Proactive Stateful Fault-Tolerant System for Kubernetes Containerized Services

MINH-NGOC TRAN, XUAN TUONG VU, AND YOUNGHAN KIM, (Member, IEEE)
School of Electronic Engineering, Soongsil University, Seoul 06355, South Korea
Corresponding author: Younghan Kim (younghak@ssu.ac.kr)

This work was supported in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grants through the Korea Government (MSIT), Development of Fast and Automatic Service Recovery and Transition Software in Hybrid Cloud Environment, under Grant 2022-0-01015, Grant 6GRC, and Grant 2020-0-00946.

ABSTRACT Recently, the development of Kubernetes (K8s) containerization platform has enabled cloud-based, lightweight, highly scalable, and agile services in both general and telco use-cases. Ensuring high availability, reliable and continuous containerized services is a major requirement of service providers to provide fault-tolerance, transparent service experiences to end-users. To satisfy this requirement, fault prediction and proactive stateful service recovery features must be applied in cloud systems. Prior proactive failure recovery approaches mostly focused on either improving fault prediction performance based on different machine learning time series forecasting techniques or optimizing recovery service placement after fault prediction. However, a mechanism that enables stateful containerized service migration from the predicted faulty node to the healthy destination node has not been studied. Service migration in previous proactive works is only simulated or performed by virtual machine (VM) migration techniques. In this paper, we propose a proactive stateful fault-tolerant system for K8s containerized services that pipelines a Bidirectional Long Short-Term Memory (Bi-LSTM) fault prediction framework and a novel K8s stateful service migration mechanism for service recovery. Experimental results show how the Bi-LSTM model improved prediction performance against other time-series forecasting models used prior proactive works. We then combined the Bi-LSTM fault prediction framework with both the default K8s and our stateful migration mechanisms. The comparison between these two proactive systems proves our system efficiency in terms of reducing Quality of Service (QoS) violation percentage and service recovery time.

INDEX TERMS Containerization, proactive fault-tolerant, Kubernetes.
faults which can lead to services’ QoS violation have been addressed in prior proactive works. These range from overloading resources (CPU, memory, network), resources contention, and overheating node, to sudden high workload traffic. The similarity between these faults is that they can be analyzed from current and historical monitoring data to predict the next faulty moments. Hence, several time-series forecasting neural networks such as Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU) have been utilized for this task [2], [3]. Other techniques such as integer linear programming [4], [5], Bayesian classification [6], k-means [7], or reinforcement learning [8], [9] were also used in previous proactive approaches. However, these methods are not generalizable enough or are slow to adapt in volatile cloud environments [3]. Therefore, we used a time-series forecasting model for our proactive system. Since our work is not focused on designing new machine learning models for optimization, we chose the current latest time series forecasting model for our fault prediction framework, which is Bi-LSTM, to improve previous works’ prediction accuracy. In this paper, we applied the Bi-LSTM for predicting the overloading CPU usage fault type as a representative use-case of our proactive system for other mentioned faults above because data for other faults prediction can also be modeled as a time-series prediction problem.

For the second stage – preemptive service recovery, most of the recent proactive approaches focus on solving the migrated service placement problem (choosing which service to migrate or where to migrate the service in the faulty node). While this problem has been well studied, how to perform service migration from one node to another node is usually not the main concern in prior proactive works. This process is either simulated or targets VM-based systems using VM live migration technology. In recent years, considering containerization technology is rapidly replacing VM as the application deployment choice [10], [11], migration technology needs to support containerization systems. Besides, stateful migration is a vital requirement since many containerized services such as video streaming, database, message queuing services, augmented reality, etc., require up-to-date context, real-time and historical data to create meaningful experiences. In case of migration due to the proactive fault avoidance method, without the stateful recovery feature, restarting these services from scratch will cause a long booting time, or long service completion time followed by a long prediction ahead time interval to ensure the migrated service is ready at the new node when the real fault happens. However, setting such a large prediction time interval might degrade the prediction model performance, therefore, causing more QoS violations. Therefore, a stateful migration mechanism for containerized service is necessary to enable an efficient proactive fault-tolerant system for cloud-native cloud-edge computing nowadays.

Similar to proactive fault-tolerant systems for VM-based service, current stateful migration techniques for containers can also be applied to the container-based proactive system. However, most of these techniques only support Docker container runtime, which is not enough because container services are normally deployed and managed by a container orchestration platform. Considering K8s is the most popular container platform nowadays, the stateful migration techniques need to be integrated into it. K8s has its own StatefulSet feature to support stateful services. However, this feature only retains the state of the services’ data storage. To avoid long startup time and service restarting from scratch when recovering a service at a new node, a mechanism that can checkpoint and restore the in-memory booting configuration state and running task execution state should be integrated into K8s.

Apart from the default K8s solution, there are only two research that integrated the stateful migration feature into K8s. The first work [12] only snapshot and transfer the volume that stores the service data. To retain the in-memory state, the application might need to be redesigned to store this in-memory state on this volume. This design might cost application developers considerable efforts to redesign their applications; therefore, it is not reliable. The second work [13] proposed MyeDrive, which creates an agent in each container and an additional container per application pod. The more pods/containers are deployed in the system, the more resource overhead this framework will cause. Besides, the scope of these two works is limited to single-cluster scenarios. In real cloud-edge deployment, a multi-clusters scenario solution is required.

Due to these reasons, in this paper, we propose a proactive stateful fault-tolerant system for K8s containerized services. The contributions of this work are as follows:

- A novel K8s integrated stateful service migration mechanism which adds in-memory booting and running state support besides the default storage state support.
- An architecture of a proactive stateful containerized services recovery system that pipelining a Bi-LSTM fault prediction framework (with resource overload fault as the example use-case) and the K8s stateful migration framework to avoid service QoS latency violation.
- Our experimental results confirm the combination of these two frameworks’ effectiveness in avoiding QoS latency violation compared with previous machine learning techniques and the default K8s migration method.

The rest of this paper is organized as follows: Section II discusses the related work. Section III describes the proactive stateful fault-tolerant system. Section IV shows our system implementation and evaluation. Finally, Section V concludes the paper.

II. RELATED WORKS
A. PREVIOUS PROACTIVE FAULT-TOLENTANT RESEARCH
Proactive fault-tolerant systems for cloud applications aim to avoid service performance degradation caused by different kinds of faults. Proactive approaches achieve this by predicting these faults in advance and then pre-migrating or
pre-scheduling services from predicted fault nodes to other healthy nodes. In recent years, several proactive approaches using different machine learning techniques and heuristic algorithms have been proposed to optimize this process. These works use various algorithms and techniques to optimize one or both stages of a proactive system: fault prediction and preemptive service recovery.

The applied methods for optimizing fault prediction performance range from heuristic modeling, integer linear programming, and machine learning to reinforcement learning. In [14] and [15], the authors modeled nodes’ CPU temperature over time to predict overheating nodes. The Dynamic Fault-Tolerant Migration method proposed in [5] used the integer linear programming approach to identify nodes’ bottlenecks issue based on monitored traffic. Classification methods such as Bayesian and k-means were used in [6] and [7] to detect overloading nodes based on monitored CPU and memory usage from all machines. Also using these resource data, time-series forecasting models (LSTM [2] and GRU [3]) were utilized to predict multiple kinds of faults in the future, such as CPU over-utilization, abnormal memory, disk usage, or network overload. In a proactive architecture for distributed edge network function virtualization services [16], support vector machine and random forest algorithms were used to predict server machines failures based on the temperature, humidity, and working hours data from previous failures. Meanwhile, Double Deep Q-Learning [9] is the proactive approach that applied reinforcement learning for analyzing a large continuous real-time network state space, including nodes’ resources, link bandwidth, and incoming traffic workload rate to prevent services unavailability from node/link failures and high network traffic. CMFog [8] is another reinforcement learning-based approach that pre-migrates services based on user mobility. From these approaches for fault prediction tasks, heuristic algorithms often struggle to scale and adapt in heterogeneous systems, while reinforcement learning has a slow adaptation rate and high computation and training time. Considering most of the data used for predicting different kinds of faults mentioned above can be monitored and collected as time-series data (CPU, memory, network bandwidth, traffic workload), machine learning time series forecasting models are the viable choices for almost all kinds of faults. As our work is not focused on designing new machine learning algorithms for fault prediction, we apply one of the current state-of-the-art time series forecasting models that has not been used by any prior proactive works, which is Bi-LSTM. Unlike LSTM and GRU, this model can aggregate input information in a bidirectional way which can result in better prediction performance. In our experiments, we apply the Bi-LSTM model for overloading CPU resource usage fault prediction as a representative use-case for other faults, which can also be modeled as time-series problems. We compare the Bi-LSTM against other time-series models that were used in prior proactive works and fault prediction works: LSTM [2], GRU [3], Convolutional neural network LSTM (CNN-LSTM) [17]. For the preemptive service recovery stage, multiple objectives decision-making algorithms were utilized to select the optimal placement for the migrated services. In [14], after getting overheating node prediction from the fault prediction stage, integer linear programming is used to choose the destination node that maximizes service providers’ profit and minimizes migration cost. In [4], after user mobility prediction, another integer linear programming model is used to choose a destination VM that can maximize the service accepted requests and minimize the user latency. Another decision-making algorithm popularly used by prior approaches is particle swarm optimization. It was used in [15] and [7] to migrate the service from the overheated/overloaded node to the new node which minimizes migration cost and maximizes resource utilization. Besides, reinforcement learning is also a well-known method that combines both proactive fault-tolerant stages into one model [8], [9]. Other prior proactive approaches either use fault prediction models to predict healthy nodes for migration [2], [3] or use a greedy-based algorithm [16]. Based on this literature review, we notice that all these previous works focused on service placement algorithms and did not concern about the service migration mechanism to execute these placement decisions. The service migration process is either simulated or is simply mentioned that it is based on VM live migration technique. Hence, in this paper, instead of addressing a well-studied service placement problem, we focus on a stateful service migration mechanism for containerized systems that aligns with the containerization trend for cloud applications.

B. CURRENT CONTAINER STATEFUL MIGRATION TECHNIQUES
Since no stateful migration techniques for container applications were mentioned in prior proactive fault-tolerant works, in this part, we discuss some current standalone container stateful migration approaches.

Machen et al. [18] proposed a multi-layer framework container service live. The framework copies the base layer, which contains the operating system (OS) and kernel to all nodes. When a container service needs to be migrated, the application layer which contains the idle version of the service will be migrated first during runtime. Then the service is suspended, and only the instance layer needs to be transferred to the destination node. These splitting methods reduce migration downtime because only the state needs to be transferred. However, this framework only supports LXC container runtime. Checkpoint and Restore In Userspace (CRIU) [19] is another approach that enables container stateful migration problems by performing checkpoint and restore processes in the user space via available kernel interfaces. The checkpoint process uses the pctrace system call [20] to control the execution of a process. Then, it injects a parasite code to dump the memory pages of the process into image files from within the process’s address space. The container can be restored in another node with the previous state using these dumping files, and it has the same process identifier.
it had before checkpointing. Thanks to CRIU’s support for many popular container runtimes such as Docker, Containerd, runC, it is widely adopted by many containerized service migration works. ARNAB system in [21] enables transparent edge service continuity via double-tier migration. The first-tier hands-off user connectivity, while the second-tier leverages CRIU for a pre-copy stateful migration process. Container states are iteratively dumped at the source node and sent to the destination node. At the time of migration, only new states are transferred. Zakaria et al. [22] adopted CRIU to provide container migration in an OpenStack-based platform through which containerized applications can be deployed and managed at the edge/fog level. CloudHopper [23] used CRIU for live migration for applications that run over multiple interdependent containers across different clouds. Migration order is based on the size of containers in descending order. This system was implemented using runC container runtime with Docker images. In [24], the author migrated the container by transferring the base layer of the Docker images first, then using CRIU for state checkpoint and restoration following the pre-copy algorithm. To avoid the downtime caused by the stop-and-copy phase of the live migration approach using CRIU, the work [25] proposes to buffer requests at the destination node when the checkpoint and transferring phase of CRIU are being executed. After the container is restored at the destination node, the buffered request will be replayed. In [26], the authors tried to reduce the migration time by only using CRIU for the top storage layer of a Docker container as only this layer is changed during runtime.

From these works, we notice that although CRIU has been widely applied for stateful container migration, it only stays at the container runtime level currently. Meanwhile, deploying containerized applications on the cloud requires an orchestrator such as K8s. Thus, integration of this feature into K8s is required. Although K8s has provided StatefulSet feature, this feature only supports retaining the storage state of the services, which can be simply understood as the state of the database of the services. Each StatefulSet pod has its own identity and a linked Persistent Volume for the deployed service to store its data. This volume does not be deleted when the pod dies or shuts down. When the pod with the same identity is recovered, that Persistent Volume will be re-attached to the pod.

However, stateful migration is not only about the storage state. There are two other kinds of state: booting state and running state, which are stored in memory, not storage disk. The booting state consists of caches, configuration, library loading, and resource initialization setup. If this kind of state is not retained during service migration, the service will have a long startup time due to re-initializing the mentioned configuration above. Some example services that depend on the booting state are MongoDB, Redis, RabbitMQ, Java or Ruby-based applications, etc. On the other hand, the running state describes the current execution state of the service task. If this state is not retained during migration, the service will be restarted from scratch. Examples are live stream video service, multimedia processing service (video, image converter, analyzer), deep learning service, etc. These booting and running states have not been supported by K8s yet but can be retained by CRIU.

To our best knowledge, at the time of writing this paper, there were only two works that integrated stateful service migration into K8s. In the first work [12], the authors only focused on the storage state. They proposed to use the OverlayFS file system to snapshot the persistent volume to retain the state of the data inside it. To utilize this solution for booting and running state, the application developers might be required to redesign their services’ application layer to dump these in-memory states into the persistent volume, which is not practical. In the second work [13], the authors proposed a K8s stateful migration framework called MycDrive. This framework integrates the Distributed MultiThreaded Checkpointing (DMTCP) [27] technique into K8s. DMTCP is also a technique that supports retaining in-memory booting and running states. This work’s solution requires a DMTCP container running at each pod and an execution agent at each container to perform a stateful migration process. This design might create significant resource overhead when many pods and containers are running in a large containerized system. Besides, these two prior approaches only concern single-cluster scenarios. Stateful migration support for multi-cluster scenarios should also be considered, especially in ge-distributed environments where applications are normally deployed over different clusters.

Therefore, we propose our own K8s integrated stateful migration mechanism that supports both single and multi-clusters scenarios. Our solution utilizes the default K8s StatefulSet feature to retain storage state and integrates CRIU into K8s to retain booting and running states. Moreover, this paper is the first work that integrates a K8s stateful migration mechanism into a proactive fault-tolerant system. We evaluate the benefits of the K8s stateful migration technique to a proactive fault-tolerant system by comparing it with the default K8s migration system.

III. PROACTIVE STATEFUL FAULT-TOLERANT SYSTEM DESIGN

A. PROACTIVE STATEFUL RECOVERY ARCHITECTURE

Figure 1 shows the components of the system and their relationship. There are 3 main components:

- Monitor framework: this component collects resources metrics over clusters (e.g., CPU, memory, bandwidth, traffic load) and stores microservice mapping. The monitor framework is deployed at the cloud.
- Fault prediction framework: this component has two processes. The offline process is responsible for training the fault prediction model based on data fed from the monitor framework and updating it over time based on historical data. Bi-LSTM is our proposed machine learning model for time series data. The trained model is used in the online process to predict fault in advance to trigger the stateful migration.
mechanism through the task executor at the cloud and migration API converter at each edge cluster.

- Containerized stateful migration built-in feature: We modified the original K8s in both master and worker nodes to support this feature. The migration operator is added to the master node, and the migration executor is added at worker nodes.

In the next parts, we describe in detail the fault prediction framework and K8s stateful migration feature.

**B. FAULT PREDICTION FRAMEWORK**

1) **Bi-LSTM**

Recurrent neural network (RNN) has recently shown promising results in a variety of research domains, especially in time series forecasting. LSTM is a type of RNN that is capable of learning long-term dependencies to solve the drawback of the vanishing gradient problem in RNN by using sophisticated recurrent hidden and gated units known as memory cells. Each LSTM unit stores the network temporal state by three gates namely, input gate, output gate, and forget gate. The gate input controls the flow of inputting a new value into the cell. The forget gate regulates which data is removed from the cell. Finally, the output gate determines which information from the cell is used to compute the output activation of the LSTM unit. Output data is calculated by processing input data, previous hidden state, and previous cell state through these three gates. An LSTM model learns using forward dependencies in a window. Bi-LSTM is an extension of LSTM, learns using both forward and backward dependencies, and hence enriches the understanding of sequence patterns. Its model architecture is shown in Figure 2. It includes two LSTM networks stacked on top of each other. The first LSTM is trained on the original input sequence (forward). The reverse form of the input sequence is given to the second LSTM (backward). This bidirectional learning enhances the learning ability of the model compared with LSTM. Therefore, we chose the Bi-LSTM as the prediction model.

In this work, we will compare Bi-LSTM model’s performance with other RNN variants that have been used in previous cloud computing fault prediction works: LSTM [2], GRU [3], and CNN-LSTM [17]. Regarding model architecture, compared with Bi-LSTM, LSTM and GRU models have only one network stack containing its corresponding units. Unlike LSTM unit, GRU unit only has two gates: reset gate and update gate. The update gate of GRU unit aggregates LSTM unit’s forget gate and input gate. While the update gate decides how much previous data can be used in the future, the reset gate decides how much data can be removed. Unlike LSTM, GRU unit does not use the cell state; it processes input data and previous hidden state through its two gates to generate the output. CNN-LSTM, on the other hand, stacks the LSTM network on top of the CNN network. The CNN network consists of three layers, namely: convolutional, pooling, and fully connected. It is used to discover the ordered relationship in time-series data before feeding the processed input into the LSTM network [17]. As our work only aims to apply the current latest time-series prediction model and is not deep-learning algorithm focused, we only briefly introduce the architecture of these models. More details about these models can be found in related papers.

Bi-LSTM model can be used to solve any time-series prediction problem. Since the data which is used to predict different kinds of faults such as CPU, memory, disk input/output, or network bandwidth can be collected and modeled as time-series data as shown in several previous works [2], [3], [14], [15], the prediction problems for these kinds of fault are similar. Hence, we applied the Bi-LSTM model for only CPU overloading fault prediction as the representative use-case for other faults.

2) **MODEL TRAINING**

We used the VM workload dataset from Bitbrain cloud [28] to train our model. The Bitbrains dataset records CPU, memory, and network and disk input/output values in 2 months duration. Since our goal is to predict the CPU fault, we chose CPU metrics in terms of percentage from the dataset. We split the dataset into two sets: training set and test set with ratios 80% and 20%, respectively, then normalized them to the range (0, 1) before training. After that, we prepared historic subsequences from the normalized dataset by using the sliding window method as input sequences for our models, which uses n time steps as inputs to predict the next time step in one-step-ahead prediction. For this dataset, we used 12 past time steps sliding window to predict the next time step. Past time steps and many other hyperparameters are fine-tuned.
during the training process. Our model configuration details are shown in Table 1.

### 3) FRAMEWORK DESIGN

The fault prediction framework acts as a trigger mechanism that defines when the system needs to migrate the container service from the node that was predicted as the failure node to another healthy node. This module operates in two processes, online and offline. The algorithm for processing the Fault Prediction framework is shown in Algorithm 1.

In the offline process, at the beginning, the Bi-LSTM mode is trained using 80% of the Bitbrain dataset. Then, to avoid the performance of the model declining over time due to new patterns in the data flow, the Bi-LSTM model will be periodically retrained and updated using collected data in a pre-defined period.

In the online process, at each prediction time step $t$, a data processing unit will receive nodes’ CPU usage metrics from the monitor framework. This data will be combined with previous historical data based on the pre-defined sliding window value. Then this pre-processing data will be normalized and passed as input into the current Bi-LSTM model. The Bi-LSTM model, which is loaded by the model loader, will predict the average future resource usage of each node in the cluster in the next time step $t + 1$. The predicted values will be compared with the pre-defined CPU overloading threshold values to decide whether each node will be failed or not. In our testbed, to avoid QoS latency violation caused by an overloading node, a worker node with over 80% average CPU consumption in the next time step will be marked as the potential overloading node and vice versa. The Task Executor will call the Migration API to migrate container service on the predicted overloading node to one of the other predicted healthy nodes.

In our framework design, real-time data which is used for prediction will also be stored in an external database at the same time to retrain the model on both new and old data.

### C. KUBERNETES STATEFUL MIGRATION FEATURE

To achieve transparent service continuity following service interruption caused by faults in the cloud infrastructure, the primary goal of our system design is reallocating running pods from one node to another node within a K8s cluster or even between two different clusters without terminating and restarting pods. To do that, we leveraged the concept of CRIU state snapshot. Each application always runs as a process when being deployed. The application state in this paper is the process information that is dumped in an “image” file and can be restored so that it can be resumed at the exact moment before reallocating. In general, the pod migration process is performed via the following basic steps:

1. Request to migrate a pod are sent to the system.
2. Checkpoint and capture all containers’ states in the pod.
3. Transfer the pod state to the destination node.
4. Restore the new pod from the checkpoint.
5. Terminate the old pod.
6. Redirect service to the new pod.

Based on the K8s architecture, we designed and developed the novel stateful migration as an extended K8s feature to achieve the mentioned goal above. Our changes to the default K8s platform are:

- We developed the migration API converter that helps the K8s cluster listen to the external computing engine, such as the fault prediction framework in our case, to execute the migration process.
- We developed the Pod migration operator that helps the K8s cluster control plane verify the migration request. The request describes which application should be migrated, and it will be moved from which node to which node, inside or outside of the cluster.
• We developed the Pod migration executor at every worker node. We did this by extending the K8s agent. In detail, kubelet and the container runtime interface (CRI) source code are extended to call and implement container checkpoint and restore functions, which are powered by CRIU project.

1) STATEFUL MIGRATION FRAMEWORK COMPONENTS
   a: POD MIGRATION OPERATOR AT THE CONTROL PLANE
In our general system design in Figure 1, the Fault prediction framework decides when and where the pod migration will be triggered. We try to extend K8s to listen to the fault prediction framework to control and orchestrate the pod migration process without breaking the K8s architecture design. One way to do it is to follow the K8s operator pattern. Hence, we developed the pod migration operator following the K8s operator concepts, which consists of a custom resource and a controller. Figure 3 describes the detailed components inside the pod migration operator and how the pod migration operator performs its functionality. The Custom Resource Definition (CRD) is a mechanism that allows users to define their data types in K8s. This allows us to specify the desired state through the API, and a custom controller to achieve that state [29]. As the custom resource is the API extension mechanism in K8s, we designed the Pod-migration CRD and the Pod-migration CRD controller to perform the pod migration life cycle. The Pod-migration CRD defines the main information of the service migration process as follows:

1) Source pod name: This is the name of the running pod that needs to be migrated. The controller uses this field to check whether the pod is running and verify the source node in which the source pod is running. If the given pod is found running in the cluster, the controller can start checkpointing it for migration.

2) Destination host: This is the name of worker nodes to where the pod is requested to be migrated to. The destination host will be invoked to perform the initial steps and then wait to restore the pod from the checkpoint information received from the source node.

3) Snapshot path: This is the path in which the checkpoint dump information will be stored. The default snapshot path is set in the case of live migration. However, if we do not need to restore the application right after the checkpoint is completed or restoring multiple pods from one checkpoint is required; the user can define the snapshot path.

Through the API server, the Pod-migration controller can process these pieces of information to guide the pod migration executors at the worker nodes to complete the migration process’s actions. Consequently, a user or an external analysis framework, such as the fault prediction framework in our case, can trigger the migration process by sending the API server a request to create or modify the pod migration CRD.

Because the K8s API server can only understand the request in a very strict template, we developed a migration API converter at each K8s cluster to provide a RESTFUL endpoint that can translate simple requests such as a RESTFUL request to the K8s API server language. Every time the migration request is sent to the migration API converter, it will adjust the pod-migration CRD. The Pod-migration CRD controller watches the custom resources type and takes the application-specific actions to make the current state match the desired state in that resource. For example, in the case of checkpointing, the controller will monitor the state of the pod. If the application has not been checkpointed, the controller will send a trigger to the migration executor to checkpoint the pod by changing the pod metadata.

b: POD MIGRATION EXECUTOR AT THE WORKER AGENT
K8s uses kubelet as an agent that runs on each node to manage pods. It is responsible for creating, terminating, or updating pods. However, it currently cannot capture and resume the pod state. Hence, to enable these features, a plugin that can checkpoint and restore containers should be installed and integrated into kubelet at every worker node.

In this work, we created this plugin by leveraging the CRIU project and the Container Runtime Interface (CRI) extension in [30]. To support multiple container runtimes, K8s has a defined CRI, which is an interface that any container runtime can implement to be compatible with it. CRI contains two interface definitions as gRPC [31] services. The first is RuntimeService, which is used for managing pod sandboxes and containers. The second is ImageService, which is used for pulling images from the storage. Currently, the container management methods defined in RuntimeService include CreateContainer, StartContainer, StopContainer, RemoveContainer, ListContainer, and ContainerStatus, etc. There is no method for checkpointing and restoring containers. Therefore, we extended the CRI by defining two new methods in the RuntimeService definition:CheckpointContainer and RestoreContainer. The Checkpoint Container method snapshots the container running state, and the RestoreContainer method enables container restoration. With these two new extended methods defined in the CRI as gRPC services, kubelet can now request the container runtime site to checkpoint and restore containers.

To call these two new RuntimeService’s checkpoint and restore methods at the kubelet, we designed a method to handle the pod migration requests sent from the API server. We extended the kubelet’s syncPod function with two new handler functions: Checkpoint handler and Restore handle. With these functions, the kubelet agent at each node can read the requested migration action type (restore, checkpoint, or check pod status) in the pod annotation [32] to ask the remote-CRI to perform the corresponding actions. Noted that by using the pod annotation to provide the migration actions information, there is no need to define any additional API objects or pod specifications, thereby reducing the complexity.

Figure 4 shows components of the migration executor - the extended kubelet at a worker node which performs pod migration tasks that the migration operator from the control plane assigned to the corresponding node. The red dash line
shows the component of kubelet that we extended to perform migration actions: checkpoint and restore. After the pod metadata called annotation is modified by the pod-migration controller, the extended kubelet will watch this information and take the action corresponding to this information. For example, if the pod annotation is set as restore, the kubelet will go to the restore handling phase and call the remote-cri that sends the request via gRPC to the container runtime to restore the pod and save it to the checkpoint shared database (shared DB).

c: CHECKPOINT STORAGE
There are many ways to transfer the checkpoint from the source migration node to the destination migration node. The runtimes which support checkpoint and restore require a path to store the checkpoint. Hence, any file-based transfer method can be used to transmit checkpoints. The most straightforward approach is to store the checkpoint locally, then transfer it to the target node (e.g., using Secure Shell), and then restore it from there. The checkpoint is written and read from the disk twice and transferred once through the network in this approach. Another approach is using a checkpoint folder mounted from the target node to the source node, which reduces the read-and-write operations while still transferring over the network just once. Although this approach is transparent to the container runtime and requires no manual transfer of the checkpoint, this approach requires mounting a shared folder between every pair of nodes in the cluster, which leads to a quadratically growing number of connections. One node must be configured as a file server to perform a migration, and the other needs to mount the shared folder. Therefore, a dedicated file server might be a better solution. Both the source node and the destination node mount the same volume from the file server to exchange the checkpoint. Furthermore, the checkpoint can be shared between every node in the cluster or even between multi clusters. For simplicity, in our implementation, we use a Network File System (NFS) server to store the checkpoint information.

2) STATEFUL MIGRATION WORKFLOW
Figure 5 illustrates pod migration workflow in detail, showing how existing and extended components work together to perform the service migration process. If the application is considered the necessary one to be migrated to another location, the request is sent to the migration converter API to translate it to the K8s API object. Then the K8s API can understand and extract this request to change the related migration CRD. The migration operator watches the migration CRD information to verify which nodes and which actions are requested. Then the requirement which is translated by the migration operator will go through the API server again, and the migration executor at an appropriate node will watch this requirement via the API server to perform the checkpoint or restore process. In the checkpoint process, the worker node agent will checkpoint the container state and finally store this data in a shared DB to transfer to the destination node. In the restore process, the worker node agent as the extended kubelet will first initialize a pod with the same metadata as the source pod. Then, it will wait until the container state is fully created and saved to the shared DB. Finally, it will pull the container state from the shared DB and restore the application pod in this worker node. Additionally, migration between multi-clusters is supported by the migration API converter as a RESTFUL API. For example, if the fault prediction decides that the application needs to be migrated to another cluster, it will simply send a checkpoint request to the source cluster and a restore request to the destination cluster.

Our K8s stateful migration framework is made available on GitHub [33].

IV. IMPLEMENTATION AND EVALUATION
A. EXPERIMENTAL SETUP
Figure 6 depicts the physical topology of our stateful fault-tolerant testbed used in our implementation. We used our expanded K8s version to set up four K8s clusters. For each cluster, we used multiple Intel Xeon 2.4 GHz, 64GB RAM, and 200GB disk physical servers for setting up one master node and at least four worker nodes. The services running inside these nodes were deployed using the K8s StatefulSet feature. Each service has multiple instances deployed as a pod at several nodes in different clusters. The number of different services in each node was randomized, as illustrated in the figure. The default number of pods in each cluster for all experiments is 16. As a K8s StatefulSet pod, each pod has a local Persistent Volume storage which is synchronized.
with the main remote storage of the same service at the NFS servers. These NFS servers were also used for storing container snapshots while migrating. We also used the Intel physical servers to set up these NFS, and the number of NFS instances can be changed based on our experiment scenarios. Each cluster was set up with a separate and dedicated bandwidth connection link to the NFS servers. All the nodes inside each cluster share this bandwidth. An Intel D54250WWYK NUC was used to deploy the High Availability proxy (HA-proxy), which can load-balance user requests to service instances and redirect them to the new pod after pod migration is completed.

For testing applications, we considered two representative applications for each type of stateful service. For booting-state-dependent services, we used MongoDB [34] and Redis [35] database applications. They are typical stateful services
that have a long booting time when restarting if their configuration states are not retained. User requests to these services are in the form of reading or writing data operations through a simple temperature sensor application. For running-state-dependent services, we chose FFMPEG [36] - a video files converter between different formats application, and a Github deep learning training application using the CNN model [37]. When being shut down and migrated, these applications will start from the beginning if their running tasks are not snapshots. Regarding user requests, we used 1280 × 720 resolution, 150MB size MP4 videos as input for FFMPEG, and MNIST handwritten digits dataset as input for the CNN model training application.

For fault simulation, we use the “Stress-ng” tool to generate CPU overload fault for worker nodes based on the Bitbrain dataset. We concatenate the CPU usage data (in terms of percentage) of 1750 nodes into one dataset. Then, as mentioned in the Model Training part, 80% of the data is used for training, and 20% of the data is used for testing. From the testing data, we randomly picked up several 10-minute long patterns to use for experiments. The number of patterns equals the number of nodes from all clusters. Each node’s CPU usage will be controlled by the “Stress-ng” tool based on the chosen pattern. We ran each experiment 100 times, with each time being 10-minute long. The results in the next evaluation part are average values from these runs.

B. EVALUATION SCENARIOS AND METRICS

We conducted two comparison experiments. The first one is the fault prediction performance comparison. We compared 3 kinds of fault prediction models: our Bi-LSTM model, the CNN-LSTM model used in [17], the LSTM model used in [2], and the GRU model used in [3]. The second one is the stateful K8s (default StatefulSet with extended stateful migration feature) migration performance versus the default K8s (only default StatefulSet, migration based on shutting down pods and recreating them). We used the best model in the first experiment to combine with the stateful and default K8s.

For the first comparison, we used Root Mean Square Error (RMSE) and Mean Absolute Error (MAE), Prediction Accuracy, and Fraction of QoS violation metrics. RMSE and MAE functions are shown in (1) - (2), where $y_i$ and $\hat{y}_i$ are the actual value and predicted value, to evaluate the accuracy of the model on the test set.

\[
RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2} \tag{1}
\]

\[
MAE = \frac{1}{n} \sum_{i=1}^{n} \left| y_i - \hat{y}_i \right| \tag{2}
\]

For the second comparison, we compared the stateful and default proactive systems in terms of service recovery time, and the QoS latency avoidance capability in terms of fraction of QoS violation. Service recovery time is the duration from the moment the migration decision is triggered by the model and the moment the service is available at the new node. Fraction of QoS Violation is the ratio of the total time when the service has a higher response time than the accepted QoS to the total running time of the service. We evaluated the fraction of QoS violation over different prediction time step length $t$ (the interval between two times the system makes a prediction). We also evaluated the performance of our stateful K8s system when enlarging the size of the containerized system by increasing the number of pods in each node.

This second experiment is conducted separately for each type of stateful service: booting-state-dependent and running-state-dependent due to the different characteristics of these two types of services, which will be explained further in the next part.

C. RESULTS

For the fault prediction model comparison, we took the average values over 100 runs. Figure 7 shows the evaluation score comparison between our chosen Bi-LSTM model and other baseline models. On average, the RMSE score and the MAE score of the Bi-LSTM model are the lowest. This is due to the ability to learn data dependencies in both backward and forward directions of the Bi-LSTM model.

For stateful and default proactive systems comparison, we used our Bi-LSTM model for the fault prediction framework as it had the best performance results. We analyzed the comparison results with two different service types mentioned above.

It must be noted that only booting states and running states of these corresponding types of service are transferred in our stateful service recovery migration process. The storage states, which are the states of the data that is stored in each pod local Persistent Volume, do not need to be transferred. These Persistent Volumes at all nodes are never deleted, and they always synchronize with the main remote storage of each service at NFS servers as explained in our Experiment Setup part. We set up like this for experiment simplicity. In production, framework such as Longhorn [38] can be used for quick storage recovery between clusters. When a pod is migrated to the new node, it can simply re-attach to its corresponding local Persistent Volume thanks to K8s StatefulSet functionality. In case of MongoDB and Redis service, this storage state is the current data in these services’ databases. In case of FFMPEG and CNN Training model service, there...
is no storage state since these services does not need to store any data.

1) BOOTING-STATE-DEPENDENT SERVICES

The notable characteristics of the Booting-state-dependent services are their long booting/startup time if being restarted using default K8s or the significant snapshot transferring time if being checkpointed and migrated using our stateful K8s. In particular, considering MongoDB and Redis in our experiment, their in-memory caches, master-slave configurations, and resource pool initialization cause this long booting time and significant checkpoint snapshot size. They are the key factors that affect the service recovery time and QoS violation avoidance capability of the stateful and default K8s systems.

First, we compare the service recovery time between the two systems. In the case of the default K8s system, the application is restarted from scratch at the new node. Hence, the service recovery time is the application booting time. Meanwhile, in the case of the stateful K8s system, the service recovery time is the sum of 3 steps: checkpointing state, transferring state, and restoring from the previous state. The state here is the in-memory booting state which consists of caches, configuration, etc. Different bandwidth types between the NFS server and clusters can cause different transferring times. Therefore, we analyzed the impact of bandwidth in Figure 8. The chosen testing bandwidth range (10-100Mbps) is based on a recent study of fog computing networks [39]. The results show that the stateful system has shorter service recovery times than the default system with most bandwidth types (>25 Mbps). At the highest bandwidth (100Mbps), the service recovery time can be reduced by approximately 50%. Considering network bandwidth will continue increasing in current and upcoming network generation (5G, 6G), the stateful approach is clearly the better solution for now and in the future.

Second, we evaluated the QoS avoidance capability of the two systems. We exponentially increased the prediction time step length t to observe its impact. The results are shown in Figure 9. There are two causes of QoS violation: incomplete migration before the overloading moment and model failure to predict node overloading in the next time step. For the first case, because the service is migrated at the start of each time step t, considering the service recovery time is m seconds, if overloading happens at the first m seconds of t, the migration process is not yet completed. The latency QoS violation will happen from the overloaded moment until the service is available at the new node. The faster the service recovery time is, the less QoS latency violation the system suffers. Hence, the stateful system with a shorter recovery time has less violation rate than the default one in every time step. However, when the prediction time step t becomes too long, the model accuracy is significantly decreased, as shown in Figure 8(a) (below 90% for 2 minutes time step and only 80% for 4 minutes time step). This happens because the model predicts the average resource usage value in the next step. Then, if the step is too long, the predicted average value is likely less than the overloaded threshold. For example: in a 4-minute time interval, overloading happens 3 times and the total overloading duration is only 20 seconds, then the average predicted value is likely to be less than the threshold. Therefore, no migration is triggered, and because both stateful and default systems use the same prediction model, they will mostly suffer the same QoS violation at overloading durations. This explains why when the time step increases, less of the first and more of the second violation type happens. Hence, both systems’ violation rates increase and slowly converge, as can be seen at the 4-minutes time step in the figure. The exceptional decrease in the violation rate of the default system between 30-seconds and 1-minute time steps caused by its migration time m (average 23s of these two applications) is too close to the time step t (30-seconds), which leads to the major violation rate type being the incomplete migration. With reasonable time steps (1 or 2 minutes in our experiment), the QoS violation percentage of the stateful system was 2 to 3% lower than the default one.

Third, we evaluated whether the stateful k8s proactive system can keep the performance when the containerized environment becomes larger. Considering all nodes in the same cluster share the same dedicated bandwidth link to the NFS servers, we raised this concern because when the number of pods inside each node or the number of nodes inside each cluster increases, more snapshots will be transferred in the same link. It will cause slower transferring speed and, thus, longer transferring time as well as longer service recovery time. As we analyzed in previous figures, a longer service recovery time will decrease the QoS avoidance capability. Therefore, we propose the solution to this issue by increasing the number of NFS instances and load-balancing the checkpoint snapshots between them from the migration operator at each cluster. Each NFS server and cluster connection has a separate bandwidth link. We evaluated this solution’s efficiency by sequentially increasing the number of pods in each node, then increasing the number of NFS servers. The experiment is conducted using MongoDB application with the best configuration found in the previous evaluation, which is a 1-minute prediction time step length and 100Mbps bandwidth. The result is shown in Figure 10 and Figure 11.
The results in these figures confirmed the effect of a large containerized environment on the stateful K8s proactive system as the service recovery time as well as the fraction of QoS violation gradually increased when the number of pods increased from 16 to 128 pods. The performance of the stateful system can become even worse than the default K8s system if the number of pods continues to increase. However, by increasing the number of NFS servers from 1 to 3 and giving separated bandwidth links from clusters to each of these NFS servers, the bottleneck issue can be solved. The service recovery time and QoS avoidance capability decreased as more NFS servers were available. If each node has a separate connection link to the NFS servers instead of each cluster, the performance will be even better. Therefore, we concluded that the stateful K8s proactive system can retain the performance in a large containerized system with an appropriate number of NFS servers and a dedicated bandwidth link setup.

There is one more notable observation that can be seen in Figure 10. The checkpoint time increase is just 1-2 seconds and is not significant as we increased the number of pods in each node from 4 to 32 (equals 16 to 128 pods in each cluster). This result showed that our stateful K8s proactive system does not create significant resource overhead when there is more concurrent pod checkpointing processes happen. The reason is that our system integrates the checkpoint process into K8s, which is better than the solution MyceDrive proposed in [13]. MyceDrive solution requires an additional DMTCP sidecar container running in each pod and an extra Execution Agent running inside each service container. Hence, the more pods/containers in the system, the more resource overhead this solution might have. We cannot show MyceDrive performance here because of its code unavailability. Meanwhile, our CRIU-based migration execution agent is integrated inside the K8s kubelet at each node. Therefore, it does not create significant extra overhead when increasing the number of pods.

2) RUNNING-STATE-DEPENDENT SERVICES
In contrast to booting-state-dependent services, running-state-dependent services such as FFmpeg and CNN model training services in our experiment have quick booting time but much longer service completion time than read-write data operations of databases. Because of their short booting time in case of using default K8s system and short restore time and transferring time due to small application size in case of using stateful K8s system, there is no significant difference in service recovery time between the two systems for this kind of service. However, the stateful K8s system matters.
when considering service completion time when migration happens. By retaining the state before migration, migrated services at the new node start from the moment they are interrupted. Meanwhile, with the default K8s system, the service will be restarted from scratch. Therefore, with running-state-dependent service, we make evaluations based on service completion time instead of service recovery time. The results are shown in Figure 12. All experiments for running-state-dependent services used 100Mbps bandwidth and 1-minute prediction time step length based on the result of previous results evaluation on system configuration for our testbed.

As the state is retained, the service completion time for FFmpeg and CNN model training services when using the stateful K8s proactive system is the sum of its normal service execution time (previous executed time at the old node and the rest at the new node) and the service migration time (around 4-5 seconds for checkpointing, transferring, and restoring). It was about 12 seconds for the FFmpeg service and 16 seconds for the CNN model training service. Meanwhile, when using the default K8s system, the service completion time is the sum of the previous executed time at the old node, migration time, and the whole service execution time from the beginning at the new node. That made the service completion time much longer when using the default system: about 20 seconds (66% higher) for FFmpeg and 25 seconds (56% higher) for CNN model training. The range of the time is also longer because it depends on when the service is interrupted, which is random.

The difference in service completion time between the two systems affected their QoS avoidance capability, as shown in Figure 13. The stateful system reduced the QoS violation rate by 20 to 22%. This happened because with such a long service completion time, the migrated service using the default system always violated QoS latency violation even when the fault prediction model correctly forecasted the overloading fault. Every time migration happened, the default system’s services violated QoS. Meanwhile, when using the stateful system, it only violated QoS when the fault prediction made a wrong call or the migration time was high due to the bandwidth bottleneck between the cluster and the NFS server. We already mentioned the solution for this issue in the previous part when evaluating the stateful system performance in a large containerized environment.

Based on the above results, we conclude that, with a fine-tuned time step length, the proactive stateful system has a better QoS violation avoidance rate and lower migration cost than the default one for different kinds of stateful services.

**V. CONCLUSION**

This paper presents the architecture of a proactive stateful failure recovery system for containerized services with K8s as the containerization platform in multi-cluster scenarios. It integrates two stages: a failure fault prediction using Bi-LSTM model and a novel stateful service migration scheme for K8s services. Our system allows K8s to proactively retain booting and running services to recover services from their interrupted moment caused by cloud infrastructure failure. Our experimental results showed the proposed system efficiency against baseline methods over different kinds of state-dependent services. For booting-state-dependent services, the service recovery time is reduced by 50% and the QoS violation percentage is reduced by 2-3%. For running-state-dependent services, the service recovery time is reduced by 40-60% and the QoS violation percentage is reduced by 20-22%.

For future works, we plan to integrate this approach in specific telecommunication use-cases, in which maintaining user sessions’ states is a vital requirement.

**REFERENCES**

[1] Z. Huang and H. Huang, “Proactive failure recovery for stateful NFV,” in *Proc. IEEE 26th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Hong Kong, Dec. 2020, pp. 536–543.

[2] S. Jeong, N. Van Tu, J.-H. Yoo, and J. W.-K. Hong, “Proactive live migration for virtual network functions using machine learning,” in *Proc. 17th Int. Conf. Netw. Service Manage. (CNSM)*, Izmit, Turkey, Oct. 2021, pp. 335–339.

[3] S. Tuli, G. Casale, and N. R. Jennings, “PreGAN: Preemptive migration prediction network for proactive fault-tolerant edge computing,” in *Proc. IEEE INFOCOM Conf. Comput. Commun.,* May 2022, pp. 670–679.

[4] D. Goncalves, K. Velasquez, M. Curado, L. Bittencourt, and E. Madeira, “Proactive virtual machine migration in fog environments,” in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Natal, Brazil, Jun. 2018, pp. 742–745.

[5] V. M. Sivagami and K. S. Easwarakumar, “An improved dynamic fault tolerant management algorithm during VM migration in cloud data center,” *Future Gener. Comput. Syst.*, vol. 98, pp. 35–43, Sep. 2019.

[6] A. Sharif, M. Nickray, and A. Shahidinnejad, “Fault-tolerant with load balancing scheduling in a Fog-based IoT application,” *IET Commun.*, vol. 14, no. 16, pp. 2646–2657, Oct. 2020.
[7] S. Negi, M. M. S. Rauthan, K. S. Vaïsla, and N. Panwar, “CMODLB: An efficient load balancing approach in cloud computing environment,” J. Supercomput., vol. 77, no. 8, pp. 8787–8839, Jan. 2021.

[8] M. C. Araujo, B. Sousa, M. Curado, and L. F. Bittencourt, “CMFog: Proactive content migration using Markov chain and MADM in fog computing,” in Proc. IEEE/ACM 15th Int. Conf. Utility Cloud Comput. (UCC), Leicester, U.K., Dec. 2020, pp. 112–121.

[9] L. Wang, W. Mao, J. Zhao, and Y. Xu, “DDQP: A double deep Q-learning approach to online fault-tolerant SFC placement,” IEEE Trans. Netw. Service Manage., vol. 18, no. 1, pp. 118–132, Jan. 2021.

[10] V. Medel, O. Rana, J. A. Bahares, and U. Arronategui, “Modelling performance & resource management in Kubernetes,” in Proc. 9th Int. Conf. Utility Cloud Comput., Shanghai, China, Dec. 2016, pp. 257–262.

[11] D. T. Nguyen, N. L. Dao, V. T. Tran, K. T. Lang, T. T. Pham, P. H. Nguyen, C. D. Pham, T. A. Pham, D. H. Nguyen, and H. T. Nguyen, “Enhancing CNF performance for 5G core network using SR-IOV in Kubernetes,” in Proc. 24th Int. Conf. Adv. Commun. Technol. (ICACT), PyeongChang, South Korea, Feb. 2022, pp. 501–506.

[12] P. S. Junior, D. Miorandi, and G. Pierre, “Stateful container migration in geo-distributed environments,” in Proc. IEEE Int. Conf. Cloud Comput. Technol. Sci. (CloudCom), Bangkok, Thailand, Dec. 2020, pp. 49–56.

[13] P. S. Junior, D. Miorandi, and G. Pierre, “Good shepherds care for their cat: Seamless pod migration in geo-distributed Kubernetes,” in Proc. IEEE 6th Int. Conf. Fog Edge Comput. (ICFEC), Taormina, Italy, May 2022, pp. 1–10.

[14] B. K. Ray, A. Saha, S. Khatua, and S. Roy, “Proactive fault-tolerance technique to enhance reliability of cloud service in cloud federation environment,” IEEE Trans. Cloud Comput., vol. 10, no. 2, pp. 957–971, Jan. 2022.

[15] 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 Trans. Cloud Comput., vol. 6, no. 4, pp. 1191–1202, May 2016.

[16] H. Huang and S. Guo, “Proactive failure recovery for NFV in distributed edge computing,” IEEE Commun. Mag., vol. 57, no. 5, pp. 131–137, May 2019.

[17] M. Souriha, C. Fu, and F. Khom, “Infrastructure fault detection and prediction in edge cloud environments,” in Proc. 4th ACM/IEEE Symp. Edge Comput., Washington, DC, USA, Nov. 2019, pp. 222–235.

[18] A. Machen, S. Wang, K. K. Leung, B. J. Ko, and T. Salonidis, “Live service migration in mobile edge clouds,” IEEE Wireless Commun., vol. 25, no. 1, pp. 140–147, Aug. 2017.

[19] Checkpoint/Restore in Userspace. Accessed: May 9, 2022. [Online]. Available: https://criu.org/Main_Page

[20] Ptrace (2)—Linux Manual Page. Accessed: Apr. 20, 2021. [Online]. Available: https://man7.org/linux/man-pages/man2/pttrace.2.html

[21] O. I. Abdullaziz, L.-C. Wang, S. B. Chundrigar, and K.-L. Huang, “Enabling mobile service continuity across orchestrated edge networks,” IEEE Trans. Netw. Sci. Eng., vol. 7, no. 3, pp. 1774–1787, Jul. 2019.

[22] Z. Benomar, F. Longo, G. Merlino, and A. Puliafito, “Cloud-based enabling mechanisms for container deployment and migration at the network edge,” ACM Trans. Internet Technol., vol. 20, no. 3, pp. 1–28, Jun. 2020.

[23] T. Benjaipongtak, M. Karakate, and K. Siripandikulchai, “Enabling live migration of containerized applications across clouds,” in Proc. IEEE INFOCOM Conf. Comput. Commun., Toronto, ON, Canada, Jul. 2020, pp. 2529–2538.

[24] B. Xu, S. Wu, J. Xiao, H. Jin, Y. Zhang, G. Shi, T. Lin, J. Rao, L. Yi, and J. Jiang, “Sledge: Towards efficient live migration of Docker containers,” in Proc. IEEE 15th Int. Conf. Cloud Comput. (CLOUD), Beijing, China, Oct. 2020, pp. 321–328.

[25] K. Govindaraj and A. Artemenko, “Container live migration for latency critical industrial applications on edge computing,” in Proc. IEEE 23rd Int. Conf. Emerg. Technol. Factory Autom. (ETFA), Turin, Italy, Sep. 2018, pp. 83–90.

[26] L. Ma, S. Yi, N. Carter, and Q. Li, “Efficient live migration of edge services leveraging container layered storage,” IEEE Trans. Mobile Comput., vol. 18, no. 9, pp. 2020–2033, Sep. 2019.

[27] J. Ankel, K. Arya, and G. Cooperman, “DMTCP: Transparent checkpointing for cluster computations and the desktop,” in Proc. IEEE Int. Symp. Parallel Distrib. Process., Rome, Italy, May 2009, pp. 1–12.

[28] S. Shen, V. Van Beek, and A. Iosup, “Statistical characterization of business-critical workloads hosted in cloud datacenters,” in Proc. 15th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput., Shenzhen, China, May 2015, pp. 465–474.

MINH-NGOC TRAN received the B.E. degree from the Hanoi University of Science and Technology, in 2018, and the M.S. degree from Soongsil University, in 2020, where he is currently pursuing the Ph.D. degree in information and communication convergence. His research interests include cloud computing, applied machine learning, and 5G networking.

XUAN TUONG YU received the B.E. degree in information and communications technology, Vietnam, in 2018. He is currently pursuing the master’s degree in information and Communication Convergence with Soongsil University. His current research interests include cloud networking and multi-access edge computing (MEC).

YOUNGHAHN KIM (Member, IEEE) received the B.S. degree from Seoul National University and the M.Sc. and Ph.D. degrees in electrical engineering from KAIST. He was the President of the Korea Information and Communications Society (KICS). He is currently a Full Professor with the Department of Electronic Engineering, Soongsil University. His current research interests include cloud computing, 5G networking, and next-generation networks.

* * *