned from any replica. Active standby refers to maintaining one passive instance whenever there is an active function. The passive function is activated when the function fails and triggers the creation of a new passive instance. These approaches are better than function retries but can yield a higher cost as more requests are submitted. Request replication results in multiple unused function instances and one passive instance may become a bottleneck with multiple consecutive function failures. The aforementioned shortcomings in the existing approaches require developing techniques to improve fault tolerance and reliability in FaaS. The dynamic replication and checkpointing approaches in *Canary* adjust the replication factor and checkpointing frequency to provide improved fault tolerance and reduced cost as compared to these approaches.

VII. Conclusion and Future Work

Function-as-a-Service (FaaS) platforms use the retry-based failure recovery strategy which incurs significant overhead for executing stateful applications where each application stage depends upon the output of previous stages. In this paper, we address this challenge and present a fault-tolerant and resilient stateful FaaS framework, *Canary*, that extends the existing FaaS platforms by adding new software modules for storing function states, replicating function runtimes, and checkpointing critical data for faster failure recovery. Our evaluation using diverse FaaS applications shows that *Canary* can tolerate large failures and reduces the recovery time and dollar cost by up to 83% and 12%, respectively over the default retry-based recovery strategy. Moreover, *Canary* provides improved application availability at the additional average execution time and cost overhead of 14% and 8%, respectively over the ideal scenario that does not incur any failure. In our future work, we will extend the *Canary* framework to predict and proactively mitigate failures, and explore advanced techniques, such as request and function replication, to recover from failures. Moreover, we plan to incorporate user requirements into the failure recovery strategy to maximize the performance and cost benefits of using FaaS platforms.

Acknowledgment

This work is supported in part by the National Science Foundation (NSF) under Awards No. 2106634 and 2106635. Results presented in this paper are obtained using the Chameleon and CloudLab testbeds supported by the NSF.

REFERENCES

[1] D. Jackson and G. 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)*, 2018, pp. 154–160.

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, p. 107–113, January 2008.

[3] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[4] D. Chahal, R. Ojha, M. Ramesh, and R. Singhal, "Migrating large deep learning models to serverless architecture," in *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2020, pp. 111–116.

[5] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, 2018, pp. 257–262.

[6] D. Chahal, M. Ramesh, R. Ojha, and R. Singhal, "High performance serverless architecture for deep learning workflows," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2021, pp. 790–796.

[7] A. J. Younge, K. Pedretti, R. E. Grant, and R. Brightwell, "A tale of two systems: Using containers to deploy hpc applications on supercomputers and clouds," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017, pp. 74–81.

[8] Y. Wang, R. T. Evans, and L. Huang, "Performant container support for hpc applications," in *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, ser. PEARC '19. New York, NY, USA: Association for Computing Machinery, 2019.

[9] K. Assogba, M. Arif, M. M. Rafique, and D. S. Nikolopoulos, "On realizing efficient deep learning using serverless computing," in *2022 IEEE/ACM 22nd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022.

[10] K. Z. Ibrahim, T. Nguyen, H. A. Nam, W. Bhimji, S. Farrell, L. Oliker, M. Rowan, N. J. Wright, and S. Williams, "Architectural requirements for deep learning workloads in hpc environments," in *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2021, pp. 7–17.

[11] *IBM Cloud Functions*, August 2022. [Online]. Available: https://www.ibm.com/cloud/functions

[12] *AWS Lambda*, August 2022. [Online]. Available: https://aws.amazon.com/lambda/

[13] *Google Cloud Functions*, August 2022. [Online]. Available: https://cloud.google.com/functions

[14] *Microsoft Azure Functions*, August 2022. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/

[15] *Apache OpenWhisk*, August 2022. [Online]. Available: https://openwhisk.apache.org/

[16] A. Ellis *et al.*, "Openfaas: Serverless functions made simple," 2019. [Online]. Available: https://github.com/openfaas/faas

[17] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro, "A fault-tolerance shim for serverless computing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020.

[18] H. Lee, K. Satyam, and G. Fox, "Evaluation of production serverless computing environments," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 442–450.

[19] *AWS Lambda Quotas*, August 2022. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html

[20] *IBM Action Limits*, August 2022. [Online]. Available: https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-limits#limits_syslimits

[21] K. Alpernas, A. Panda, L. Ryzhyk, and M. Sagiv, "Cloud-scale runtime verification of serverless applications," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 92–107.

[22] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, "On the faas track: Building stateful distributed applications with serverless architectures," in *Proceedings of the 20th International Middleware Conference*, ser. Middleware '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 41–54.

[23] *Handle Errors in Serverless Applications*, August 2022. [Online]. Available: https://aws.amazon.com/getting-started/hands-on/handle-serverless-application-errors-step-functions-lambda/

[24] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2010.

[25] N. El-Sayed and B. Schroeder, "Reading between the lines of failure logs: Understanding how hpc systems fail," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1–12.

[26] M. Arif, K. Assogba, M. M. Rafique, and S. Vazhkudai, "Exploiting CXL-based memory for distributed deep learning," in *2022 ACM 51st International Conference on Parallel Processing (ICPP)*, 2022.

[27] M. Arif, M. M. Rafique, S.-H. Lim, and Z. Malik, "Infrastructure-aware tensorflow for heterogeneous datacenters," in *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2020, pp. 1–8.

[28] Y. Bouizem, D. Dib, N. Parlavantzas, C. Morin, and F. Lahfa, "Request Replication for FaaS Fault Tolerance," Inria, Research Report RR-9444, Jan. 2022. [Online]. Available: https://hal.inria.fr/hal-03510322

[29] G. Wang, L. Zhang, and W. Xu, "What can we learn from four years of data center hardware failures?" in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017, pp. 25–36.

[30] K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 193–204.

[31] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, p. 350–361, aug 2011.

[32] K. S. Yim, "Evaluation metrics of service-level reliability monitoring rules of a big data service," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, 2016, pp. 376–387.

[33] R. Potharaju and N. Jain, "Demystifying the dark side of the middle: A field study of middlebox failures in datacenters," in *Proceedings of the 2013 Conference on Internet Measurement Conference*, ser. IMC '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 9–22.

[34] C. Li, C. Zhao, H. Yan, and J. Zhang, "Event-driven fault tolerance for building nonstop active message programs," in *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, 2013, pp. 382–390.

[35] I. I. Yusuf and H. W. Schmidt, "Parameterised architectural patterns for providing cloud service fault tolerance with accurate costings," in *Proceedings of the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering*, ser. CBSE '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 121–130.

[36] T. Islam and D. Manivannan, "Predicting application failure in cloud: A machine learning approach," in *2017 IEEE International Conference on Cognitive Computing (ICCC)*, 2017, pp. 24–31.

[37] L. Toka, G. Dobreff, D. Haja, and M. Szalay, "Predicting cloud-native application failures based on monitoring data of cloud infrastructure," in *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2021, pp. 842–847.

[38] *For the Love of Serverless*, August 2022. [Online]. Available: https://newrelic.com/sites/default/files/2021-08/serverless-benchmark-report-aws-lambda-2020.pdf

[39] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *The Journal of Supercomputing*, vol. 65, no. 3, pp. 1302–1326, 2013.

[40] K. Keller and L. Bautista-Gomez, "Application-level differential checkpointing for hpc applications with dynamic datasets," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 52–61.

[41] K. Parasyris, K. Keller, L. Bautista-Gomez, and O. Unsal, "Checkpoint restart support for heterogeneous hpc applications," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 242–251.

[42] O. Subasi, G. Yalcin, F. Zyulkyarov, O. Unsal, and J. Labarta, "Designing and modelling selective replication for fault-tolerant hpc appli-

cations," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017, pp. 452–457.

[43] Z. Hussain, T. Znati, and R. Melhem, "Partial redundancy in hpc systems with non-uniform node reliabilities," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 566–576.

[44] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in *Proceedings of the 21st International Middleware Conference*, ser. Middleware '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–13.

[45] P. Karhula, J. Janak, and H. Schulzrinne, "Checkpointing and migration of iot edge functions," in *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking*, ser. EdgeSys '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 60–65.

[46] *Apache Ignite*, August 2022. [Online]. Available: https://ignite.apache.org/

[47] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbah, A. Rocha, and J. Stubbs, "Lessons learned from the chameleon testbed," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 219–233. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/keahey

[48] K. Hightower, B. Burns, and J. Beda, *Kubernetes: Up and Running Dive into the Future of Infrastructure*, 1st ed. O'Reilly Media, Inc., 2017.

[49] J. Turnbull, *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[50] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB: the definitive guide: time to relax*. O'Reilly Media, Inc., 2010.

[51] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, "Rfc3530: Network file system (nfs) version 4 protocol," 2003.

[52] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[53] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 169–182.

[54] M. D. Flouris and E. P. Markatos, "The network ramdisk: Using remote memory on heterogeneous nows," *Cluster computing*, vol. 2, no. 4, pp. 281–293, 1999.

[55] *Apache Spark*, August 2022. [Online]. Available: https://spark.apache.org/

[56] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: A serverless benchmark suite for function-as-a-service computing," in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 64–78.

[57] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[58] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[59] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *ATT Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2010.

[60] *Apache Spark*, August 2022. [Online]. Available: https://www.postgresql.org/

[61] *US Census Data*, August 2022. [Online]. Available: https://www2.census.gov/programs-surveys/popest/technical-documentation/file-layouts/2010-2017/cc-est2017-alldata.pdf

[62] D. Harnik, R. Kat, D. Sotnikov, A. Traeger, and O. Margalit, "To zip or not to zip: Effective resource usage for Real-Time compression," in *11th USENIX Conference on File and Storage Technologies (FAST 13)*. San Jose, CA: USENIX Association, Feb. 2013, pp. 229–241.

[63] *igraph – Network analysis software*, August 2022. [Online]. Available: https://igraph.org/

[64] *AWS Lambda Pricing*, August 2022. [Online]. Available: https://aws.amazon.com/lambda/pricing/

[65] C. Engelmann and S. Böhm, "Redundant execution of hpc applications with mr-mpi," in *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*, 2011, pp. 15–17.

[66] Y. Bouizem, N. Parlavantzas, D. Dib, and C. Morin, "Active-standby for high-availability in faas," in *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, ser. WoSC'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 31–36.

[67] *Durable Functions: Semantics for Stateful Serverless*, August 2022. [Online]. Available: https://angelhof.github.io/files/papers/durable-functions-2021-oopsla.pdf

[68] *Amazon S3*, August 2022. [Online]. Available: https://aws.amazon.com/s3/

[69] *Google Cloud Storage*, August 2022. [Online]. Available: https://cloud.google.com/storage

[70] *IBM Cloud Object Storage*, August 2022. [Online]. Available: https://www.ibm.com/cloud/object-storage

[71] *RedisLabs - Redis*, August 2022. [Online]. Available: https://redis.io/

[72] Memcached, "Memcached - a distributed memory object caching system," https://memcached.org/, accessed: 2022-08-26.

[73] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 2438–2452, jul 2020.

[74] C. Wu, J. M. Faleiro, Y. Lin, and J. M. Hellerstein, "Anna: A kvs for any scale," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 2, pp. 344–358, 2021.

[75] D. Barcelona-Pons, P. Sutra, M. Sánchez-Artigas, G. París, and P. García-López, "Stateful serverless computing with crucial," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 3, mar 2022.

[76] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 419–433. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shillaker

[77] N. Mansouri, "Adaptive data replication strategy in cloud computing for performance improvement," *Frontiers of Computer Science*, vol. 10, no. 5, pp. 925–935, 2016.

[78] M. Amoon, N. El-Bahnasawy, S. Sadi, and M. Wagdi, "On the design of reactive approach with flexible checkpoint interval to tolerate faults in cloud computing systems," *Journal of Ambient Intelligence and Humanized Computing*, vol. 10, no. 11, pp. 4567–4577, 2019.

[79] S. Chinnathambi, A. Santhanam, J. Rajarathinam, and M. Senthilkumar, "Scheduling and checkpointing optimization algorithm for byzantine fault tolerance in cloud clusters," *Cluster Computing*, vol. 22, no. 6, pp. 14 637–14 650, 2019.

[80] R. K. Devi and M. Muthukannan, "Self-healing fault tolerance technique in cloud datacenter," in *2021 6th International Conference on Inventive Computation Technologies (ICICT)*, 2021, pp. 731–737.

[81] M. A. Mukwevho and T. Celik, "Toward a smart cloud: A review of fault-tolerance methods in cloud systems," *IEEE Transactions on Services Computing*, vol. 14, no. 2, pp. 589–605, 2021.

[82] P. K. Patra, H. Singh, and G. Singh, "Fault tolerance techniques and comparative implementation in cloud computing," *International Journal of Computer Applications*, vol. 64, no. 14, 2013.

[83] K. Plankensteiner, R. Prodan, and T. 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*, 2009, pp. 313–320.

[84] J. Liu, S. Wang, A. Zhou, S. A. P. Kumar, F. Yang, and R. Buyya, "Using proactive fault-tolerance approach to enhance cloud service reliability," *IEEE Transactions on Cloud Computing*, vol. 6, no. 4, pp. 1191–1202, 2018.

[85] C. Engelmann, G. R. Vallee, T. Naughton, and S. L. Scott, "Proactive fault tolerance using preemptive migration," in *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009, pp. 252–257.

[86] S. M. A. Ataallah, S. M. Nassar, and E. E. Hemayed, "Fault tolerance in cloud computing - survey," in *2015 11th International Computer Engineering Conference (ICENCO)*, 2015, pp. 241–245.

[87] S. Chakravorty, C. Mendes, and L. V. Kale, "Proactive fault tolerance in large systems," in *HPCRI Workshop in conjunction with HPCA*, vol. 2005. Citeseer, 2005, pp. 1–7.

[88] E. AbdElfattah, M. Elkawkagy, and A. El-Sisi, "A reactive fault tolerance approach for cloud computing," in *2017 13th International Computer Engineering Conference (ICENCO)*, 2017, pp. 190–194.

[89] A. Power and G. Kotonya, "A microservices architecture for reactive and proactive fault tolerance in iot systems," in *2018 IEEE 19th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*, 2018, pp. 588–599.

[90] S. Ginzburg and M. J. Freedman, "Serverless isn't server-less: Measuring and exploiting resource variability on cloud faas platforms," in *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, ser. WoSC'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 43–48.

[91] W. Zhang, V. Fang, A. Panda, and S. Shenker, "Kappa: A programming framework for serverless computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020.

[92] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, "On the faas track: Building stateful distributed applications with serverless architectures," in *Proceedings of the 20th International Middleware Conference*, 2019.

[93] J. Schleier-Smith, L. Holz, N. Pemberton, and J. M. Hellerstein, "A faas file system for serverless computing," *CoRR*, vol. abs/2009.09845, 2020. [Online]. Available: https://arxiv.org/abs/2009.09845

[94] Z. Jia and E. Witchel, "Boki: Stateful serverless computing with shared logs," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 691–707.

[95] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu, "Fault-tolerant and transactional stateful serverless workflows," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 1187–1204.

[96] M. de Heus, K. Psarakis, M. Fragkoulis, and A. Katsifodimos, "Distributed transactions on serverless stateful functions," in *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 31–42.

[97] S. G. Kulkarni, G. Liu, K. K. Ramakrishnan, and T. Wood, "Living on the edge: Serverless computing and the cost of failure resiliency," in *2019 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, 2019, pp. 1–6.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

Function-as-a-Service (FaaS) platforms are popular for hosting stateful and stateless applications due to their ease of deployment, scalability, and minimal management overhead. However, failures in FaaS platform make it unreliable for guaranteeing function execution and ensuring performance requirements. This project aims to address the reliability of FaaS platforms and develop a framework, called Canary, that mitigates the impact of failures, improves application availability and reduces the overhead of function restart. Here is a brief description of the testbed used in this work:

(1) The current version of the project is tested on bare-metal servers and on cloud infrastructure with Ubuntu 18.04 LTS OS and Apache OpenWhisk as serverless platform. Scalability experiments are conducted in clusters of 2, 4, 8 and 16 nodes.
(2) The evaluation testbed deployment is automated along with installation of all dependencies including Kubernetes (*v1.21*), Docker (*v20.10.16*), Helm (*v2.16.1*), MongoDB (*v5.0.8*), and OpenWhisk CLI (*v1.2.0*).
(3) FaaS functions are launched on Docker containers that are deployed in Kubernetes pods using the default 'one-container-per-pod' Kubernetes model.
(4) We built Docker images for each workload based on the dependencies of each application.
(5) We built a custom Docker image *'hpdsl/canary:dltrain'* for deep learning training by merging Apache OpenWhisk Python 3 action runtime image with *'tensorflow/tensorflow:2.4.1'* Docker image. Our deep learning workload consists of training ResNet50 model with MNIST dataset.
(6) We built a custom Docker image *'hpdsl/canary:dbquery'* for web database query including Apache OpenWhisk and *'psycopg2 v2.9.3'* as PostgreSQL database adapter. We store the queried data in a MongoDB database. The database is deployed as a docker container launched during the deployment of the OpenWhisk platform.
(7) We built a custom Docker image *'hpdsl/canary:sparkdiversity'* for US Census data mining including Apache OpenWhisk and a jar file with Apache Spark *v3.0.0*.

We consider three scenarios to evaluate the performance of Canary:

(1) *Ideal scenario* experiences no function failure in the FaaS platform and applications execute without function restart.
(2) *Default scenario* experiences frequent function failures resulting in function restarts causing applications to experience loss of computation, states, and data.
(3) *Canary* illustrates our framework and mitigates the impact of functions failures.

We have conducted the following experiments to evaluate Canary:

(1) We run three stateful applications on FaaS to compare the recovery time of the three studied scenarios and reported the averages of ten executions for each experiment.
(2) We evaluated the total execution time of the three studied scenarios which includes the time taken to recover from failures.
(3) We evaluated the cost of the three studied scenarios in terms of dollar cost incurred to recover from failures at various errors rates.
(4) We evaluated the impact of runtime replication and checkpointing on the function recovery time for failed functions at random times during the lifespan of an application.
(5) We evaluated the impact of batch jobs on the total execution time for the three studied scenarios by varying the total number of functions invocations.
(6) We conducted several scaling experiments to evaluate the performance of three studied scenarios by increasing the number of cluster nodes from two to sixteen.

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1
Persistent ID: `https://github.com/hpdsl/canary`
Artifact name: GitHub Repository for Canary

Artifact 2
Persistent ID: `https://hub.docker.com/r/hpdsl/canary`
Artifact name: Docker Repository for Canary

Reproduction of the artifact with container: The experimental testbed is based on a distributed Apache OpenWhisk cluster orchestrated with Kubernetes. The deployment instructions are included in our GitHub repository. The container images are listed in the README file which also explains how to execute the workloads that we have used to evaluate the proposed framework.