Computer Communications 236 (2025) 108111

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

Resiliency focused proactive lifecycle management for stateful microservices in multi-cluster containerized environments

Abd Elghani Meliani *, Mohamed Mekki, Adlen Ksentini

Eurecom, Biot, 06410, Nice, France

ARTICLE INFO

Keywords:
Zero-touch management
Stateful migration
Kubernetes
Application lifecycle
Cloud-native
Microservices

ABSTRACT

Containerization has become fundamental to deploying cloud-native applications, allowing for the packaging and independent execution of applications. This approach speeds up deployment processes and facilitates the creation of various environments for feature testing. However, the ephemeral nature of containers poses a significant challenge to data persistence, especially during container restarts or migrations across different hosts. This paper proposes a proactive zero-touch management solution for stateful microservices applications, ensuring seamless application lifecycle management. Our solution integrates seamlessly with container platforms such as Kubernetes and supports multi-cluster environments, enhancing fault tolerance and data persistence in stateful applications. The solution has been thoroughly tested on different hardware configurations in the public cloud and with our on-premises servers.

1. Introduction

Containerization [1] has recently emerged as the default solution for deploying most modern industrial applications, particularly with the release of Kubernetes, one of today's leading container orchestrators. This technology enables seamless deployment and has driven the adoption of microservices architecture [2] in real-world software solutions. These applications often require migration, known as service migration [3], from one computing node to another. The migration can be driven by various factors, including (i) user mobility, to reduce latency by connecting to the closest computing node [4] ; (ii) minimizing execution costs on specific cloud provider platforms or conserving energy through server consolidation [3]; and (iii) enhancing host fault tolerance and system resiliency by preemptively migrating applications in anticipation of node failures or malfunctions.

Container platforms like Kubernetes [5] provide basic re-scheduling mechanisms, which involve suspending an application on one node and redeploying it on another. However, these mechanisms are limited to stateless services, leaving a critical gap when dealing with stateful applications. Applications such as machine learning models, in-memory databases, and video streaming platforms maintain significant runtime data directly at the execution point, representing their working state. The stateless nature of Kubernetes migrations results in challenges such as service degradation, data loss or inconsistency, and extended downtime, which are unacceptable for latency-sensitive and state-critical applications. These challenges have led to the emergence of the concept of stateful container migration, which involves moving a running container from one machine to another while preserving its working memory state. Stateful migration, a seemingly simple task, poses numerous challenges, such as the need to transfer data from the source machine to the destination machine, which can be time-consuming depending on the data volume. Moreover, the migration process should be transparent to the end user, allowing continuous use of the container during the migration period. Finally, implementing a ready-to-use solution to address this issue must ensure compatibility with container orchestrators like Kubernetes, which is a complex task due to the sophisticated architecture and implementation of its components. Kubernetes, in particular, lacks a native mechanism to enable seamless stateful migrations that are compatible with its sophisticated architecture. Generally, existing approaches to stateful container migration often involve specialized tools or significant manual configuration, and they are typically designed for single-cluster scenarios, making them unsuitable for modern architectures such as the cloud–edge continuum environments [6]. Also, many of these approaches rely on modifying the container runtimes or altering Kubernetes' orchestration behavior, which disrupts the default functioning of container orchestrators and introduces complexity in managing and maintaining these changes. Additionally, most stateful migration solutions in the literature are reactive in nature, addressing failures only after they occur, rather than proactively preventing performance degradation or resource exhaustion. Furthermore, these solutions often overlook the unique requirements of microservices applications, focusing instead on monolithic applications, which fail to

* Corresponding author.
E-mail address: abdelghanimeliani7@gmail.com (A.E. Meliani).

https://doi.org/10.1016/j.comcom.2025.108111
Received 29 August 2024; Received in revised form 16 January 2025; Accepted 22 February 2025
Available online 8 March 2025
0140-3664/© 2025 Elsevier B.V. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

address the needs of modern, distributed architectures. This limitation underscores the novelty and necessity of our work, which introduces a unified, proactive framework to manage stateful container migrations in both intra- and inter-cluster contexts. To address the previous challenges and the limitations of existing works, in this paper, we present a proactive zero-touch management (ZTM) solution designed to enhance the lifecycle management (LCM) of stateful applications across multiple containerized environments. The proposed solution aims to preemptively address the performance degradation of deployed applications caused by increased consumption of computing resources (CPU and memory) on computing nodes. This is achieved by predicting potential node resource insufficiency, hence, enhancing the system's resiliency against sudden rises in resource usage at the computing nodes. The proposed solution aims to preemptively prevent performance degradation in deployed applications by predicting potential shortages in computing resources (CPU and memory) on computing nodes. By anticipating resource insufficiencies, the solution enhances the system's resilience against unexpected spikes in resource usage at a node.

To this end, we assume a federated infrastructure managed by a central resource orchestrator (RO) component, which acts as a first-tier scheduler to select the most available cluster for running the application. This RO operates on top of local management systems (LMS) such as K8s and K3S, which are managed by a cluster manager acting as a second-tier scheduler to choose the most available worker node within the selected cluster. The cluster manager serves as the intermediary between the RO and the LMS. The cluster manager triggers the migration decision when the anticipated output from the system's predictor component exceeds a certain defined Migration Threshold. Once the decision to migrate is made, we will distinguish between the two scenarios. The first one corresponds to the case that the migration is done between two nodes of the same cluster. Here, the local cluster manager runs the migration procedure we propose without notifying the RO. The second scenario is if there are not enough resources at the local cluster, hence a need to migrate to another cluster. In this case, the RO enters into play by selecting another cluster and running the migration procedure. Migrating containers, or processes in general, necessitates saving the process state and then restarting the process from this state at the target location. To do so, we designed a Kubernetes operator that leverages the recent integration of the Checkpoint 2.4 feature in Kubernetes. Our operator strategically creates checkpoints for stateful containers (This checkpoint captures the entire state of the container, including its working memory, process state, open files, and other critical system resources), constructs a new image based on these checkpoints, and then uploads this newly built image to an image registry. The image is used to redeploy the container on another node as it encapsulates the state present in the original container.

This paper proposes a novel solution to manage the lifecycle of stateful applications. Our main contributions are summarized as follows:

- **The design and implementation of a comprehensive stateful application lifecycle Management Architecture:** The architecture is designed to manage multi-cluster applications and simplify the process of stateful migration, allowing seamless application migration between clusters or from one node to another within the same cluster. The architecture encompasses several key components, such as the resource orchestration module tasked with selecting the most suitable cluster to run the application and the Cluster Manager, which leverages monitoring data provided by the resource exposer to choose the most available node for the application. The Cluster Manager also triggers migration decisions based on the system predictor's output. Finally, the resource Exposer will be used by different entities to manage cluster resources, along with the implementation of a resource discovery module that utilizes the exposers deployed in various clusters to provide the RO with a global infrastructure view.
- **The design and implementation of a resiliency-focused time series forecasting system:** This system is designed to enhance the resiliency of service management by predicting potential node failures and resource bottlenecks. It features a predictor component that includes two specialized time series forecasting models: one for CPU usage and another for memory consumption. These models analyze historical data from real-world datasets to forecast future resource needs and detect early signs of potential issues. By integrating these forecasts into the decision-making process for service migration, the system proactively manages resources to prevent disruptions and ensure continuous service availability, aligning with the Zero-touch Service Management (ZSM) vision supported by ETSI [7].
- **The design and implementation of a stateful Migration Checkpointing System:** This system is designed to facilitate seamless stateful migration by reliably saving the application's state before migration processes commence. It ensures that all critical application data is preserved, allowing for smooth transitions between clusters or nodes. The checkpointer component captures the state of the application at key points, thereby enabling accurate restoration and continuity of operations during migration and minimizing the risk of data loss or inconsistency. The checkpointer is designed to work seamlessly with existing low-level tools without altering Kubernetes' default behavior, ensuring compatibility and ease of adoption. We provide the source code [8] for this software, along with a demo video [9] that demonstrates how to run it, encouraging further contributions to future research.

The remainder of this paper is organized as follows: Section 2 provides background information and introduces the technological tools employed in our solution. Section 3 reviews related work, while Section 4 presents the details of our proposed approach. Section 5 discusses the results obtained and Section 6 highlights the advantages and limitations of the solution. Finally, Section 7 concludes the paper.

2. Background

2.1. Kubernetes

Kubernetes, also known as K8s, is an open-source system for automating the deployment, scaling, and management of containerized applications [5]. It has become the most widely used container orchestration tool in the industry, offering a wide range of features that help developers and system administrators deploy their applications smoothly. Kubernetes is built on a complex architecture that integrates several components to function seamlessly. Several container orchestration tools have been built either on top of Kubernetes or inspired by it. For instance, Red Hat OpenShift [10] and OKD [11], its community distribution, are based on Kubernetes, extending its functionality and offering additional features. On the other hand, tools like KubeEdge [12] and K3s [13] are inspired by Kubernetes and aim to be more lightweight and suitable for edge computing scenarios. K3s, in particular, is designed to provide a streamlined version of Kubernetes that is easier to deploy and manage at the edge of the network.

2.2. Kubernetes scheduler

Kubernetes manages a set of nodes that fall into two categories: master nodes and worker nodes. Generally, master nodes host the Kubernetes control plane, which is responsible for managing the overall cluster state and orchestrating the scheduling of workloads. In contrast, worker nodes are utilized to run the user applications and execute the containers. While Kubernetes is a container orchestration platform, the smallest deployable unit it manages is called a pod. The latter serves

as an abstraction layer over containers, encapsulating one or more containers that share the same network namespace and storage resources. This abstraction helps manage the containers as a single unit, providing a consistent deployment and management experience. To choose the node on which to run a certain pod, Kubernetes leverages one of its control plane components, the scheduler. The scheduler automatically selects nodes for the pods at deployment time, ensuring efficient and balanced resource utilization across the cluster. However, Kubernetes also allows users to influence pod scheduling on specific nodes or sets of nodes using attributes like *nodeSelector*, *nodeAffinity*, and *nodeName*. These attributes provide flexibility for users to direct pods to particular nodes based on various criteria. This flexibility enables users to build their own scheduling mechanisms on top of the Kubernetes scheduler and enforce their custom scheduling decisions. Additionally, Kubernetes provides the ability to use alternative schedulers instead of the default one, allowing for even more tailored scheduling solutions to meet specific needs.

2.3. Kubelet

To run containers, Kubernetes executes an agent named kubelet on every node. Its primary responsibility lies in effectively managing containers residing on these nodes. Kubelet engages with container runtimes, network plugins, and storage management components through Google Remote Procedure Call (gRPC)[1] interfaces, specifically the Container Runtime Interface (CRI)[2], Container Network Interface (CNI)[3], and Container Storage Interface (CSI)[4]. For communication with container runtimes, Kubelet utilizes the CRI interface to execute request calls. This flexibility allows vendors to develop their own solutions based on their specific needs. Several options for container runtimes exist to work with Kubernetes, including Docker [14], Podman [15], containerd [16], and CRI-O [17].

2.4. Checkpoint and restore

Starting from version 1.25.0, Kubernetes introduced the container checkpoint feature as an alpha feature at the kubelet level [18]. This means that kubelet now has a gRPC call to establish a checkpoint for running containers. However, exposing this feature only at the kubelet level poses challenges for system administrators seeking to perform checkpoints on pods running in the cluster. They would need access to the specific node's kubelet instance and the necessary credentials to interact with kubelet. Additionally, Kubernetes does not currently provide a solution to restore checkpointed containers. This limitation arises from the lack of an Open Container Initiative (OCI)[5] definition for checkpointed images. Consequently, containers checkpointed with a specific container runtime must be restored using the same container runtime that was used during the checkpoint. It is important to note that, as of now, only the CRI-O container runtime has implemented the checkpoint and restore feature by leveraging the CRIU module [19]. To initiate the container's restore process, users must build images from scratch, incorporating the checkpoint file into the image. Additionally, certain annotations must be included to inform CRI-O that the image is a checkpointed image rather than a typical OCI-compatible one. Once these steps are completed, CRI-O can proceed with the restoration process.

[1] A high performance, open source universal RPC framework.

[2] A plugin interface which enables the Kubelet to use a wide variety of container runtimes, without having a need to recompile the cluster components.

[3] a specification and libraries for writing plugins to configure network interfaces in Linux containers.

[4] is an initiative to unify the storage interface of Container Orchestrator Systems.

[5] is an open governance structure for the express purpose of creating open industry standards around container formats and runtimes.

2.5. Kubernetes operator pattern

Kubernetes provides a way to extend its API [20] via custom resources (CR) and custom controllers. A custom resource is an extension of the Kubernetes endpoint list (API) that is not necessarily available in a default Kubernetes installation. It represents a customization of a particular Kubernetes resource. Many core Kubernetes functions are now built using custom resources, making Kubernetes more modular. Custom resources, on their own, allow you to store and retrieve structured data. When combined with a custom controller, custom resources provide a true declarative API. The Operator pattern combines custom resources and custom controllers to encode domain knowledge for specific applications into an extension of the Kubernetes API [21]. Developers often use the Operator pattern as it is the simplest way to add custom features to Kubernetes. In our work, we use the Kubernetes Operator pattern to expose the checkpoint feature at a higher level as a native Kubernetes API. This allows developers to build tools on top of it, such as the cluster manager in our system. Our checkpoint operator can also function as a standalone component, enabling system administrators to checkpoint, build, and push stateful container images to public/private image registries so that it can be used for various purposes.

2.6. Buildah

When discussing the process of building images, an instrumental tool is Buildah [22], designed to streamline the creation of Open Container Initiative (OCI) images. In our solution, the checkpoint operator uses the Buildah Golang package to construct images directly from the checkpoint. This approach is more efficient and time-saving compared to the alternative of installing the Buildah command-line tool and running it inside the container that hosts the checkpointer controller. Utilizing the Buildah Golang package enhances the lightweight nature of the image-building process, contributing to a more expedient operation.

3. Related work

When it comes to application migration, several works exist, as well as research on stateful microservices management. However, to the best of our knowledge, we are the first to address both of these domains simultaneously.

In [23], the authors employed reinforcement learning to devise a microservices migration policy, taking into account factors such as user mobility and latency. The researchers utilized a Markov Decision Process (MDP) [24] in conjunction with the Dyna-Q algorithm to achieve optimal policy learning. The primary objective of this project was to strategically place and migrate the microservices that constitute a specific application based on user mobility patterns. Despite the innovative elements introduced in this study, including its focus on the use case of a microservices application, a notable limitation is the absence of tangible implementations of placement operations, particularly the migration of these microservices. The authors concentrated solely on formulating the policy for placing or relocating microservices, leaving the practical implementation of these operations unaddressed.

In [25], the authors propose using the OverlayFS[6] file system's layered structure for container relocation in Kubernetes. This involves discreetly taking snapshots of storage content and moving them before the actual container relocation. The method utilizes a pre-copy plan [26] to transfer storage data from the old to the new location,

[6] OverlayFS operates by merging two filesystems — the upper (read-write) layer and the lower (read-only) layer. This unique approach allows us to combine the contents of these layers into a single unified view while keeping their original attributes intact.

focusing primarily on hard disk information. It consists of two main steps: Volume Snapshot (capturing the initial state and transmitting new files) and Migration Coordination (using an ephemeral container and coordinator with full access rights). This initiative aims to relocate disk state, rather than the process state typically addressed in production scenarios where cloud providers handle data movement between data centers. Therefore, in our case, this is not a major concern.

In [27], the same authors introduced MyceDrive, a seamlessly integrated container migration solution within Kubernetes. They demonstrated the feasibility of migrating geographically distributed Kubernetes pods while preserving the complete application state and user data. The process involves stopping the source pod, checking memory and system resources, transferring data, and restarting a new pod from the checkpoint. MyceDrive comprises two key elements: an Execution Agent integrated into each container, managing its lifecycle, and a Migration Coordinator coordinating pod migration through interactions with the Kubernetes API. MyceDrive uniquely focuses on transferring stateful containers within the same Kubernetes cluster, irrespective of persistent volumes. Unlike other initiatives using the checkpoint concept, the authors preferred DMTCP [28] over CRIU, eliminating the need for a kernel module. However, to function, this solution requires a specific execution agent instance inside each container. This forces developers to build new containers for their applications that include it. Additionally, a DMTCP container instance in each pod can potentially decrease the system performance for applications that require a considerable number of pods to be deployed.

In [29], the authors introduce a novel approach to improving fault tolerance inside K8s environments. They propose a system that combines a Bidirectional Long Short-Term Memory (Bi-LSTM) fault prediction framework with a distinctive stateful service migration mechanism. However, their model primarily addresses CPU overload, whereas our solution also considers memory utilization. Furthermore, our approach manages the entire application lifecycle and is compatible with various Kubernetes distributions while giving particular focus to microservices applications. Another key difference is that our solution is compatible with a large set of container orchestrators, which makes it suitable for most cloud–edge environments.

The work in [30] presents a solution closely related to ours, offering fault tolerance for stateful containerized applications in a transparent manner, meaning the application does not need to structure or manage its state in any specific way. The solution uses a Kubernetes operator to periodically checkpoint containers and restore them from the latest checkpoints in case of a node failure. However, compared to our approach, this solution has several shortcomings. First, it is reactive to node states, whereas ours is proactive we will show the importance of proactivity will be highlighted in the results and the discussion & limitations sections. Second, the periodic checkpointing approach not only burdens the node storage with unused checkpoints but also risks state loss if the Kubernetes node becomes “Not Ready” between checkpoint intervals while the container processes multiple requests. Lastly, our solution specifically addresses the unique requirements of microservices applications, whereas [30] is limited to monolithic applications.

In [31], the authors focus on improving the availability of stateful microservices by developing a Kubernetes controller to ensure high availability (HA) for stateful applications running on Kubernetes. They identified a key limitation in Kubernetes’ default architecture, which is not well-suited for HA stateful pods. To address this, they proposed a solution that replicates the state managed by the StatefulSet controller to standby pods. These standby pods remain in a dormant state, ready to take over in case of a failure in the active pods. A custom controller was implemented to handle state assignment and replication, enhancing scalability and making the approach practical for real-world applications without requiring modifications to Kubernetes’ source code. While this solution shares a similar general goal with ours—enhancing the lifecycle of stateful microservices—it operates in a different context. Our approach focuses on improving resiliency in the infrastructure hosting the containerized setup, particularly by enabling the migration of stateful microservices across dynamically managed nodes in both single- and multi-cluster environments. Although duplicating container states can improve pod recovery time, it does not address challenges related to node failures, which is the core issue our solution aims to resolve.

The work presented in [32] focuses on improving the management of stateful applications by introducing the Persistent Volume Autoscaler. This tool automatically manages volume resources in Kubernetes, adjusting various components needed for stateful applications to effectively utilize these volumes. The proposed solution extends Kubernetes’ capabilities by enabling autoscaling for storage volumes, supporting both scale-up and scale-down operations—a feature not natively supported in Kubernetes, which primarily handles autoscaling at the pod level using metrics like CPU and memory utilization. This innovative approach enhances the availability and reliability of stateful applications. However, it is important to note that the problems addressed in this work are distinct from those tackled in our study.

In [33], the authors propose a novel Kubernetes-based approach to enable and optimize the migration of specific machine learning workloads in Industry 4.0 scenarios. Their solution involves modifying core Kubernetes components to support the migration process. The key innovation lies in allowing certain operations in the migration lifecycle to run in parallel—specifically, by transferring actively used data first, starting the application, and subsequently transferring the remaining volume data. This method significantly reduces application downtime during migration. However, their approach has notable limitations compared to our solution. It is tailored to a specific range of applications, focusing on machine learning workloads for anomaly detection in specialized environments. Furthermore, while their work emphasizes the migration of Kubernetes volume state (disk-hosted data), our solution targets the migration of internal process state stored primarily in main memory. As a result, the two approaches operate within distinct scopes, addressing different aspects of the migration challenge.

The work in [34] is one of the few that focuses on microservice placement and migration, whereas most other studies primarily address the migration of monolithic applications. The authors model the microservice migration problem as a Markov Decision Process and propose an intelligent migration algorithm based on reinforcement learning, taking into account several variables such as edge server bandwidth, user mobility, and application requirements. While this is an interesting and innovative approach, it is specifically designed for stateless microservices and does not handle the persistence of the application state.

As outlined in this section, we focused on presenting recent related works. These works primarily addressed one of three areas: microservices management, stateful application challenges in Kubernetes, or general application migration strategies. A summary of the discussed works is provided in Table 1.

4. Proposed solution

To ensure the smooth operation of deployed stateful microservices applications throughout their entire lifecycle, we propose a zero-touch management solution. This solution manages the lifecycle of stateful microservices applications from initial placement to migration or termination, ensuring no data loss during migration. Migration decisions are proactive and are based on machine learning models to forecast time series data. These models are trained on real-world datasets to predict future memory and CPU consumption values using historical data provided by the resource exposer component. One of the key features of our solution is the ability to migrate services across multiple clusters, providing a larger set of options during re-scheduling if the original cluster does not have the necessary resources available in any of its

Table 1
Comparison of related works based on various aspects.

Work	General idea	Application arch	Application nature	Migration	Proactive	Solution scope	Addressed state	Implemented
[23]	RL based migration	Microservice	Stateless	Yes	Yes	General	No State	Not Implemented
[25]	Use of OverlayFs to migrate application volumes	Monolithic	Stateful	Yes	No	General	Volumes	Single k8s cluster
[27]	C/R using DMTCP	Monolithic	Stateful	Yes	No	General	Container State	Single k8s cluster
[29]	Proactive solution to manage stateful applications	Monolithic	Stateful	Yes	Yes	General	Container State	Multiple custom k8s clusters (needs custom runtime version)
[30]	Fault Tolerance for Stateful Applications in Kubernetes with C/R	Monolithic	Stateful	Yes	No	General	Container State	Single k8s cluster
[31]	managing the availability of stateful microservice	Microservices	Stateful	No	No	General	Container State	Single k8s cluster
[32]	Persistent Volume Autoscaler	Monolithic	Stateful	No	No	General	Container State	Single k8s cluster
[33]	optimize the migration of some ML workloads	Monolithic	Stateful	Yes	Yes	Specific manich learning workloads	Volumes	Single k8s cluster
[34]	Migration Strategy under Resource Constraints	Microservices	Stateless	Yes	No	General	No state	Not Implemented
Our	Resiliency Focused Proactive lcm for Stateful Microservices.	Microservices	Stateful	Yes	Yes	General	Container State	Multiple k3s k8s and Opentishf clusters

computing nodes. Our solution implements a two-tier scheduling algorithm: the resource orchestrator component selecting the cluster in the first tier and the cluster manager selecting the computing node in the second tier. This approach balances scheduling time and finding near-optimal placements for scheduled microservices pods and a minimal latency between the application microservices. A high-level conceptual architecture of our proposed solution is illustrated in Fig. 1. The figure demonstrates how the architecture integrates with an infrastructure containing two Kubernetes clusters, each with two worker nodes. Additionally, it shows two microservices applications: their components are highlighted in yellow (Application 1) and green (Application 2). While all microservices of Application 2 are deployed within one cluster, some microservices of Application 1 are either scheduled or migrated to the second cluster due to a lack of resources in the first cluster at certain points in the application's lifecycle. For simplicity, not all Kubernetes components are depicted in the schema; instead, we focus on showcasing the components specifically added to the architecture. Our contributions to this architecture are highlighted in red. In the next subsections, we will detail each component and provide the end-to-end workflow of our solution to offer a better understanding.

4.1. Resource discovery and resource exposer

The resource discovery module has the role of providing the RO with information regarding the available network and compute resources across all clusters. It gathers this information from the resource exposer agent instances deployed on each cluster. Each of these components provides the client components deployed on their levels with real-time information related to the consumed and available resources in the infrastructure.

4.2. Resource orchestrator- RO

As mentioned earlier in the introduction, we assume a cloud-native system under the control of a central RO that orchestrates and manages applications' lifecycle on top of the local management systems of the clusters. Currently, a local management system can be K8s, k3s or openshift, targeting a wide range of device compatibility in the cloud or at the edge. To ensure application lifecycle management, the RO maintains a comprehensive view of all available resources in multiple accessible clusters. Information is sourced from the resource discovery module, which retrieves data from resource exposer components running on each cluster. Using these data, the RO acts as a first-tier (higher-level) scheduler when performing initial placement or migration. Scheduling a microservice involves choosing the most available cluster in terms of memory and CPU, which is done by calculating the cluster score for each cluster, at a time, for every pod, using the following formula:

$$Cscore = (1 - \alpha) \cdot VC_c + \alpha \cdot VM_c \tag{1}$$

Where α is a configurable parameter between 0 and 1, VC_c represents the percentage of current CPU available in the concerned cluster, and VM_c is the amount of available memory in the concerned cluster. The value of α influences the scheduling decision: the closer it is to 1, the more the RO will consider clusters equipped with memory resources; the closer it is to 0, the more it will prioritize clusters with higher available CPU values. By default, the value of α is set to 0.5, giving equal importance to memory and CPU. However, the administrator can reconfigure this value for specific resource-sensitive applications or microservices. Ultimately, the RO will choose the cluster with the

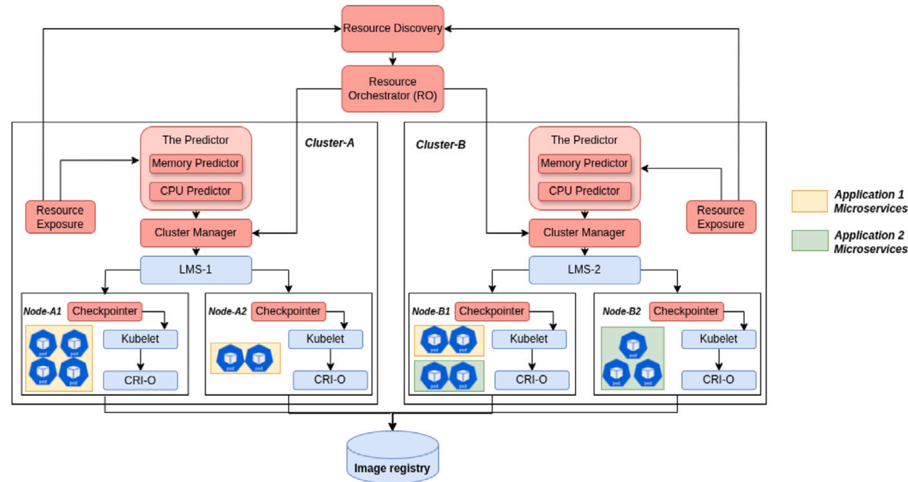


Fig. 1. Stateful applications LCM Architecture.

highest $Cscore$ and send the request to the cluster manager instance deployed on this cluster to continue the scheduling task.

An evident consequence of scheduling a pod on a certain cluster is the reduction of available resources in that cluster. This will likely affect the next scheduling target, potentially resulting in having two pods, of two microservices, from the same application, in different clusters. This can be problematic for applications following the microservices architecture, which are generally latency-sensitive. Having two microservices of the same application on different clusters can considerably affect performance. To address this issue, the RO proposes batch scheduling, which takes into account the application-level grouping of microservices. The mechanism ensures that microservices belonging to the same application are placed in the same cluster whenever sufficient resources are available. This approach aims to maximize the proximity of related microservices, reducing inter-cluster communication latency. The scheduling process prioritizes microservices based on their availability KPI, a value ranging from 0 to 100, which represents the required up-time for the microservice. For instance, microservices with high availability requirements (e.g., 99.99% uptime) are scheduled first and, in case of failures, are migrated only as a last resort to minimize disruptions.

4.3. Cluster manager

The cluster manager serves as the intermediary between the RO and the LMS, providing a unified language for higher-level components to access the underlying resources managed by the LMS, such the checkpointer operator. Unlike the RO, the cluster manager has a local view of the available resources in the nodes of the clusters. This information is provided by the resource exposer instances deployed on each cluster. The cluster manager acts as a second-tier (lower-level) scheduler, selecting the most available computing node for the microservice pod. This process is similar to what the RO does when choosing the most available cluster. The cluster manager calculates the node score N_{score} using the following formula:

$$Nscore = (1 - \alpha) \cdot VC_n + \alpha \cdot VM_n \tag{2}$$

Here, α is the same α used in the previous step, and VC_n and VM_n represent the current values of available CPU percentage and available memory on each node at the time. The node chosen to deploy the pod is the one with the highest $Nscore$. To ensure that our solution maintains high availability, we define a Safety Threshold. If the resource usage of a node exceeds this threshold, the node will not be considered during scheduling, provided there are other available nodes to use. However, microservices already hosted on that node will not be migrated immediately when the Safety Threshold is exceeded. Migration will only occur when the Migration Threshold is surpassed.

4.4. The predictor

As our solution is based on a proactive approach, having visibility on potential threats that can degrade system performance in the near future can help the system act before these threats manifest. One significant threat is the rapid increase in resource demand from different parts of a computing node, which can potentially degrade application performance or cause temporary node failures. These failures can lead to pod restarts, resulting in state loss due to the ephemeral nature of containers. To address this, we developed the predictor component, which consists of two machine learning time series forecasting models designed to predict future memory and CPU values for each computing node in every cluster within the federated infrastructure. Both models are based on the Long Short-Term Memory (LSTM) algorithm and are trained on a subset of the extensive public dataset GWA-T-13 Materna. This dataset was chosen due to its comprehensive coverage of real-world resource usage metrics in a distributed environment, it contains time series data from 1500 virtual machines, intervals making it highly relevant for capturing the complex temporal patterns of resource demand in federated infrastructures. Its large size and diversity also provide a solid foundation for building robust and generalizable predictive models. The data in GWA-T-13 Materna dataset are collected in 5-minutes intervals, for the best of our knowledge, this is the shortest data collection interval for publicly available datasets. The models demonstrate promising results, which are presented in Section 5.

The predictor component, which should be deployed on every cluster, gathers current CPU and memory consumption values from the resource exposer component and feeds these values to the ML models as vectors. The models then output the predicted CPU and memory consumption values. The number of input values provided to the models each time, known as the time window, is a crucial parameter in time series models. We will also discuss its impact on our models in Section 5. It is worth noting that both CPU and memory values are provided to each model because our experiments revealed that CPU and memory usage are generally correlated. When one is affected, the other is often impacted as well. Once predictions are made, the predictor sends the model outputs to the cluster manager, which compares these values to configurable thresholds. By default, the threshold values are set to 80% of the total memory size and 80% CPU usage. This means that if the predicted memory consumption value exceeds 80% of the total provisioned memory, or if the CPU is predicted to be working at least 80% of the time after the last prediction, the cluster manager will initiate stateful pod migration from the concerned machine.

Migrating a pod involves checking for another available node and saving the state of the containers inside the pod, which will lead

```
apiVersion: cache.eurecom.com/v1alpha1
kind: Migrator
metadata:
  name: migrator-sample
spec:
  source_pod_name: cache
  source_pod_name_space: default
  source_pod_container: redis
  imagePath: quay.io/netsofimageregisty/restore-redis:checkpointed-version
  registryUsername: abdelghaniemiliani
  registryPassword: password
  sourceNodeId: worker-node-A1
```

Fig. 2. Checkpointer Custom Resource sample.

us to discuss the checkpointer operator in Section 4.5. This proactive migration decision is crucial for maintaining system reliability and performance, ensuring no data loss during the migration process, and leveraging our machine learning models to predict and mitigate potential resource shortages.

4.5. Checkpointer operator

As presented in the background Section 2.4, Kubernetes now offers the checkpoint feature as an alpha feature at the kubelet level. However, this implementation presents significant challenges for application developers and system administrators. Currently, checkpointing containers require access to each running kubelet instance in the cluster and the credentials used by the Kubernetes API server to connect with kubelet. The kubelet API, typically can only be interacted with by the API server, but must be accessed directly by anyone needing to perform a checkpoint, necessitating the presentation of the API server's certificate. To effectively manage application lifecycles, it makes more sense to expose this feature at a higher level, such as the Kubernetes API server. The Kubernetes operator pattern is, by far, the simplest way to extend the Kubernetes API, providing additional features or managing new resources. Based on custom resource definitions (CRDs) [35], it allows developers to define the template of their custom resource instances and then code the logic that should be applied to every part of this resource's lifecycle (creation, update, deletion). A simple instance of our checkpointer custom resource is presented in Fig. 2. With our checkpointer operator, creating a container checkpoint is as simple as creating a Kubernetes pod. Users (or clients in general) only need to specify the pod namespace, the pod name, and the container name inside the pod. This resource can then be applied just like any native Kubernetes resource. Checkpoints are generally created to restore the process (or container) from which they were made. As mentioned in 2.4, to perform a restore, a container image must be created from scratch (without a base image) and must contain the checkpoint archive at the root of its file system. Additionally, specific annotations must be included in the container image metadata to inform CRI-O that the container is a checkpointed one. This allows CRI-O to proceed with restoring the old container from the checkpoint.

Since we are performing this for the purpose of container migration, the newly built image must be transferred to a different location, effectively moving the state of the container to the destination node. To facilitate this, we propose a shared image registry between all clusters of the federated infrastructure. This image registry can hold both application and microservice base images as well as checkpointed images, allowing them to be pulled from the destination during migration. This approach avoids the need for direct connections between all computing nodes, which can be very challenging to secure and maintain. The checkpointer operator uses the Buildah Golang package to build and push images to the image registry, making it lightweight and independent of any other system daemons or executable software. It is also designed to use threads, supporting multiple container saves simultaneously and showing good execution times, which will be presented in Section 5; however, first, to provide a clearer understanding of our solution in the next subsection, we present the general workflow of the component interactions.

4.6. Worfkow

A general simplified workflow between the system elements is represented in Fig. 3. To provide a better understanding of the different application lifecycle steps treated by our solution, we divide the workflow into eight steps, from initial deployment to application migration. For simplicity, we will not showcase the interaction between the LMSs and the underlying components, mainly kubelet and the container runtime, as these are not our contributions. In the remaining part of this section, we will detail each step of the workflow.

4.6.1. Step 1; first-tier scheduling

Before the deployment process begins, the application developer (or the entity deploying the application) must provide the RO with an application descriptor file, formatted in YAML or JSON. This file contains the detailed specifications of the application's microservices, including global parameters such as open ports, computing resource requests and limits, environment variables, and, most importantly, the availability KPI for each microservice. For brevity, we omit the detailed contents of this file. Upon receiving the application deployment request, the RO communicates with the resource discovery to obtain an overview of the infrastructure. Based on this information, the RO selects the most suitable cluster for deployment by considering resource availability. The microservices are then sorted by their availability KPI, ensuring that those with higher availability constraints are prioritized. Finally, the RO sends a request to the cluster manager instance deployed within the selected cluster to handle the microservices' deployment.

4.6.2. Step 2; second-tier scheduling

Upon receiving the deployment request from the RO, the cluster manager gathers information from the resource exposer component to select the most available node within the chosen cluster. After selecting the computing node for a given microservice, the cluster manager contacts the local LMS to enforce the decision and deploy the corresponding pod. This step, along with the next one, are repeated for each microservice in the application. However, in cases where the resource consumption of all nodes exceeds the predefined threshold, some microservices may remain unscheduled.

4.6.3. Step 3; pod deployment

The Local Management System (LMS) pulls the base images for the microservices from the image registry and starts the pods. After the loops in Steps 2 and 3 are completed for all microservices, it may happen that some microservices remain unscheduled. This can occur if the local LMS determines that the selected node lacks sufficient resources to meet a microservice's requirements, even if the node's overall resource consumption has not exceeded the threshold. In such cases, at the end of the loops of steps 2 and 3, the cluster manager reports back to the RO, indicating the non-deployed microservices. The RO then restarts the scheduling process from step 1 by searching for another suitable cluster. This iterative process continues until all microservices are successfully deployed.

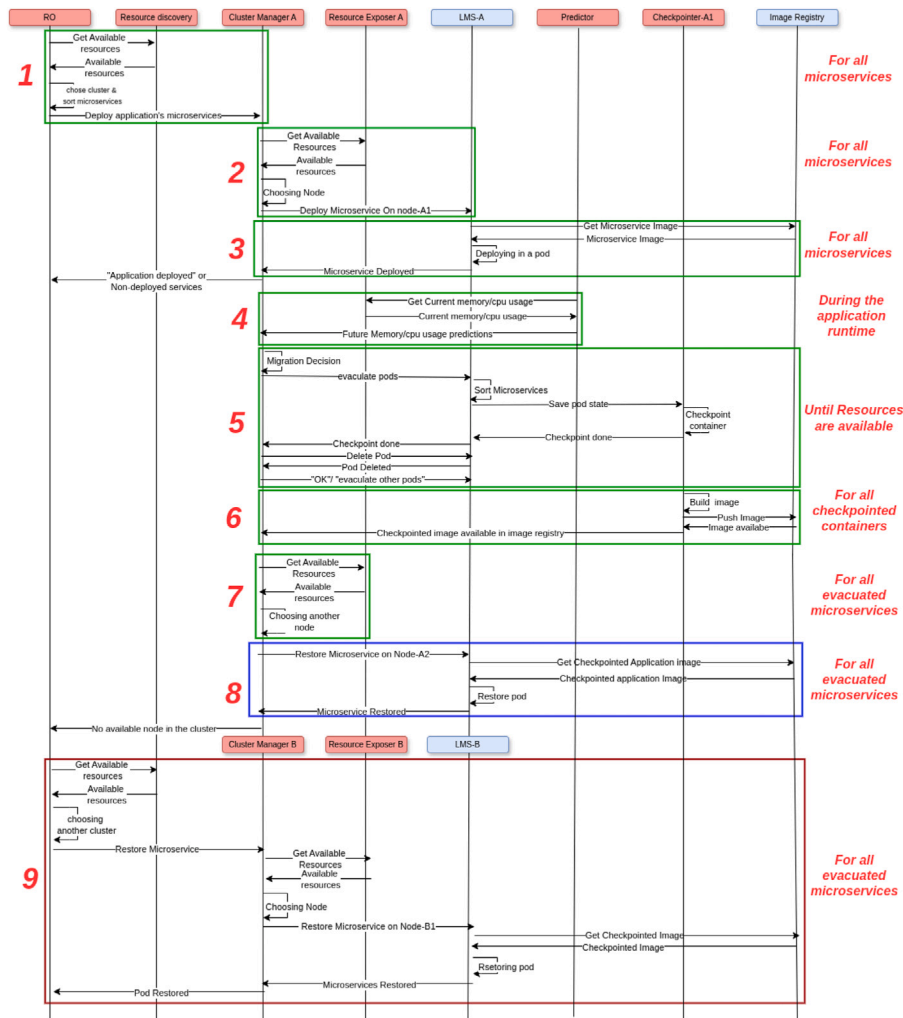


Fig. 3. Stateful application's life cycle management workflow.

4.6.4. Step 4; resource predictions

After scheduling all the application' microservices, the predictor continuously gathers information about resource consumption to predict future values. These predictions are sent to the cluster manager to inform proactive decisions.

4.6.5. Step 5; saving the container state

If the cluster manager detects that the predicted resource usage exceeds the set thresholds on a specific computing node, it triggers a migration process for the microservices deployed on that node. This process involves several steps, starting with sorting the microservices. However, unlike the initial deployment, the migration prioritizes microservices with lower availability requirements. This approach is chosen because migration can be time-consuming, depending on various factors highlighted in Section 5. By migrating microservices with lower availability constraints first, it is possible to reduce resource consumption to a level where migrating higher-priority microservices becomes unnecessary. Once the microservices are sorted by availability, the LMS proceeds by checkpointing the container states, deleting them, and repeating this process until the node's resource usage returns to acceptable levels.

4.6.6. Step 6 and 7; build images, looking for nodes

In parallel with Step 5, during Step 6, the checkpointer builds Docker images from the checkpoint files saved earlier and pushes them to the image registry to facilitate the migration process. Simultaneously, in Step 7, the cluster manager begins searching for other available nodes within the same cluster. Our solution prioritizes migration operations within the same cluster to maintain low latency between microservices. This approach is generally faster than inter-cluster migration due to factors such as the reduced latency between the infrastructure and the image registry. By focusing on local migrations, we aim to minimize downtime and ensure efficient rescheduling of evacuated microservices.

4.6.7. Steps 8, 9 ; rescheduling and restoring the pod

Depending on the results from step 7, If an appropriate node is found, Step 8 involves the cluster manager requesting the LMS to deploy the pod on this new node. Unlike Step 3, where the LMS uses the base microservice image, here, the LMS pulls the updated image that includes the container's state. This allows the pod to resume operation from its previous state. However, in situations where no suitable nodes are available, such as during peak times or in single-node clusters, if all nodes exceed certain resource thresholds, Step 8 comes into play. In this step, the issue is flagged to the RO, who then seeks out a new cluster for the pod. The RO instructs the cluster manager of the new cluster to allocate a node for the pod, akin to the initial placement process. Once a node is assigned, the new LMS retrieves the checkpointed image and restores the pod on the selected node in the new cluster, steps 7, 8 or 9, will be repeated until all evacuated microservices are re-scheduled again.

Table 2
Disk performance metrics.

Config	R IOPS	W IOPS	Sequential read	Sequential write
Config 1	30,000	20,000	400 MB/s	400 MB/s
Config 2	37,500	25,000	500 MB/s	500 MB/s
Config 3	45,000	30,000	600 MB/s	600 MB/s

5. Tests and results

In this section, we will showcase the results obtained from our solution across different phases of the application lifecycle in different environments and with various hardware resources and use cases. This section is divided into two main subsections. The first subsection will discuss the results obtained from our solution before migration, focusing primarily on the predictor component's results. The second subsection will examine the solution's performance during migration, focusing on the container's state-saving time, rescheduling time, and pod restart time under different scenarios, such as single cluster and multi-cluster migration, in on-premises and public cloud servers and to validate our results, we repeated our tests multiple times under the two different scenarios using different hardware configurations.

- In the multiple cluster migration scenario, we used resources from the IONOS public cloud provider. We provisioned two single-node clusters in separate geographical data centers: the original cluster, which hosted the pods, was located in a data center in France, while the destination cluster was situated in Germany. Both nodes were equipped with 4 Intel Ice Lake CPU cores, 4 GB of RAM, and an SSD hard disk. The IONOS cloud platform offers various (but limited) disk performance options, so we tested different configurations on the source node to investigate how changes in disk performance impact migration time, particularly the container state saving time, the disk configurations that we used in our tests for the multi-cluster migration scenario are summarized in Table 2. In this scenario, we used the public QUAY image registry as the image registry. This choice was made to ensure that the setup is cluster-independent. In this scenario, we will refer to the source node as *France-node* and the destination node as *Germany-node*.
- In the single cluster migration scenario, we implemented a Kubernetes cluster configuration consisting of one virtual machine serving as the master node and two additional virtual machines designated as worker nodes. The image registry, which hosts the checkpointed images, was deployed on the cluster master node. Each machine was configured with 4 CPU cores and 4096 MB of RAM. The physical machine hosting these virtual machines features a 13th Gen Intel Core i7-1365U processor, with a maximum frequency of up to 3.70 GHz for E cores and up to 5.00 GHz for P cores, accompanied by 32 GB of LPDDR5-6400 MHz memory. The physical machine provisioning the cluster is equipped with an SSD hard disk, which has the following Input/Output Operations Per Second (IOPS) and sequential read and write performance metrics: Sequential Read: Up to 7000 MB/s; Sequential Write: Up to 5100 MB/s; Random Read IOPS: Up to 1,000,000 IOPS; Random Write IOPS: Up to 1,000,000 IOPS. In this scenario, *worker-node01* will serve as the source node for the migration, while *worker-node02* will act as the destination node.

For both scenarios, we used Kubernetes v1.26.0 as the local management system for the clusters and CRI-O v1.26.0 as the container runtime to run containers in the nodes of the clusters.

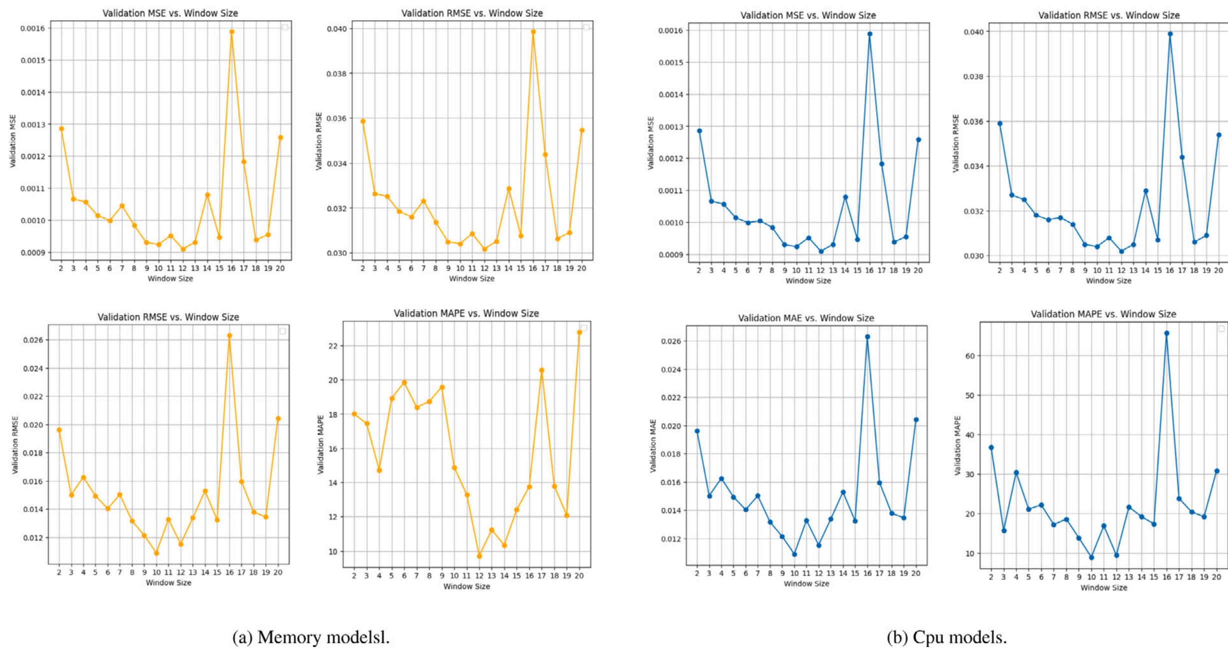
5.1. Before the migration

As we mentioned earlier, the predictor component of our solution consists of two LSTM models designed to forecast CPU and memory time series data. In the training phase, both models take the same input: a vector containing past memory and CPU usage values. This approach was chosen because our experiments demonstrated that using both metrics as input yielded better results than training each model solely on its respective output (i.e., training the CPU model only on CPU values and the memory model only on memory values). This improvement is due to the correlation between the two metrics. The models are trained on two subsets of the GWA-T-13 Materna dataset, each subset contains around 10K time series values, obtained by merging data from different types of machines to ensure compatibility with a wide range of input values. The LSTM algorithm was selected for its ability to effectively capture long-term dependencies and patterns in sequential data, making it well-suited for our large subsets. We trained the models using different time window sizes, representing the length of past data sequences used as input for the model. To evaluate and compare the results of our models, we use commonly employed time-series evaluation metrics: Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and Mean Absolute Percentage Error (MAPE).

Figs. 4(a) and 4(b) present the changes in evaluation metrics for the memory and CPU models as the time window values vary from 2 to 20. Examining both figures, we see that the models demonstrate similar behavior in response to changes in the time window parameter. Initially, increasing the time window positively affects model performance, enhancing accuracy up to a certain point—12 time periods for the memory model and 10 time periods for the CPU model. Beyond these points, further increases in the time window do not improve model performance and may lead to unpredictable results. This pattern can be explained by the hypothesis that current memory and CPU consumption are more strongly related to recent values rather than older data. Given that the GWA-T-13 Materna dataset is sampled at 5-minute intervals and since we are predicting only a single near feature value, the model does not need to detect from distant past periods, so it is reasonable to assume that current memory and CPU values are mainly influenced by the most recent 10 to 12 time periods. Therefore, incorporating older values with a larger time window does not provide significant benefits for predicting current memory and CPU usage. Considering that the predictor needs to function in real-time, relying solely on time series validation metrics is insufficient. The predictor must operate smoothly so the entire system remains unaffected by model delays. Therefore, we also present the average inference time of both types of models, measured after running the models on an *Intel i7 1355 U* processor, in Fig. 5.

Looking at these values, we observe that increasing the time window generally results in a longer inference time. However, this increase is negligible (on the scale of milliseconds), allowing us to select the best-performing pair of models for deployment in our solution.

The metrics for the selected models are detailed in Table 3. As observed from the table, the values for the MSE, MAE and RMSE are notably low, indicating strong model performance. MSE, in particular, is designed to penalize larger errors more heavily due to its quadratic nature, meaning that it gives greater emphasis to outliers. The low MSE values suggest that our models effectively minimize significant errors. This observation is further corroborated by the Mean Absolute Percentage Error (MAPE) values, which show that the average prediction errors are 9.71% for the memory model and 8.96% for the CPU model, which we consider good values in our context.



(a) Memory modelsl. (b) Cpu models.

Fig. 4. model's MSE, RMSE, MAE, and MAPE across Different Window Sizes.

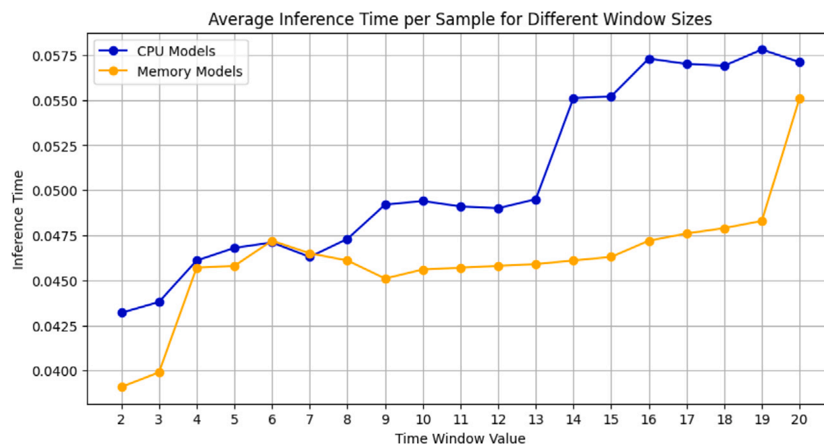


Fig. 5. Average Inference Time (Seconds) for both resource models.

Table 3
Models performance metrics.

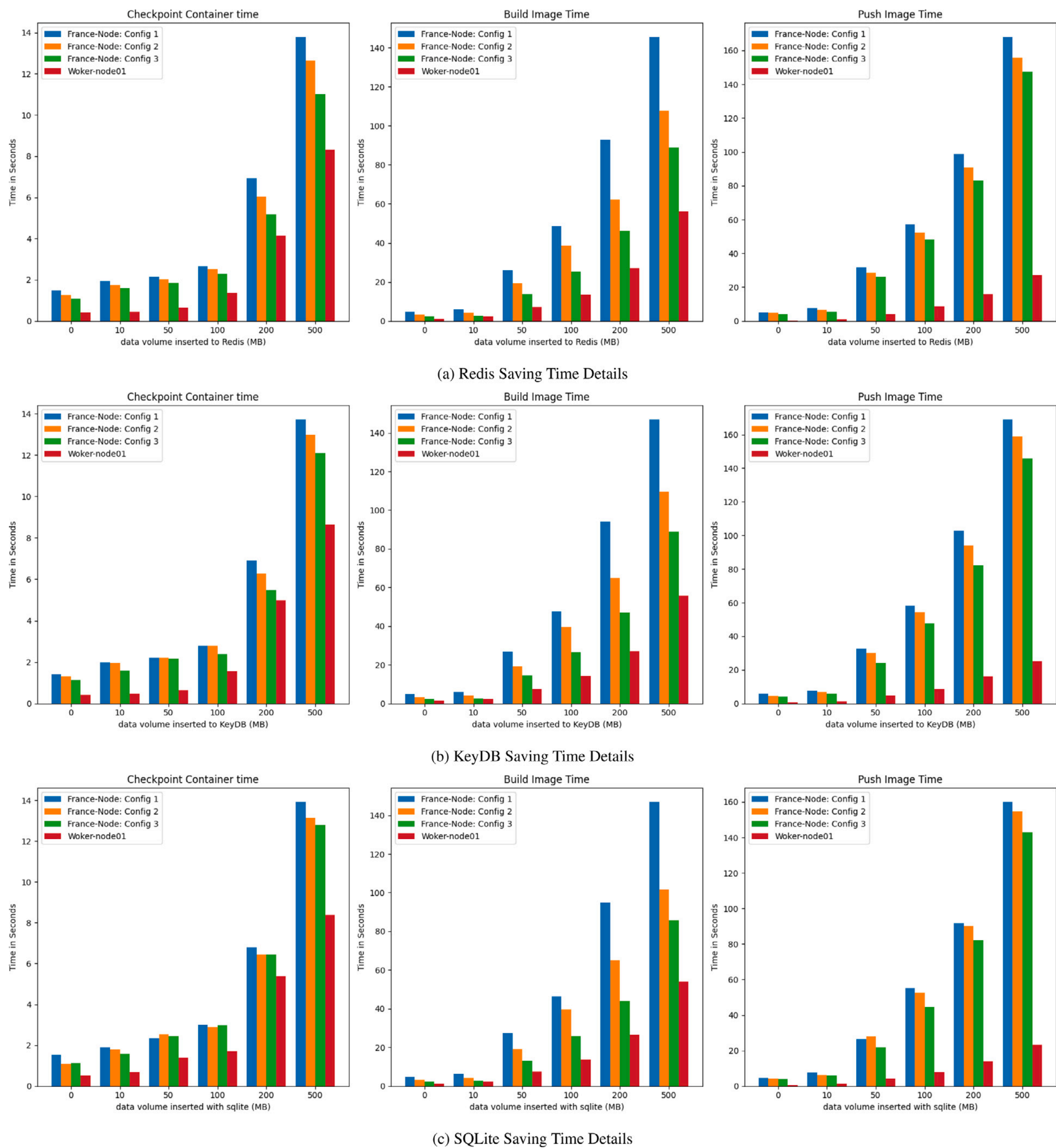
Model	MSE	RMSE	MAE	MAPE (%)	Inference time (S)
Memory	0.002148	0.046349	0.017732	9.716583	0.0458
CPU	0.000924	0.030405	0.010894	8.962406	0.0494

5.2. During the migration

In this section, we will focus on the results of our solution during the migration process. This period spans from the migration decision trigger to the restart of the application at the destination. Specifically, the migration time includes the sum of the time required to save the pod containers, the re-scheduling time, and the time needed to restart the application on the destination node. Minimizing the migration time involves reducing each of these three components. We will discuss each of these times in separate subsections. To evaluate the solution's effectiveness with different volumes of container states, we various in-memory databases, specifically Redis, KeyDB, and SQLite. We injected datasets of different sizes—10 MB, 50 MB, 100 MB, 200 MB, and 500 MB—to generate containers of varying size. These containers represented different states that our operator needed to save.

5.2.1. Container state saving time

As previously mentioned, saving the container state involves checkpointing the container, creating a new image from this checkpoint, and pushing this image to an image registry. Once the image is successfully pushed, the container state is considered saved. To provide a deeper insight into how each of these processes contributes to the total migration time, Fig. 6 presents the average time, in seconds, required to complete each of these processes for different container sizes or each type of database on the source computing nodes in both scenarios. Specifically, Fig. 6(a) shows the detailed saving time for the Redis database, Fig. 6(b) presents the saving time for KeyDB, a Redis alternative, and Fig. 6(c) details the saving time for the SQLite database. It is worth noting that for SQLite, we are using the standard Python library, which results in a larger base image (76.68 MB compressed) compared to the official images used for Redis (16.04 MB) and KeyDB (26.67 MB).



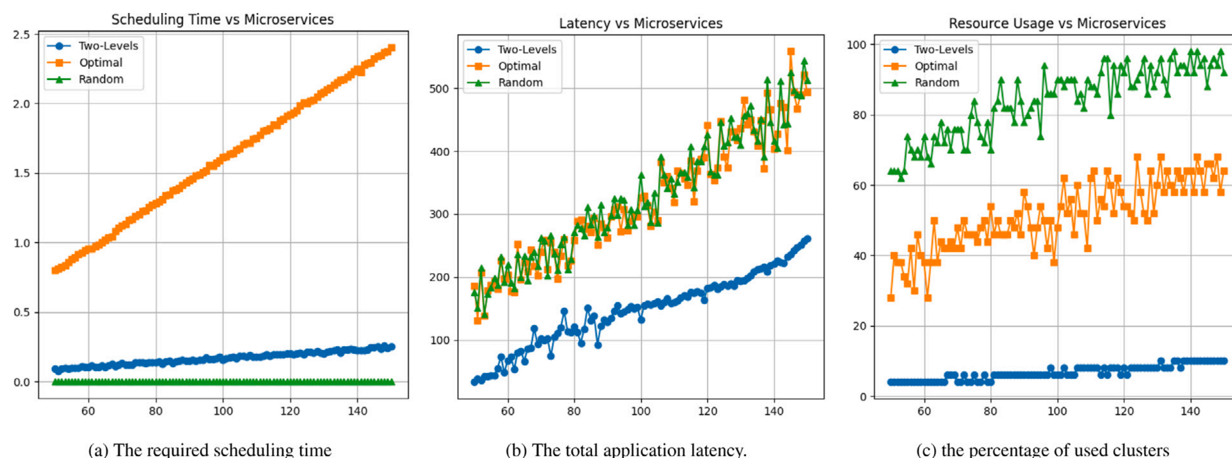
(a) Redis Saving Time Details

(b) KeyDB Saving Time Details

(c) SQLite Saving Time Details

Fig. 6. Average time, required to complete checkpointing, image creation, and image pushing processes for different container sizes and container images on the source computing nodes in both migration scenarios.

By examining the figure, we can observe two key points. (1) First, as container size increases, the time required for checkpointing, creating a new image from this checkpoint, and pushing the image also increases for all types of applications, resulting in longer saving times. (2) Second, altering the hard disk configuration impacts each of these times, particularly the image-building time. This is because building an image from a checkpoint file involves copying the filesystem from the node to the container, a process that benefits from better disk performance and thus takes less time with higher-performing disks. Diving deeper, we observe that, except for the single-cluster migration scenario, image pushing time constitutes the largest portion of the overall container saving time. This can be explained by the fact that, in multi-cluster migration scenarios, the image registry is typically remote and shared among multiple entities. This setup introduces significant network latency between the remote registry and the source migration nodes, unlike in single-cluster scenarios where the image registry is local and private. Additionally, unlike computing resources, IONOS cloud providers offer limited and non-configurable network resources, which further contributes to the increased time required to push the image in multi-cluster migrations compared to single-cluster scenarios. Another observation is that the different phases of saving a container's state are generally influenced by hardware resources, the registry location, and the size of the state. However, the type of application does not appear to be a decisive factor, as all three databases exhibit similar patterns across every phase of the saving process. This means that factors like the base image size will not affect the service container saving time, only the newly injected data will do. This finding is valuable because it simplifies the design and optimization of systems handling container



(a) The required scheduling time (b) The total application latency. (c) the percentage of used clusters

Fig. 7. Multi-Clusters Scheduling techniques compassion.

state management. Since the application type has minimal impact, developers and operators can focus on optimizing hardware resources, registry placement, and state size without needing to tailor solutions for specific database types.

5.2.2. Rescheduling time

After saving the container state, the second phase in the migration process involves assigning the pods hosting the saved containers to new, healthy nodes. This process, known as rescheduling the pods, occurs at two levels: first, by choosing the cluster, then by selecting the computing node based on their resource availability scores. In our tests, the time taken to reschedule the application was negligible, typically in the tens of milliseconds. However, these tests were conducted on a relatively small testbed, so the results may not be representative of larger environments. Testing a multi-cluster scheduling solution across many clusters and nodes would require substantial hardware resources that we do not currently possess. To address this limitation and demonstrate the effectiveness of our solution in larger environments, we used simulation. we modeled an infrastructure consisting of 100 clusters, each containing between 5 and 10 computing nodes. We assumed that the nodes were identical in terms of the resources they provide. Initially, the percentage of available resources (ranging from 20% to 80% for both CPU and memory) was assigned randomly at the start of each experiment. Our goal was to compare our solution with two alternatives: one that selects the best available node across the entire infrastructure and another that performs random scheduling. The comparison focused on three key metrics: scheduling time, latency between microservices, and the percentage of utilized clusters in the total infrastructure. We varied the number of microservices from 50 to 150, with each microservice having specific resource and availability requirements. Each microservice application was modeled as a Directed Acyclic Graph (DAG), where nodes represented microservices, and edges represented the connections required between them. The infrastructure was represented as a weighted graph, with nodes representing the computing nodes and edges representing latency between clusters. For latency values we used abstract numerical values for measurement, latency between connected clusters were varied between 1 and 5, and for infrastructures without direct links, the latency was calculated as the shortest path between clusters. We assumed that nodes within the same cluster could directly access one another, with an inter-node latency of 0.1. The latency between two microservices was equal to the latency between the nodes that hosted them, meaning that if two microservices were hosted on the same node, the latency between them would be 0. The overall latency of the application was determined by the sum of the latencies between each pair of connected microservices.

The results presented in this section are the averages of 1000 repetitions of our experiments. Fig. 7 illustrates the outcomes of these experiments.

Fig. 7(a) represents the scheduling time required to schedule microservice applications with different configurations using the three solutions. The figure shows that the optimal solution takes more time to complete the scheduling compared to both our solution and the random solution. This is because, for each microservice, the optimal solution must search for the most available node across the entire federated infrastructure, which increases the search space. In contrast, our approach first selects the cluster and then schedules as many microservices as possible within that cluster, reducing the search space to the set of the cluster nodes. While the optimal solution ensures that microservices are hosted on nodes with the highest available computing resources, this approach can become time-consuming depending on the number of microservices and the size of the infrastructure. The random solution, on the other hand, is the least time-consuming, as it randomly selects nodes from all those that meet the application's resource requirements.

Fig. 7(b) represents the overall application latency after being scheduled using the solution that seeks the optimal node, our solution, and the random algorithm. We observe that the latency is significantly lower with our solution compared to the two other alternatives. This is because our solution prioritizes scheduling as many microservices as possible within the same cluster, which helps minimize the inter-cluster penalty associated with connecting microservices.

Fig. 7(c) shows the percentage of used clusters after scheduling the microservices using the three solutions. We can observe that our solution limits infrastructure usage for the same reason—maximizing the number of microservices hosted within the same cluster. In contrast, the optimal solution searches for the best node regardless of its location, which leads to a higher percentage of clusters being used.

To summarize, we believe that the two-tier scheduling solution strikes a good balance between selecting nodes with high availability to host the microservices and minimizing scheduling time, which is much closer to the random approach that imposes no additional constraints. Additionally, our solution results in a lower percentage of used clusters and provides the minimum latency for the scheduled application compared to both other alternatives (see Fig. 7).

5.2.3. Restore time

Fig. 8 shows the average application's restore time for the three in-memory databases used in our experiments, based on the data injected prior to migration. This time measurement spans from the moment the destination computing node is chosen to the point when the container is fully operational again. The results reveal several important observations. Increasing the size of the container state leads to a longer restore time due to the additional data that must be transferred during the restore phase. Disk performance also plays a role in the restoration process, with better disk performance resulting in shorter

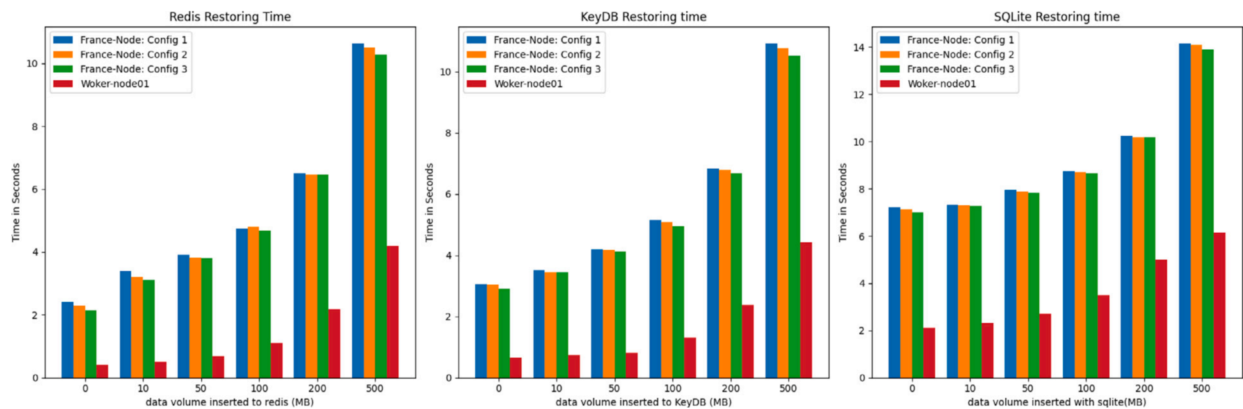


Fig. 8. Application restore times with varying disk configuration.

restore times, though this impact is negligible compared to the disk's influence on the checkpointing process. Furthermore, in a single-cluster migration scenario, the restoration of the container state takes less time compared to multi-cluster scenarios because the local image registry in the same cluster reduces latency when accessing the checkpointed image. In contrast, multi-cluster scenarios involve remote image registry access, which introduces additional latency. Among the tested databases, SQLite required more time to restore than Redis and KeyDB. This is primarily due to the need to fetch the checkpointed image from the image registry and the additional step of fetching the base image from its respective registry. The larger the base image, the longer the pull time. However, if the base image is already present on the machine, this process is bypassed. These findings provide valuable insights for future research, particularly in the development of scheduling policies that prioritize infrastructure already hosting the service to be migrated. Such an approach ensures the base image is already present, reducing restoration time. In our tests, we intentionally cleaned the images from the machines' disks before each iteration to ensure that results were derived from independent tests and not influenced by residual data from previous iterations.

6. Discussion and limitations

The migration of stateful microservices offers both opportunities and challenges, especially in environments that demand high reliability and minimal downtime. While the proposed solution presents a novel approach to address this issue, it is important to consider its broader implications, including trade-offs, limitations, and practical challenges in real-world deployments. The proposed method manages the entire lifecycle of microservice applications, from initial deployment to handling failures and migrations. This minimizes human intervention and ensures that changes are seamless for application users. Additionally, the architecture's components are standalone, which simplifies adapting the solution for other use cases, such as user mobility or energy consumption reduction. In such cases, only the decision-making components need adjustment, without altering the decision-enforcement mechanisms.

To assess the impact of automating failure handling through migration, we need a deeper evaluation of the prediction method used and a comparison with the reactive approach. For this purpose, we used the same sub-dataset from the GWA Materna traces to train six additional models. This time, our objective is not only to predict the next value, but also to forecast the next 2, 3, and 4 values. Specifically, we aim to predict changes in both CPU and memory usage for the next 10, 15, and 20 min. The overall model architecture remains the same as in the first two models, but the output dimension changes from 1 (predicting resource consumption for the next 5 min) to 2, 3, and 4 values, corresponding to the next 10, 15, and 20 min. For simplicity, we will refer to these models as ***Model1***, ***Model2***, ***Model3***, and ***Model4*** throughout the remainder of this paper, for both CPU and memory predictions.

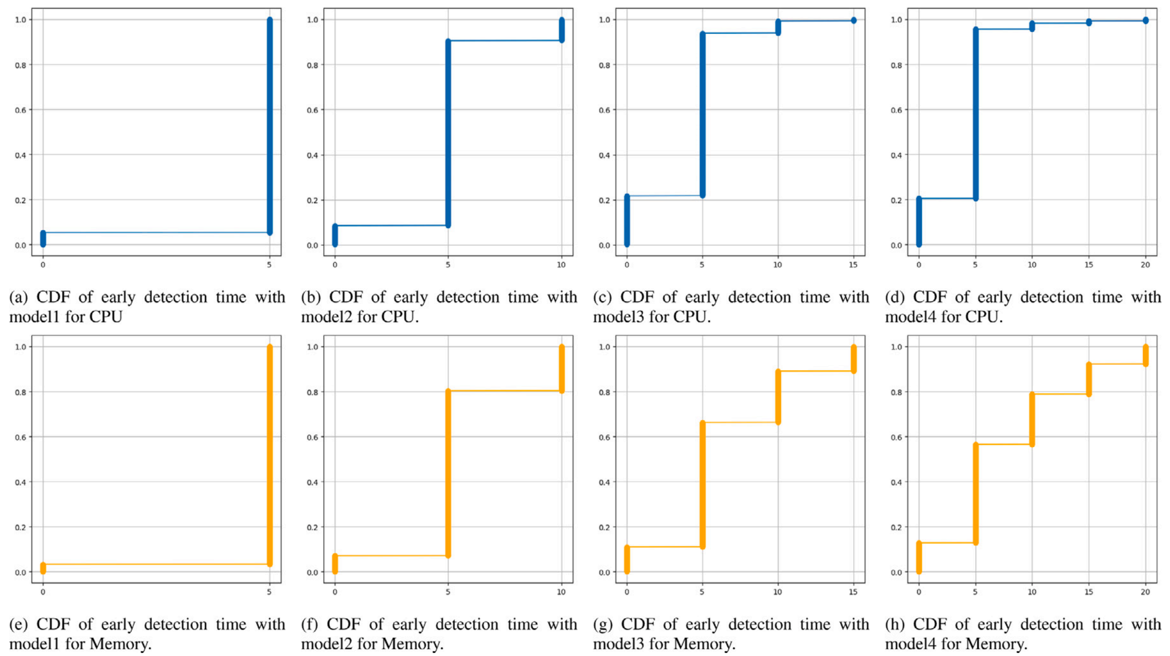
Our goal is to compare our approach with the reactive approach using two main metrics. The first metric is the early detection time—how much earlier our proactive approach identifies threshold breaches. Detecting these breaches earlier is crucial because it allows for timely mitigation actions (eg. container states saving), reducing the risk of system instability, service degradation, or downtime. This ensures smoother resource management and provides sufficient time to optimize migration strategies, leading to improved system performance and higher availability. The second is the number of avoidable migrations when using our solution compared to the reactive approach. For both the CPU and memory models, we define a migration threshold S representing the percentage of utilized resources. Migration decisions are triggered when resource usage exceeds this threshold. Predicted and real values are categorized as either greater than or equal to S or less than S, resulting in four possible cases:

1. **True Positive Prediction:** Both predicted and real values are greater than or equal to S.
2. **False Positive Prediction:** The predicted value is greater than or equal to S, but the real value is less than S.
3. **True Negative Prediction:** Both predicted and real values are less than S.
4. **False Negative Prediction:** The predicted value is less than S, but the real value is greater than or equal to S.

In general, true positive predictions lead to time savings, false positive predictions result in avoidable migrations, and false negative predictions lead to identical behavior in both reactive and proactive approaches. However, for simplicity, we omit the detailed calculations of time saved and the number of avoidable migrations. We varied the threshold S between 75% and 80% for both CPU and memory models.

6.1. Early threshold detection

Fig. 9 presents the cumulative distribution functions (CDFs) of the early detection time for each model. Subfigures (a) to (d) represent the CPU models, while subfigures (e) to (h) correspond to the memory models. In all cases, a value of 0 indicates scenarios where the proactive approach behaves similarly to the reactive one. This occurs when the real value exceeds the threshold S without the model predicting it at any stage. We observe that the CPU and memory models share certain patterns but diverge in specific instances. Figs. 9(a) and 9(e) illustrate *Model1* for both CPU and memory. These models demonstrate strong performance, with the CPU model predicting threshold breaches 5 min in advance in 94.63% of cases, and the memory model achieving



(a) CDF of early detection time with model1 for CPU

(b) CDF of early detection time with model2 for CPU.

(c) CDF of early detection time with model3 for CPU.

(d) CDF of early detection time with model4 for CPU.

(e) CDF of early detection time with model1 for Memory.

(f) CDF of early detection time with model2 for Memory.

(g) CDF of early detection time with model3 for Memory.

(h) CDF of early detection time with model4 for Memory.

Fig. 9. CDFs of early detection time using each model. Subfigures (a) to (d) represent CPU models, while subfigures (e) to (h) represent memory models.

Table 4
Detection rates for CPU and memory models at various time intervals.

Model	Not detected	5-min early	10-min early	15-min early	20-min early	Total detected
cpu-model1	5.37%	94.63%	/	/	/	94.63%
cpu-model2	8.51%	82.08%	9.41%	/	/	91.49%
cpu-model3	21.72%	72.12%	5.41%	0.7%	/	78.28%
cpu-model4	20.51%	75.18%	2.66%	0.98%	0.67%	79.49%
mem-model1	3.27%	96.73%	/	/	/	96.73%
mem-model2	7.12%	73.24%	19.63%	/	/	92.12%
mem-model3	11.02%	55.31%	22.8%	10.87%	/	88.98%
mem-model4	12.81%	43.72%	22.37%	13.31%	7.78%	87.19%

96.73%. This allows the system more time to respond compared to the reactive approach.

Figs. 9(b) and 9(f) represent *Model2* for CPU and memory. In this case, the CPU model predicts 82. 08% threshold exceedances 5 min earlier and 9. 41% 10 min earlier, but it fails to detect 8. 51% of the cases. This is a higher failure rate compared to CPU Model1. A similar pattern is observed in Memory Model2, where 73.24% of cases are detected 5 min earlier, 19.63% 10 min earlier, and 7.12% remain undetected.

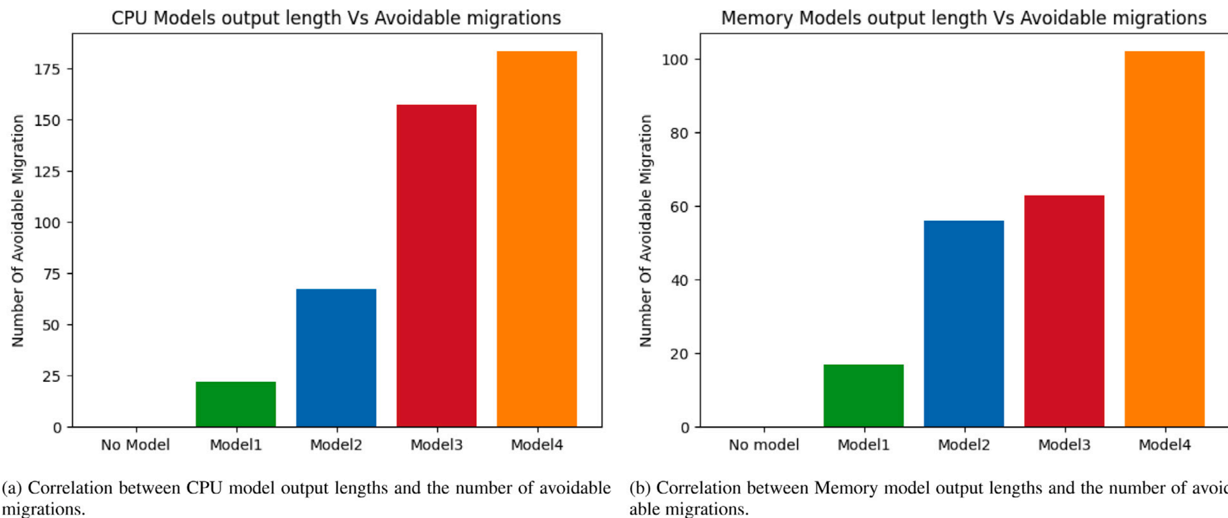
The remaining values are summarized in Table 4. A common pattern observed in both types of models is that increasing the prediction time interval allows the models to detect some threshold exceedances earlier. This effect is especially notable in the memory models, where considerable time is gained compared to the CPU models. We attribute this to a main factor: the nature of CPU data, which fluctuates much more rapidly than memory consumption. However, while predicting 10, 15, or even 20 min ahead provides ample time to mitigate potential failures, we observe a decline in overall model accuracy as the output dimension increases. For example, in *CPU-Mdel4* and *Memory-Model4*, the percentage of undetected threshold exceedances reaches 12.81% out of 10,000 samples tested, which we consider a substantial figure in our case. Relying on such models could result in the system frequently responding reactively rather than proactively. Based on the results described in Section 5.2.1, we can assume that a 5-minute early detection is sufficient to preserve the states of a wide variety of stateful microservice containers, especially since we prioritize intra-cluster migrations. This is why we chose *Model1* for both types, which predicts only the very next value, and we consider it suitable for a significant number of use cases. However, to further improve this finding, we were also interested in evaluating the number of avoidable migrations. This refers to the cases where the deployed model predicts that the consumption value will exceed the threshold, but the actual value does not.

6.2. Avoidable migrations

Fig. 10 shows the number of avoidable migration decisions for Model1, Model2, Model3, and Model4, for both CPU and memory. It is evident that as the output length increases, the number of avoidable migrations rises significantly. For instance, in the memory models, the number of avoidable migrations increases from 17 in *Memory-Model1* to 56 in *Memory-Model2*, while in the CPU models, it increases from 22 in *CPU-Model1* to 67 in *CPU-Model2*. We believe this increase is due to the nature of the data used to train these models, which is collected at 5-minute intervals. This interval is relatively long in this context, as significant changes can occur between two-time steps, making resource consumption forecasting more challenging.

As discussed in Sections 5.2.1 and 5.2.3, enforcing migration decisions is sometimes time-consuming depending on the infrastructure performance, especially when the actual resource utilization situation requires inter-cluster migrations. For this, avoidable migrations should be limited.

In this section, we examined the implications of automating microservices migrations, this decision should be carefully considered based on several factors, including the specific application context, the availability requirements of the microservices, the resources available



(a) Correlation between CPU model output lengths and the number of avoidable migrations.

(b) Correlation between Memory model output lengths and the number of avoidable migrations.

Fig. 10. Relationship between CPU and memory model output lengths and the number of avoidable migrations. The results show that avoidable migrations increase with longer model outputs.

within the infrastructure, and the importance of preserving the microservices' states. Additionally, the location of the image registry and the destination for migration—whether within the same cluster or to a different one—are crucial considerations. Infrastructure owners need to assess their ability to automate microservices migrations effectively and determine the level of automation that can be reasonably assumed, balancing the benefits of proactive resource management with the operational complexities involved.

7. Conclusion

In this work, we introduced a zero-touch management solution for stateful microservices applications, designed to minimize human intervention throughout the application lifecycle and enhancing system resilience under increasing computational demands. Our approach operates effectively in multi-cluster environments, leveraging a decoupled software architecture to enable seamless integration across diverse other use cases.

The solution employs a comprehensive workflow encompassing all stages of the application lifecycle, from scheduling to migration decisions. It is powered by a two-tier scheduling algorithm that efficiently places and migrates microservices. Based on our tests, this algorithm achieves a good balance between finding available nodes and maintaining reasonable execution times compared to the optimal solution, all while keeping latency lower than alternative methods.

By leveraging time series models trained on real-world datasets, our solution predicts future resource demands with high accuracy and short inference times. When predicted demands exceed a configurable migration threshold, the system initiates migration operations. These migrations are supported by a checkpointing operator that saves the state of the affected microservices as container images, which are stored in an image registry for post-migration use.

Our results demonstrate that container state saving times are influenced by several factors, such as the size of the container state, the computing power of the node, and the location of the image registry. Nodes with higher hardware capabilities save states more quickly, and using local image registries within the same cluster significantly reduces latency during the push phase. Additionally, we observed that smaller base image sizes expedite the restoration of microservices, highlighting the need to prioritize such applications during migrations.

We also discussed the impact of automating migration decisions, considering the level of automation that can be reasonably assumed, as well as the factors influencing this decision. The solution was rigorously tested across a wide range of multi-cloud infrastructure configurations and local environments, with the scheduling algorithm evaluated through simulation.

Based on our findings, our future research will focus on three key areas:

1. **Image Registry Placement:** As shown in the results section, the time spent saving the state to the image registry is the most time-consuming part of the entire migration process. A significant portion of this time is spent during the transfer of the state to the image registry. In the case of local migrations, this time is relatively minimal because the image registry is located closer to the source node. In contrast, during multi-cluster migrations, where a remote image registry is used, this time increases due to the added network distance. To address this issue, we are exploring a solution aimed at minimizing the time spent on state transfer. We believe that the strategic placement of image registries, considering factors such as network bandwidth and latency, can help reduce the time required for this operation.
2. **Parallel Image Layer Pulling:** Docker images are composed of multiple layers, each hosted in an image registry and identified by its unique ID. Currently, these layers are typically pulled sequentially by target hosts to avoid overloading network resources. However, we believe that developing techniques to pull image layers in parallel could significantly reduce application provisioning times and, consequently, application migration times. This approach has the potential to provide a tangible contribution to various industries.
3. **Real-World Time Series Datasets:** We noted in the discussion and limitations section that collecting data at longer intervals can negatively impact the accuracy of machine learning models over time. To overcome this, we envision the development of a new time series model that collects compute and network resource consumption metrics at shorter intervals (e.g., 5–10 s). We believe this will enhance the accuracy of machine learning models, including time series forecasting.

CRediT authorship contribution statement

Abd Elghani Meliani: Writing – original draft, Software, Investigation, Formal analysis, Conceptualization. **Mohamed Mekki:** Writing – review & editing, Supervision, Software. **Adlen Ksentini:** Writing – review & editing, Supervision, Methodology.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work was partially supported by the European Union's Horizon Research and Innovation program under AC3 project and grant agreement No 101093129.

Data availability

Data will be made available on request.

References

[1] O. Bentaleb, A.S.Z. Belloum, A. Sebaa, et al., Containerization technologies: taxonomies, applications and challenges, J. Supercomput. 78 (2022) 1144–1181, http://dx.doi.org/10.1007/s11227-021-03914-1.
[2] N. Dragoni, et al., Microservices: Yesterday, today, and tomorrow, in: M. Mazzara, B. Meyer (Eds.), Present and Ulterior Software Engineering, Springer, Cham, 2017, pp. 195–216, http://dx.doi.org/10.1007/978-3-319-67425-4_12.
[3] N. Toumi, M. Bagaa, A. Ksentini, Machine learning for service migration: A survey, IEEE Commun. Surv. & Tutorials 25 (3) (2023) 1991–2020, http://dx.doi.org/10.1109/comst.2023.3273121.
[4] A. Aissioui, A. Ksentini, A.M. Gueroui, T. Taleb, On enabling 5G automotive systems using follow me edge-cloud concept, IEEE Trans. Veh. Technol. 67 (6) (2018) 5302–5316, http://dx.doi.org/10.1109/TVT.2018.2805369.
[5] kubernetes an opensource container orchestrator: https://kubernetes.io/.
[6] S. Moreschini, F. Pecorelli, X. Li, S. Naz, D. Hästbacka, D. Taibi, Cloud continuum: The definition, IEEE Access 10 (2022) 131876-131886, http://dx.doi.org/10.1109/ACCESS.2022.3229185.
[7] H. Chergui, A. Ksentini, L. Blanco, C. Verikoukis, Toward zero-touch management and orchestration of massive deployment of network slices in 6G, IEEE Wirel. Commun. 29 (1) (2022) 86–93, http://dx.doi.org/10.1109/MWC.009.00366.
[8] A. MEliani, Migrator operator: Checkpointer operator source code, 2024, GitHub. Retrieved from https://github.com/abdelghanimeliani/migrator_operator.
[9] Stateful service migration between computing clusters, 2024, [YouTube video]. Retrieved from https://www.youtube.com/watch?v=wLnlEQ1wy54.
[10] redhat OpenShift documentation website: https://www.redhat.com/fr/technologies/cloud-computing/openshift.
[11] redhat OKD; Origin community Distribution of Kubernetes: https://docs.okd.io/.
[12] Y. Xiong, Y. Sun, L. Xing, Y. Huang, Extend cloud to edge with KubeEdge, in: Proceedings of the 2018 IEEE/ACM Symposium on Edge Computing, SEC, IEEE, 2018, pp. 373–377, http://dx.doi.org/10.1109/SEC.2018.00048.
[13] k3s;The certified Kubernetes distribution built for IoT & Edge computing: https://k3s.io/.
[14] docker: https://www.docker.com/.
[15] podman: a daemonless, open source, https://podman.io/.
[16] containerd: An industry-standard container runtime https://containerd.io/.
[17] CRI-O a CRI Compatible Container Runtime https://cri-o.io/.
[18] kubelet checkpoint api documentation: https://kubernetes.io/docs/reference/node/kubelet-checkpoint-api/.
[19] A. Widjajarto, D.W. Jacob, M. Lubis, Live migration using checkpoint and restore in userspace (CRIU): Usage analysis of network, memory, and CPU, Bull. Electr. Eng. Informatics (ISSN: 2302-9285) 10 (2) (2021) 837–847, http://dx.doi.org/10.11591/eei.v10i2.2742, Available at: https://www.beei.org/index.php/EEI/article/view/2742 (Accessed 13 December 2024).
[20] P. Martin, Kubernetes API introduction, in: Kubernetes Programming with Go, Apress, Berkeley, CA., 2023, http://dx.doi.org/10.1007/978-1-4842-9026-2_1.
[21] kubernetes operator patter documentation: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/.
[22] buildah project source code: https://github.com/containers/buildah.
[23] K. Ray, A. Banerjee, N.C. Narendra, Proactive microservice placement and migration for mobile edge computing, 2020 IEEE/ ACM Symp. Edge Comput. (SEC) 2 (2020) 8–41, http://dx.doi.org/10.1109/SEC50012.2020.00010.
[24] F. Garcia, E. Rachelson, Markov decision processes, in: O. Sigaud, O. Buffet (Eds.), Markov Decision Processes in Artificial Intelligence, 2013, http://dx.doi.org/10.1002/9781118557426.ch1, (Chapter 1).
[25] P.S. Junior, D. Miorandi, G. Pierre, Stateful container migration in geo-distributed environments, 2020 IEEE Int. Conf. Cloud Comput. Technol. Sci. (CloudCom) 4 (2020) 9–56, http://dx.doi.org/10.1109/CloudCom49646.2020.00005.
[26] A. Shribman, B. Hudzia, Pre-copy and post-copy VM live migration for memory intensive applications, in: I. Caragiannis, et al. (Eds.), Euro-Par 2012: Parallel Processing Workshops, vol. 7640, Springer, Heidelberg, 2013, pp. 533–542, http://dx.doi.org/10.1007/978-3-642-36949-0_63.
[27] P.S. Junior, D. Miorandi, G. Pierre, Good shepherds care for their cattle: Seamless pod migration in geo-distributed kubernetes, 2022 IEEE 6th Int. Conf. Fog Edge Comput. (ICFEC) 2 (2022) 6–33, http://dx.doi.org/10.1109/ICFEC54809.2022.00011.
[28] J. Ansel, K. Arya, G. Cooperman, DMTCP: Transparent checkpointing for cluster computations and the desktop, in: 2009 IEEE International Symposium on Parallel & Distributed Processing, Rome, Italy, 2009, pp. 1–12, http://dx.doi.org/10.1109/IPDPS.2009.5161063.
[29] M.-N. Tran, X.T. Vu, Y. Kim, Proactive stateful fault-tolerant system for kubernetes containerized services, IEEE Access 10 (2022) 102181–102194, http://dx.doi.org/10.1109/ACCESS.2022.3209257.
[30] H. Schmidt, Z. Rejiba, R. Eidenbenz, K.-T. Förster, Transparent fault tolerance for stateful applications in kubernetes with checkpoint/restore, 2023 42nd Int. Symp. Reliab. Distrib. Syst. (SRDS) 12 (2023) 9–139, http://dx.doi.org/10.1109/SRDS60354.2023.00022.
[31] L.Abdollahi. Vayghan, M.A. Saied, M. Toeroe, F. Khendek, A Kubernetes controller for managing the availability of elastic microservice-based stateful applications, J. Syst. Softw. 175 (2021) 110924, http://dx.doi.org/10.1016/j.jss.2021.110924.
[32] J.-H. Na, et al., PVA: The persistent volume autoscaler for stateful applications in Kubernetes, IEEE Access 12 (2024) 179130–179143, http://dx.doi.org/10.1109/ACCESS.2024.3507194.
[33] P. Bellavista, S. Dahdal, L. Foschini, D. Tazzioli, M. Tortonesi, R. Venanzi, Kubernetes enhanced stateful service migration for ML-driven applications in industry 4.0 scenarios, 2024 IEEE Annu. Congr. Artif. Intell. Things (AIoT) 2 (2024) 5–31, http://dx.doi.org/10.1109/AIoT63253.2024.00015.
[34] P. Bellavista, S. Dahdal, L. Foschini, D. Tazzioli, M. Tortonesi, R. Venanzi, Kubernetes enhanced stateful service migration for ML-driven applications in industry 4.0 scenarios, IEEE Annu. Congr. Artif. Intell. Things (AIoT), Melb. Aust. 2024 (2024) 25–31, http://dx.doi.org/10.1109/AIoT63253.2024.00015.
[35] Martin, P, Extending kubernetes API with custom resources definitions, in: Kubernetes Programming with Go, Apress, Berkeley, CA., 2023, http://dx.doi.org/10.1007/978-1-4842-9026-2_8.