MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

# Improving Checkpoint/Restore Functionality in Kubernetes

Master's Thesis

JAN BRÁZDA

Brno, Spring 2025

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

# Improving Checkpoint/Restore Functionality in Kubernetes

Master's Thesis

JAN BRÁZDA

Advisor: RNDr. Viktória Spišaková

Brno, Spring 2025

FACULTAS ARTIS INFORMATICÆ • UNIVERSITAS MASARYKIANA •

# Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Jan Brázda

**Advisor:** RNDr. Viktória Spišaková

## Acknowledgements

I would like to thank my supervisor, RNDr. Viktória Spišaková, for providing guidance and valuable advice throughout my work on this thesis. Furthermore, my sincere thanks go to Radostin Stoyanov for his helpful consultations and for generously sharing his expertise, which provided valuable perspectives on my work.

# Abstract

This thesis addresses the integration of container checkpoint and restore (C/R) functionality within the Kubernetes environment. It starts off with an examination of the current status of C/R integration, highlighting prevailing limitations and areas for improvement. The core contribution of this work lies in the subsequent design and implementation of targeted enhancements that demonstrably increase the usability of C/R mechanisms in Kubernetes.



# Keywords

Kubernetes, CRIU, Checkpoint/Restore, Containerd

# Contents

**Introduction** **1**

**1 Containers & Kubernetes background** **3**
1.1 Virtualization . . . . . 3
1.2 Introduction to Containers . . . . . 3
1.2.1 Container Image . . . . . 4
1.2.2 Container Runtimes . . . . . 4
1.2.3 Container Orchestration . . . . . 5
1.3 Kubernetes . . . . . 5
1.3.1 Kubernetes Architecture . . . . . 5

**2 Container Checkpoint and Restore** **7**
2.1 Use Cases . . . . . 7
2.1.1 Live Migration . . . . . 7
2.1.2 Dynamic Resource Reallocation . . . . . 8
2.1.3 Fault Tolerance . . . . . 8
2.1.4 Forensic Inspection of Containers . . . . . 9
2.2 CRIU . . . . . 9
2.2.1 Checkpoint Procedure . . . . . 10
2.2.2 Configuring Options . . . . . 10
2.2.3 Encryption . . . . . 11
2.2.4 CRIU Integrations Within Other Tools . . . . . 12
2.3 Checkpoint/Restore in Kubernetes . . . . . 12
2.3.1 History . . . . . 13
2.3.2 Current State of Checkpoint/Restore Integration 14

**3 Implementation** **15**
3.1 API Support for Checkpoint . . . . . 15
3.1.1 Motivation . . . . . 16
3.1.2 Implementation . . . . . 16
3.1.3 HTTP Request Method . . . . . 17
3.1.4 Redirecting Checkpoint Requests to the Kubelet 17
3.1.5 Addressing Request Timeouts for Large Container Checkpoints . . . . . 20
3.2 Controlling Post-Checkpoint Container State . . . . . 20
3.2.1 Motivation . . . . . 21

3.2.2 Adding the `leaveRunning` Parameter to the API 22
3.2.3 Default Behavior . . . . . 22
3.2.4 Interaction with Kubernetes restart policy . . . . 23
3.2.5 Parameter Propagation In The C/R Stack . . . . 25
3.3 Checkpoint Encryption . . . . . 28
3.3.1 Motivation . . . . . 28
3.3.2 Integrating Checkpoint Encryption With Kubernetes Secrets . . . . . 28
3.3.3 Parsing the Certificate in Kubelet . . . . . 30
3.3.4 Updating Container Runtime Interfaces and CRIU RPC . . . . . 31
3.4 Final Request Path and Parameters . . . . . 31

**4 Evaluation** **33**
4.1 Experiment Setup . . . . . 33
4.2 Kubernetes Checkpoint Overhead . . . . . 33
4.3 Effect on Encryption on the Duration of Checkpoint Request . . . . . 37

**5 Conclusion** **39**
5.1 Future work . . . . . 39

**Bibliography** **41**

**A Attachments** **45**

# List of Tables

3.1 Tools, repositories, and branches used for implementation. 15
3.2 Parameters for the checkpoint request. . . . . 32

# List of Figures


2.1 Flow of checkpoint request within Kubernetes. . . . . 13

3.1 Implementation of the `CheckpointLocation` function. . . 19
3.2 Sample python script connecting to MySQL database. . . 29
3.3 Password in checkpoint files. . . . . 30

4.1 Specification of Pod used to perform the experiment. . . . 34
4.2 Checkpoint duration: Kubernetes vs. direct ctr. . . . . 35
4.3 Relative duration of major operations in containerd CheckpointContainer function. . . . . 36
4.4 Impact of encryption on checkpoint duration. . . . . 37

# Introduction

In the evolving landscape of computing, ensuring process continuity, fault tolerance, and resource optimization is critical. Checkpoint and Restore (C/R) is a powerful technique that enables saving the state of a process at a given time (checkpoint) and its subsequent restoration (restore) at a later time or on a different system. This is particularly valuable for stateful applications and long-running computations, where restarting from scratch would be inefficient. This mechanism plays a crucial role in high performance computing (HPC), containerized environments, and failure recovery strategies.

Kubernetes is an open source system for automating the deployment, scaling, and management of containerized applications[1]. Currently, Kubernetes leverages container checkpointing mainly for forensic analysis of containers. The ability to restore from these checkpoints exists, but is not directly integrated within Kubernetes. Broader applications of C/R, such as enabling live container migration are not yet implemented as built-in Kubernetes features.

The goal of this thesis is to enhance the current support of C/R in Kubernetes and implement additional features to improve the utilization of C/R in Kubernetes. To achieve this goal, the thesis will explore the existing mechanisms for C/R in Kubernetes, identify current limitations, and propose enhancements to make C/R more efficient and widely applicable. This involves adding an option to remove the container once checkpoint is finished, or adding support for checkpoint encryption.

The thesis begins (Chapter 1) by laying the groundwork with a review of fundamental concepts in containerization and the Kubernetes orchestration system, providing essential context.

The discussion then transitions (Chapter 2) to the core principles of C/R technology, examining its applications, key tools like CRIU, and the existing landscape of C/R integration within Kubernetes.

Following this exploration, the thesis details (Chapter 3) the specific technical contributions made. These include the architectural shift of checkpoint initiation to the main Kubernetes API, the introduction of granular control over container state post-checkpoint, and

the integration of robust checkpoint encryption based on an existing standalone solution.

The practical implications of these enhancements are then assessed (Chapter 4) through an evaluation, which presents the experimental methodology and analyzes both the performance overhead introduced by the Kubernetes integration and the impact of encryption on checkpoint operations.

Finally, the thesis concludes (Chapter 5) with a summary of the key achievements and insights gained, alongside a discussion of potential avenues for future research and development in this domain.

# 1 Containers & Kubernetes background

In recent times containers have become the preferred choice when it comes to deploying applications. It is easy to see why, as they offer a lightweight, portable, and consistent environment for applications to run across different computing environments. This chapter provides background on what containers are, what advantages they offer and introduces the most popular tool to orchestrate containers, Kubernetes.

## 1.1 Virtualization

Virtualization is a process that allows for more efficient use of physical computer hardware[2]. It achieves this by sharing the host's physical resources such as memory, CPU or storage with multiple virtual units. This allows for more adequate use of resources, since most of applications won't make use of full potential of the hardware all of the time. Virtualized environments are also easier to setup, either from a previously taken snapshot or a recipe in predefined format. Lastly, virtualization increases security by executing an application in sandboxed environment, restricting it from causing harm to physical computers or compromising important data. Different virtualization techniques are defined by how closely they interact with the hardware of the host.

**Virtual machines** (VMs) represent older approach to virtualization. They are managed by software called **hypervisor**. Hypervisor is a layer between the physical hardware and virtual environment. It is responsible for dynamically allocating resources to each virtual machine. By abstracting the underlying hardware, it enables the creation of isolated environments for each VM, making it possible to run different operating systems simultaneously on the host machine.

## 1.2 Introduction to Containers

**Containers** represent newer way of virtualization. Unlike traditional virtual machines, which require a full operating system for each instance, containers share the host OS kernel, which makes them much

lighter and faster than VMs. They package the application along with its dependencies, libraries, and settings into a single unit that can be deployed anywhere, as long as the host system supports the container platform.

With the advent of microservice architecture, containers have become a critical component in modern application development and deployment. Microservices break applications into smaller, independent services that can be developed, deployed, and scaled independently. Containers facilitate this approach by providing lightweight and portable environments that encapsulate application code, dependencies, and configurations.

### 1.2.1 Container Image

A container image is a lightweight, standalone, and executable package that contains everything required to run a piece of software, including the application code, libraries, configuration files, and software dependencies [3]. These images are typically constructed in layers, where each layer builds upon the one beneath it.

### 1.2.2 Container Runtimes

A container runtime is a software program designed to unpack a container image and translate that into a running process on a computer[4]. In order to standardize the behavior of different runtimes, the Open Container Initiative (OCI)[1] developed an Open Container Initiative Runtime Specification [5]. The aim of this specification is to provide a consistent framework for how container images are executed across different platforms. It defines the necessary operations, such as setting up the container's filesystem, configuring namespaces for process isolation, and applying resource limits using control groups (cgroups), ensuring that any runtime conforming to the specification will behave in a predictable manner.

Containerd[6] is the default runtime used in Docker engine[7]. Other OCI-compliant runtimes, such as `CRI-O`, provide similar functionality with varying optimizations for performance and resource usage. These runtimes are widely adopted in container orchestration

1. `https://opencontainers.org/`

systems such as Kubernetes, which leverage the Container Runtime Interface (CRI)[8] to interact with them seamlessly.

### 1.2.3 Container Orchestration

Managing containerized applications at scale can become complex, especially when deployed across distributed environments. Orchestration automates deployment, scaling, and management tasks, ensuring that applications remain highly available and resilient under varying workloads. It also simplifies tasks such as load balancing and service discovery, thus reducing operational overhead and minimizing downtime. As a result, container orchestration has become essential for efficient resource management, with Kubernetes emerging as the most common orchestrator in the industry.

## 1.3 Kubernetes

Kubernetes is an open source container orchestrator released by Google in 2014[9]. Since then, it has become a dominant force in the world of container management, transforming how organizations manage and scale their applications. By decoupling applications from the physical infrastructure, Kubernetes enables seamless scaling across different environments, allowing faster deployment, better scaling, and easier management.

### 1.3.1 Kubernetes Architecture

Kubernetes is a distributed system that consists of a control plane and one or more worker nodes, forming a **cluster**. Each worker node runs the **kubelet**, which is a component responsible for communicating with the control plane and managing containerized workloads. The **control plane** ensures that the cluster is in desired state, scheduling workloads, and maintaining overall system health[10].

#### Control Plane

The control plane is responsible for managing the cluster and consists of the following key components:

- `kube-apiserver`: Acts as the central interface for all operations in the cluster, processing requests from other components in the cluster or directly from user.
- `kube-controller-manager`: Manages controllers that handle various tasks, such as maintaining the desired state of the cluster and managing node health.
- `kube-scheduler`: Assigns workloads (Pods) to suitable worker nodes based on resource availability and other constraints.
- `etcd`: A highly-available key-value store that serves as the primary data store of Kubernetes, storing all of the cluster's data.

**Worker Nodes**

Each worker node runs the necessary services to host and manage containerized applications:

- `kubelet`: An agent on each node that ensures all the containers assigned to this node are running and in healthy state.
- `kube-proxy`: A service responsible for maintaining network rules on each node. This allows communication from within the cluster as well as outside of it.
- `Container runtime`: The underlying software responsible for creating and running containers. Described in Section 1.2.2.

# 2 Container Checkpoint and Restore

**Container checkpoint** is a technique used to save the state of a running container into a group of files. Later on, the files can be used to **restore** the container back to the state when it was saved. This technique is referred to as Checkpoint/Restore (C/R). In this chapter I will explore the use cases for container C/R, the tools currently used to perform it and the current state of C/R when it comes to Kubernetes.

## 2.1 Use Cases

C/R technology unlocks a diverse range of powerful applications. These use cases extend beyond simple backup and recovery, offering sophisticated methods to improve system resilience, optimize resource utilization and other features. The following subsections explore several key applications where C/R plays a pivotal role in modern computing environments.

### 2.1.1 Live Migration

Live migration refers to the process of moving a running virtual machine or application from one host computer to another without causing any noticeable disruption to the end user. In modern virtualization platforms live migration can be achieved with zero downtime. There are several scenarios in which live migration might be utilized, among the most common are scheduled maintenance or improvement of resource utilization.

While the concept of live migration was initially introduced for virtual machines nearly two decades ago [11], their substantial size meant that live migrating VMs was not a frequent operation. This contrasts sharply with containers, which are considerably more lightweight. This makes live migration a more practical and appealing prospect for them.

### 2.1.2 Dynamic Resource Reallocation

Dynamic resource reallocation refers to the ability to optimize system performance by reallocating workloads based on resource availability, demand, and their priority. In modern computing environments, workload resource utilization can fluctuate due to varying user activity, application requirements, or infrastructure constraints. C/R enables dynamic resource reallocation by allowing tasks to be paused, moved, and resumed on nodes with more available resources, thus improving resource utilization.

Imagine a computing cluster composed of both high-performance and lower-performance nodes. When a request for a long-running task arrives but all high-performance nodes are occupied, the scheduler has no choice but to assign the task to a lower-performance node. However, at any moment, tasks on high-performance nodes can be completed, freeing up their resources. In such cases, we can checkpoint the task running on a lower-performance node and restore it on a high-performance node, significantly improving execution speed.

A 2020 paper[12] shows that by using checkpoint and restore techniques, a fair scheduler called $Gandiva_{fair}$ can quickly migrate deep learning jobs across heterogeneous GPU clusters. This dynamic migration minimizes idle GPU time, rebalances workloads with minimal overhead (around 100–250 ms), and enables efficient resource trading, ultimately improving overall cluster utilization and fairness.

### 2.1.3 Fault Tolerance

Another key application of C/R is in enhancing fault tolerance. The running tasks that should be highly available can be periodically checkpointed and once a failure occurs, they can be quickly restored on another node. The checkpoint might additionally serve as a backup.

By integrating C/R into the workflow, systems not only reduce downtime but also minimize the loss of progress in long-running tasks. When a node or container failure is detected, the runtime environment can automatically trigger a restoration process that relaunches the application from the most recent checkpoint, ensuring continuity of service. This rapid recovery mechanism is especially critical in

distributed systems where failures are common and could otherwise lead to significant disruptions.

One paper[13] from 2023 explores a transparent C/R mechanism to enhance fault tolerance for stateful applications. The approach periodically captures the complete application state, allowing for rapid recovery on another node when failures occur. It leverages custom operators to manage checkpointing with minimal changes to the application code. The checkpoint frequency is configurable to balance performance overhead and recovery speed.

### 2.1.4 Forensic Inspection of Containers

Forensic investigations can greatly benefit from using C/R. If a container is suspected of executing suspicious code, it can be checkpointed without affecting its operation. This checkpoint then can be restored in a sandboxed environment and further analyzed.

Forensic inspection served as a pretext to adding container checkpointing to Kubernetes. Container checkpointing was first introduced to Kubernetes in version 1.25 as an alpha feature[14].

A 2024 paper[15] is an excellent example of how checkpoint restore can be leveraged for forensic inspection. The authors demonstrate a technique that uses the checkpoint functionality included in the `runc` container runtime. The work includes a python script which creates a checkpoint of a given container. It then performs a string search on the files representing the memory of the checkpointed container, looking for suspicious text. In addition, it uses YARA[1] rules to detect potential malware.

## 2.2 CRIU

Checkpoint/Restore In Userspace (CRIU)[2] is an open-source tool for enabling C/R functionality on Linux operating systems. It can checkpoint and restore either a running container or an individual application. CRIU provides a command-line interface (CLI) and can also be accessed via RPC or the C API.

1. `https://yara.readthedocs.io/en/stable/writingrules.html`
2. `https://criu.org/`

Unlike earlier C/R implementations that required kernel modifications, CRIU performs most of its functions entirely in user space by leveraging existing kernel interfaces.

### 2.2.1 Checkpoint Procedure

CRIU relies extensively on the `/proc` directory to gather detailed information about the process being checkpointed, such as open file descriptors, memory mappings, and other relevant metadata. After collecting this data, it uses the `ptrace` system call to inject parasite code into the process. This injected code is then responsible for dumping the process state into image files for later restoration.

### 2.2.2 Configuring Options

CRIU can be invoked in two modes: CLI mode and RPC mode. Options can be passed in two ways: directly via the CLI or RPC, or through configuration files[16].

#### CLI Mode

In CLI mode, CRIU employs a hierarchical system for loading configuration files, where settings from higher-priority sources override those from lower-priority ones. This precedence order, from lowest to highest priority, is as follows:

1. **System-wide defaults**: CRIU first checks for a global configuration file at `/etc/criu/default.conf`.
2. **User-specific defaults**: User-specific configuration file located at `~/.criu/default.conf` (within the current user's home directory).
3. **Environment variable**: Configuration file specified via the `CRIU_CONFIG_FILE` environment variable.
4. **Explicit configuration file argument**: Configuration file is explicitly provided using the `--config` command-line argument.
5. **Direct command-line options**: Any options passed directly on the command line

When CRIU is invoked by `runc`, the situation is modified: options originating from `runc` (potentially influenced by `/etc/criu/runc.conf`) take precedence over the standard CRIU configuration files (`default.conf` files and `CRIU_CONFIG_FILE`).

#### RPC Mode

In RPC mode, the order of priority for setting options is similar, with one important distinction: the RPC configuration file has the highest priority, even overriding options passed directly via the RPC call. This has significant implications, particularly in environments like Kubernetes. For example, when CRIU is integrated with Kubernetes, it is typically invoked by `runc` via an RPC call. This means that a Kubernetes administrator could modify the configuration file to enforce specific policies, ensuring that users comply with those settings.

### 2.2.3 Encryption

Protecting the confidentiality of the checkpoint files, which may contain sensitive process memory and state, is an important security consideration.

The only known implementation of encryption for container checkpointing systems, built on top of CRIU, was developed by Radostin Stoyanov[17], available on a dedicated development branch [3]. This approach cleverly repurposes CRIU existing command-line options originally designed for securing network transfers: `--tls-cert` (for the TLS certificate) and `--tls-key` (for the TLS private key).

In this implementation, rather than securing network traffic, these parameters are utilized to encrypt the checkpoint image files directly on the filesystem during the dump operation. The public key information, typically derived from the certificate, is used for encryption. Consequently, the corresponding private TLS key must be provided during the restore operation to decrypt the image files before the process state can be successfully restored.

3. `https://github.com/rst0git/criu/tree/encrypted-images`

### 2.2.4 CRIU Integrations Within Other Tools

CRIU has gained significant popularity and has been integrated into various tools to enhance its functionality. While it can be invoked directly from the command line, this method is limited to single processes and does not support containerized workloads. Below is an example of how to invoke CRIU directly from the command line:

```
criu dump -D <DESTINATION_DIR> -t <PID>
```

`Runc`[4] is a tool that levereges CRIU via its RPC interface to perform C/R operations. It's a low-level container runtime, developed by the Open Container Initiative. Designed as a low-level tool, it is not intended for direct end-user interaction; instead, it serves as a building block for higher-level container management systems.

Containerd is a high-level container runtime that builds upon `runc`. When handling C/R requests for its managed containers, containerd does not interact directly with CRIU. Instead, it constructs and executes the appropriate `runc` command as a subprocess.

The progression of a checkpoint request through the Kubernetes environment, highlighting the interactions between the involved tools, is depicted in Figure 2.1. This figure illustrates the process before the the enhancements developed in this work.

## 2.3 Checkpoint/Restore in Kubernetes

The initial integration of C/R in Kubernetes was introduced as an alpha feature in version 1.25 [14], released in August 2023. The primary motivation for adding C/R to Kubernetes, as stated in the blog post, was to enable forensic inspection of checkpointed containers; however, this feature clearly has many other potential use cases. Initially, C/R was only supported by the CRI-O runtime, but containerd now supports it as well. Since the release of version 1.25, there have been no significant updates to C/R functionality in Kubernetes, aside from the feature gate `ContainerCheckpoint` being promoted to beta status.[5]

4. https://github.com/opencontainers/runc/tree/main
5. https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG/CHANGELOG-1.30.md#feature-8

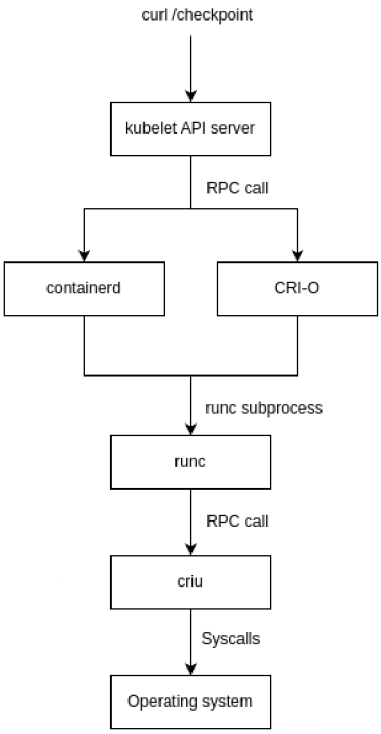


**Figure 2.1:** Flow of checkpoint request within Kubernetes.

### 2.3.1 History

The idea of integrating C/R into Kubernetes was first seriously considered in a GitHub issue created in 2015[6].

However, it wasn't until September 2020 that Adrian Reber created a Kubernetes Enhancement Proposal (KEP) titled "`Add Forensic Container Checkpointing KEP`". A KEP is a structured way to propose and coordinate changes to Kubernetes, and this marked a significant step toward implementing C/R in Kubernetes.

The first implementation of C/R to Kubernetes was a pull request[7] from Adrian Reber, opened in December 2020. The proposal highlighted the need for C/R in Kubernetes to improve the handling of workloads during node drains. Node draining is performed when we want to safely remove all Pods from a Node and reassign them to

6. `https://github.com/kubernetes/kubernetes/issues/3949`
7. `https://github.com/kubernetes/kubernetes/pull/97194`

another one. However, during this process, the Pods are terminated and later restarted. Reber argued that checkpointing and restoring them instead could enhance the handling of stateful applications and those with long startup times. Despite its potential benefits, the pull request was not approved.

The changes proposed in the KEP were later implemented in a separate pull request titled "`Minimal checkpointing support`"[8]. Unlike the previous pull request, which was ultimately not approved, this one focused solely on the checkpointing part, leaving out the restore functionality. The pull request was approved and subsequently merged in July 2022.

### 2.3.2 Current State of Checkpoint/Restore Integration

At the time of writing this thesis, the latest released version of Kubernetes was 1.31. Therefore, any references to the current state of Kubernetes refer to this version.

In Kubernetes, a **Pod** is a group of one or more containers with shared network and storage resources[18]. Although checkpoint requests are directed to a Pod, checkpointing operates at the container level. Therefore, if a Pod contains multiple containers, the specific container to be checkpointed must be identified.

Currently, Kubernetes exposes the checkpoint endpoint only through the kubelet API. Since the kubelet API is not accessible from external networks, creating a checkpoint requires executing a `curl` (or similar) command directly on the node where the container is running. Additionally, the request takes no parameters other than the container to be checkpointed, which means that the behavior of CRIU cannot be modified by passing additional options.

These limitations significantly restrict the practical use of container C/R in Kubernetes. Without the ability to pass custom options and configure checkpoint behavior, it is challenging to optimize C/R processes for different workloads. Expanding the flexibility of the checkpoint endpoint or exposing it through a higher-level Kubernetes API could improve the adoption and effectiveness of C/R in real-world deployments.

8. `https://github.com/kubernetes/kubernetes/pull/104907`

# 3 Implementation

This chapter is divided into sections, with each section describing a specific feature implemented as part of this thesis. Each section will cover the motivation behind the improvement, its practical benefits, and the process of writing the code. Where relevant, the sections will also offer suggestions and inspiration for future work. The software tools employed in this thesis, along with their specific versions or development branches, are detailed in Table 3.1.

The implementation work has been done within containerd runtime and the `runc` tool. This selection is based on their use within the CERIT-SC infrastructure, where the research and development were conducted. While CRI-O is another notable container runtime, it was not the focus of this work.

The source code for this implementation, including all implemented changes, is available online at `https://github.com/hanzbzz/DP`.

**Table 3.1:** Tools, repositories, and branches used for implementation.

| Tool | Repository | Branch |
|---|---|---|
| Kubernetes | `https://github.com/kubernetes/kubernetes` | release-1.31 |
| Containerd | `https://github.com/containerd/containerd` | release/2.0 |
| Criu | `https://github.com/rst0git/criu` | encrypted-images |
| Runc | `https://github.com/opencontainers/runc` | release-1.2 |

## 3.1 API Support for Checkpoint

Creating a checkpoint of a container in Kubernetes involves invoking a call to the kubelet API. The goal here is to extend the Kubernetes API to allow checkpointing to be initiated directly from the main API.

### 3.1.1 Motivation

A major downside of the checkpoint API being located on the kubelet API is clear — the kubelet API is exposed only on the localhost interface and is not accessible from the network. This makes interacting with the API inconvenient, and any automated workflows would struggle to overcome this limitation.

A naive workaround is to use port forwarding to expose the port, but this approach has drawbacks. It can be insecure since the kubelet API is not designed with external exposure in mind. Additionally, relying on port forwarding introduces complexity and potential security risks.

Furthermore, the kubelet API uses a different set of certificates to authenticate users than the main Kubernetes API. In a multi-tenant Kubernetes environment, allowing users to checkpoint containers would require sharing the kubelet certificates with each user. This approach would not be secure and it would complicate user management, which is significantly easier when accessing the main Kubernetes API.

Therefore, extending the checkpoint API to the main Kubernetes API would significantly improve the user experience by providing a more accessible, secure, and centralized interface. It would eliminate the need for complex certificate management while enhancing security and consistency through Kubernetes existing authentication and Role-Based Access Control (RBAC) frameworks.

### 3.1.2 Implementation

To align with best practices established in the Kubernetes codebase, it would be prudent to first investigate whether any existing endpoints from the kubelet API have already been exported to the main Kubernetes API. Fortunately, there is a precedent for this: the `/log` endpoint.

The implementation will also draw inspiration from a previous pull request[19] that addressed container checkpointing during node drain scenarios. Although this request was not accepted, certain aspects remain relevant to our implementation.

The `/log` endpoint is used whenever a request to view the logs of a container in Kubernetes is made. Since container management is handled by the kubelet, the Kubernetes API must forward the request

to the kubelet API, retrieve the response, and then relay it back to the requester. This mechanism demonstrates that it is possible to expose kubelet-managed functionality through the main Kubernetes API, providing a useful model for exporting the checkpoint API in a similar manner.

The first step toward the implementation is adding a new type to the codebase, `ContainerCheckpointOptions`. This type will specify the options accepted by the API server. Initially, the accepted options will include the name of the container to be checkpointed and a timeout argument, specifying the duration (in seconds) allowed for the checkpoint to complete.

### 3.1.3 HTTP Request Method

In the initial implementation of the aforementioned pull request, the main Kubernetes API accepted a POST request, which was then forwarded to the kubelet API as a GET request. However, since the operation involves creating an entity (the tar archive containing the checkpoint) and has the potential to modify the environment (especially with the later implementation of the `leaveRunning` parameter), it is more appropriate for both requests to utilize the POST method. The POST method is typically used for actions that result in the creation of resources or induce changes in the server's state. Therefore, I modified the kubelet API server to accept POST requests as well.

This adjustment allows the Kubernetes API to accept the request body in JSON format, optionally append parameters to it, and then transmit this object directly to the kubelet API without the need to convert it into URL parameters. This approach simplifies the extension of the API with additional options and aligns with best practices in API development.

### 3.1.4 Redirecting Checkpoint Requests to the Kubelet

Once the Kubernetes API server receives a checkpoint request for a Pod, it must forward this request to the kubelet agent running on the node where the target Pod resides. The kubelet is responsible for invoking the actual checkpoint operation via the CRI. Subsequently, the kubelet response is relayed back through the API server to the

initial requester. The API server, therefore, acts as a proxy for these operations.

The checkpoint request URL typically follows this pattern:

```
<KUBERNETES_API_SERVER>/api/v1/namespaces/<POD_NAMESPACE>
/pods/<POD_NAME>/checkpoint
```

From this URL, the API server extracts the target Pod's name and its namespace.

Using this information, the API server retrieves the full Pod object definition. This definition contains the `spec.nodeName` field, which specifies the node currently hosting the Pod. With the target node identified, the API server can proceed to construct and forward the request to that specific node's kubelet API endpoint.

The core logic for handling Pod-related API operations is primarily located within the `k8s.io/kubernetes/pkg/registry/core/pod` package[1]. Specifically, the REST storage implementation found in `storage.go` defines handlers for various Pod endpoints and subresources, including `checkpoint` functionality.

A key function involved in routing the checkpoint request is `CheckpointLocation`, also part of the `pod` package. While the Pod object lookup reveals the target node, the `CheckpointLocation` function determines the specific kubelet URL on that node and identifies the container to be checkpointed. It receives the request details, including `ContainerCheckpointOptions`, which contains the optional target container name provided in the original request. If no container name is specified in the options, `CheckpointLocation` defaults to selecting the first container defined in the Pod's specification (`pod.Spec.Containers[0]`). Figure 3.1 shows a code snippet of the implementation, with the `validateContainer` function responsible for checking the validity of container name provided in the request.

Ultimately, the API server uses the location details returned by `CheckpointLocation` to proxy the checkpoint command to the correct kubelet and container instance.

1. https://github.com/kubernetes/kubernetes/tree/master/pkg/registry/core/pod

```
    pod, err := getPod(ctx, getter, name)
    if err != nil {
        return nil, nil, "", "", err
    }
    // Try to figure out a container
    // If a container was provided, it must be
       valid
    container := opts.Container
    // if pod has only 1 container, the post
       request argument is ignored
    container, err = validateContainer(container
       , pod)
    if err != nil {
        return nil, nil, "", "", err
    }
    nodeName := types.NodeName(pod.Spec.NodeName
       )
    if len(nodeName) == 0 {
        // If pod has not been assigned a host,
           return an empty location
        return nil, nil, container, "", errors.
           NewServiceUnavailable("Pod not
           assigned to host yet")
    }
    nodeInfo, err := connInfo.GetConnectionInfo(
       ctx, nodeName)
    if err != nil {
        return nil, nil, container, "", err
    }
    // create path
    path := fmt.Sprintf("/checkpoint/%s/%s/%s",
       pod.Namespace, pod.Name, container)
    loc := &url.URL{
        Scheme: nodeInfo.Scheme,
        Host:   net.JoinHostPort(nodeInfo.
           Hostname, nodeInfo.Port),
        Path:   path,
    }
    return loc, nodeInfo.Transport, container,
       pod.Namespace, nil
}
```



**Figure 3.1:** Implementation of the `CheckpointLocation` function.

As part of this implementation effort, error handling was also improved. Previously, if a checkpoint request targeted a Pod that had not yet been scheduled to a node (e.g., immediately after creation), the internal lookup could return a `nil` value. This unhandled condition resulted in an unexplained HTTP 500 (Internal Server Error) status code being sent back to the client. This has been addressed to provide a more informative error in such cases. Furthermore, the structure of API responses was standardized; responses now consistently utilize JSON format, and failed requests reliably include a message field containing a clear explanation of the error.

### 3.1.5 Addressing Request Timeouts for Large Container Checkpoints

Experiments involving checkpointing large containers revealed an issue after migrating this functionality to the Kubernetes API: the API server's default request timeout of 34 seconds is frequently too short. The checkpointing process for large containers can extend for several minutes, far exceeding this default limit.

To resolve this limitation, the API server's request handler was modified. The updated handler identifies requests targeting the checkpoint subresource (discernible from the final segment of the request URL). For such requests, the handler overrides the default timeout. It sets the request-specific timeout to the value provided in the `timeout` parameter of the incoming request. If this parameter is not included, the standard 34-second timeout remains in effect.

When checkpointing large containers, it is crucial to specify an adequate `timeout` value in the request. Providing ample time is essential because an unexpected timeout during the checkpoint operation can lead to a non-recoverable interruption, potentially corrupting the checkpoint or leaving the container in an unstable or error state.

## 3.2 Controlling Post-Checkpoint Container State

This section details the behavior of processes after being checkpointed by CRIU and introduces a mechanism within Kubernetes to control this final state. It explores the motivation, implementation details, and

system interactions involved in allowing users to specify whether a container remains running or is terminated post-checkpoint.

### 3.2.1 Motivation

By default, after creating a checkpoint, CRIU terminates the target process (using SIGKILL). However, CRIU provides command-line flags to modify this behavior: **--leave-stopped** and **--leave-running**.

- **--leave-stopped**: To capture a consistent state, CRIU freezes the process first (using the SIGSTOP signal). When **--leave-stopped** is used, CRIU writes the checkpoint image and then leaves the process frozen in the kernel's TASK_STOPPED state. Execution remains suspended, but the process's memory, file descriptors, and other resources are preserved. CRIU simply omits the final signal that would terminate or resume the process.

- **--leave-running**: If **--leave-running** is specified, CRIU attempts to resume the process after saving the checkpoint. Since the process was paused using SIGSTOP, CRIU subsequently sends the SIGCONT signal. This allows the application to continue executing from where it was paused, experiencing only a brief interruption during the checkpoint procedure.

The underlying behavior described above applies equally when checkpointing processes within containers. However, current implementations of container checkpointing features in Kubernetes does not expose control over this final state; the container is implicitly left running after a checkpoint operation.

The objective of the work described here is to expose this choice within Kubernetes, allowing users and administrators to specify whether a container should be left running (similar to **--leave-running**) or terminated (default state if **--leave-running** is omitted) after a checkpoint is created.

Leaving a container exited after checkpointing immediately frees up CPU resources on the node, which is beneficial for resource optimization, especially when the container is being prepared for backup or cold migration. However, leaving the container running is often desired for minimal disruption during live migration or quick snapshots.

### 3.2.2 Adding the `leaveRunning` Parameter to the API

To enable user control over the post-checkpoint state, a new `leaveRunning` parameter is proposed for the Kubernetes container checkpointing API. Implementing this involves modifications across several layers:

- Extending the `ContainerCheckpointOptions` Go structure to include the new `leaveRunning` field.
- Updating associated Go code within Kubernetes components (like the kubelet) that handle checkpoint requests.
- Modifying the gRPC definition located in the proto file[2]

Initially, a parameter named `Exit` was consisdered for this functionality. However, the name was changed to `leaveRunning` to align more closely with the naming conventions used by CRIU.

### 3.2.3 Default Behavior

A key design consideration was the default behavior for the `leaveRunning` parameter, particularly when it is not specified in the API request. CRIU's default behavior (when its equivalent flag is omitted) is to terminate the process after checkpointing. Adopting this default (equivalent to `leaveRunning: false`) would be simpler from an implementation perspective, as omitted boolean fields in gRPC messages typically default to `false`.

However, defaulting to false (terminate/stop) introduces a significant risk: users unfamiliar with the new parameter, or tools not yet updated, could unintentionally stop running containers simply by making a standard checkpoint request without specifying `leaveRunning`. This could lead to unexpected service interruptions or data loss.

To prevent such accidental disruptions, the decision was made to set the default behavior to **true** (leave the container running). This ensures backward compatibility and safety; containers continue to

2. `https://github.com/kubernetes/kubernetes/blob/master/staging/src/k8s.io/cri-api/pkg/apis/runtime/v1/api.proto`

run after checkpointing unless the user explicitly requests them to be stopped by setting `leaveRunning`: false.

Implementing a `true` default for a boolean field presents a challenge in Go and gRPC. A standard `bool` type cannot distinguish between a field explicitly set to `false` by the user and a field that was omitted (which also results in a `false` value after deserialization). To overcome this ambiguity, the `leaveRunning` parameter is defined using a pointer type (`*bool` in Go). This allows the runtime implementation to differentiate the cases:

- If the `leaveRunning` pointer is `nil`, it signifies the field was omitted in the request. The implementation can then apply the desired default behavior (keep the container running, i.e., treat as `true`).
- If the `leaveRunning` pointer is non-`nil`, it signifies the user explicitly provided a value. The implementation then respects the user's choice by using the dereferenced boolean value (`true` or `false`).

### 3.2.4 Interaction with Kubernetes restart policy

The Kubernetes Pod restart policy defines how containers within a Pod should be automatically restarted upon termination. This is a core part of Kubernetes' self-healing capabilities. There are three options: `Always`, `Never`, and `OnFailure`. The `Always` and `Never` options are self-explanatory, while `OnFailure` restarts the container only if it exits with a non-zero status code. Crucially, for standard Deployments[20], the only allowed restart policy is `Always`.

This interaction raises a critical question: how should the `leaveRunning: false` parameter behave with respect to the Pod's restart policy? From Kubernetes' perspective, when a checkpoint is performed and CRIU terminates the main process (as directed by `leaveRunning: false`), the container will exit. If this exit has a non-zero status code (which is common for forced termination), Kubernetes will interpret this as a failure. Consequently, if the Pod's restart policy is `Always` or `OnFailure`, Kubernetes will attempt to restart the container. This behavior directly contradicts the user's explicit request to stop the container via the checkpoint operation.

Several solutions were considered to address this conflict:

1. **Pre-flight validation check:** One potential solution is to validate the checkpoint request against the Pod's restart policy before performing the checkpoint. If a conflict exists (e.g., `leaveRunning: false` is requested for a Pod with `restartPolicy: Always` or `OnFailure`), the API server or kubelet could reject the request, perhaps with a `BadRequest` error.

2. **Bypass restart policy via Pod deletion:** Another option is to bypass the restart policy altogether when `leaveRunning` is set to `false`. This could be implemented by having the checkpoint operation trigger not just the container stop, but the deletion of the entire Pod. This approach grants the user explicit control, ensuring the container and Pod are terminated as requested, irrespective of the restart policy.

3. **Kubelet-level handling:** A potentially more integrated and less disruptive solution involves modifying the kubelet behavior. When a container exits specifically as a result of a checkpoint operation where `leaveRunning: false` was requested, the kubelet could recognize this specific context. Instead of unconditionally applying the Pod's restart policy, the kubelet could be programmed to deliberately ignore the restart policy for this particular termination event.

Evaluating these options, modifying the kubelet (Option 3) offers the cleanest solution but its implementation complexity is out of scope for this thesis. Pre-validating the request (Option 1) is impractical, as it would block `leaveRunning`: false for standard Deployments which require restartPolicy: Always.

Therefore, the second option – deleting the Pod when `leaveRunning: false` is specified – was chosen. While this means losing the specific Pod's history, it directly enforces the user's intent, guaranteeing the container stops and bypasses the restart policy conflict. This pragmatic approach prioritizes the explicit `leaveRunning: false` paramater over standard Pod lifecycle management in this specific context.

### 3.2.5 Parameter Propagation In The C/R Stack

Following the introduction of checkpoint parameters via the Kubernetes API, this section details their propagation path throught the C/R stack, as illustrated in Figure 2.1.

The sequence initiates when the kubelet receives the API request containing the checkpoint parameters. Primary handling occurs within the `checkpoint` function located in kubelet `server.go` source file[3]. After performing initial validation (e.g., verifying Pod and container existence), the kubelet invokes the `CheckpointContainer` function defined by the CRI API. This crucial delegation step means the responsibility for implementing the checkpoint, including interpreting the parameters, shifts to the specific container runtime configured for the cluster, such as containerd or CRI-O.

#### Parameter Propagation Through Containerd

Within containerd, the implementation of the *CheckpointContainer* CRI function receives the parameters originated by the API request. Before initiating the checkpoint, containerd performs prerequisite validation. Notably, it checks if the installed CRIU version meets the minimum requirement (v3.16 or later). This check is necessary because CRIU v3.16 introduced key enhancements, including "better support for restoring containers into existing Pods" as documented in the official CRIU changelog[21].

Containerd's `CheckpointContainer` implementation requires invoking an internal checkpoint API. Two options exist: `task.Checkpoint` and `container.Checkpoint`. While both return the same Image descriptor and error, their suitability differs based on their accepted arguments. Initially, the containerd implementation used the `task.Checkpoint` method.

The key distinction lies in their options structures, since `container.Checkpoint` accepts `CheckpointOptions`, which directly maps to many CRIU parameters. Conversely, `task.Checkpoint` uses `CheckpointTaskOpts`, providing less direct means to control specific CRIU behavior.

---

3. https://github.com/kubernetes/kubernetes/blob/master/pkg/kubelet/server/server.go

Consequently, `container.Checkpoint` is better suited for integrating in Kubernetes, allowing for the propagation of detailed checkpoint configurations. Despite their different signatures at the containerd API level, both methods ultimately delegate the actual checkpoint operation to the same function within the `containerd-shim-runc-v2` shim[22].

To simplify the process of configuring checkpoints, I revised the method for passing options to the `container.Checkpoint` function. Previously, options were passed from components like `CheckpointContainer` using an array of functional options (callbacks) following the *WithCheckpointXXX* naming convention, such as `WithCheckpointTaskExit`[23]. However, this approach had limitations: while suitable for setting static boolean flags (like setting `exit=true` via `WithCheckpointTaskExit`, the fixed function signatures prevented passing variable data required for planned future enhancements.

The solution involves creating the `checkpointOptions` struct within `CheckpointContainer` and populating it with a combination of default values and specific values from the checkpoint request. Consequently, the `container.Checkpoint` function signature was updated to accept an instance of this struct as an additional parameter.

### Parameter Propagation Through Runc

The propagation of checkpoint parameters continues into the `containerd-shim-runc-v2` package, which manages the container using `runc`. Execution converges on the `checkpoint` method associated with the shim's `Init` process[24], representing the container's primary process.

Within the shim, this method translates the received options into a `runc.CheckpointOpts` structure, defined by the `go-runc` library. The shim then calls the `Checkpoint` function from this library[25].

Unlike previous steps using gRPC, the `go-runc` library interacts with `runc` by constructing its command-line arguments. It maps the fields from `runc.CheckpointOpts` – including the boolean value derived from the `leaveRunning` parameter – to corresponding `runc` flags (e.g., `--leave-running`). Finally, `go-runc` executes the generated `runc`

`checkpoint` command, triggering `runc` to perform the checkpoint with the specified options via CRIU.

Therefore, it is necessary to update the `runc.CheckpointOpts` struct to include the newly added options.

### Parameter Propagation Through CRIU

Upon execution, the `runc checkpoint` command begins the process of invoking CRIU. First, `runc` parses its own command-line arguments that were dynamically added based on the parameters propagated from the Kubernetes API request.

These parsed command-line options are then translated internally by `runc` into a Go structure named `libcontainer.CriuOpts`. This structure serves as an intermediary representation of the desired checkpoint configuration. The core logic for handling the checkpoint within `runc` resides in the `Checkpoint` function located in the `libcontainer` package[26].

Before invoking CRIU, this `libcontainer.Checkpoint` function performs several preparatory steps. It verifies that the installed CRIU version meets the minimum requirement (at least v3.0.0). It also prepares the runtime environment for CRIU, which includes tasks like creating the directory where checkpoint image files will be stored and identifying any relevant CRIU configuration files.

Finally, `libcontainer` launches the CRIU executable itself, typically using CRIU's service mode (often referred to as `swrk` mode). In this mode, CRIU starts as a daemon process and listens for commands via RPC over a Unix socket. The `runc` process then sends the checkpoint request, along with the parameters contained within the populated `libcontainer.CriuOpts` structure, to the CRIU daemon via this RPC mechanism.

The handling of `leaveRunning` parameter was already implemented within the RPC handler of CRIU, therefore no changes to the CRIU source code were needed.

## 3.3 Checkpoint Encryption

As detailed in Section 2.2.3, there now exists a version of CRIU that supports checkpoint encryption. This capability allows for the integration of encrypted checkpoints within Kubernetes container management. When this feature is utilized, the resulting checkpoint archive (commonly a tar file) is encrypted using a user-provided key, significantly enhancing the security of the checkpoint data during storage or transmission.

### 3.3.1 Motivation

Unencrypted checkpoints pose a security risk, especially in shared multi-tenant environments. CRIU captures the full memory state of a process, meaning sensitive data like passwords or API keys residing in memory can be embedded within the resulting checkpoint files (typically a tar archive containing CRIU images[27]).

For example, the Python script in Figure 3.2 connects to a database. As shown in Figure 3.3, an unencrypted checkpoint of this running script reveals the database password multiple times within the checkpoint files. Encrypting the checkpoint archive is therefore essential to protect such sensitive information from exposure.

### 3.3.2 Integrating Checkpoint Encryption With Kubernetes Secrets

To utilize CRIU's encryption capability within Kubernetes, a mechanism is needed to provide the necessary public key to CRIU. As established in Section 2.2.3, CRIU accepts an certificate in the X.509 format from which it extracts the encryption key to checkpoint the archive.

Kubernetes offers a standard way to handle TLS certificates and private keys through its **Secret** object, specifically the *kubernetes.io/tls* type. These Secrets store the certificate (containing the public key) and the corresponding private key. Leveraging TLS Secrets is therefore a natural and convenient approach to manage the encryption keys for checkpoints.

The API was updated to accept new parameter, `encryptionSecret`. When requesting a checkpoint via the API, the request specifies the

```python
import mysql.connector
import time

cnx = mysql.connector.connect(
    user='root',
    password='SecretPassword',
    host='mysql',
    database='employees'
)

i = 1
while True:
    print(i)
    time.sleep(1)
    i += 1
```

**Figure 3.2:** Sample python script connecting to MySQL database.

name of the Kubernetes **Secret** via this parameter. The Kubernetes control plane then retrieves the certificate from the Secret, which eventually gets passed to CRIU.

There is one restriction, and that is that the secret used for encryption has to be in the same namespace as the Pod being checkpointed. This makes sense in multi-tenant environments, where usually each tenant is assigned its own namespace.

To implement this, the `ContainerCheckpointOptions` API structure is extended with two fields:

- `encrypt`: A boolean flag to enable encryption.
- `encryptionSecret`: The name of the *kubernetes.io/tls* Secret containing the certificate for encryption.

By default, the `encrypt` parameter is true, ensuring checkpoints are encrypted. This can be overridden by setting `encrypt` to false to disable encryption. In a future version, the `encrypt` parameter may be removed entirely. If this occurs, all checkpoints will be encrypted,

```
> ls
cgroup.img        fdinfo-2.img  ids-1.img         mm-1.img            pagemap-1.img  seccomp.img            timen
s-0.img              tmpfs-dev-593.tar.gz.img  tmpfs-dev-596.tar.gz.img
core-1.img        files.img     inventory.img     mountpoints-13.img  pages-1.img    tcp-stream-13d772.img  tmpfs
-dev-590.tar.gz.img  tmpfs-dev-594.tar.gz.img  utsns-12.img
descriptors.json  fs-1.img      ipcns-var-11.img  netns-10.img        pstree.img     test                   tmpfs
-dev-592.tar.gz.img  tmpfs-dev-595.tar.gz.img
> grep -r SecretPassword -a
pages-1.img:



                                                                                                  f0c0:0&0c60cP:0&0cg0c0:0&0c60cp:0&0c
k0c0:0&0c60c0:0&0c0cp0N!uu0c00!p0N!u0c!0c000c0:0&0c60c0:0&0ch0cA0harsets/SecretPassword@00~0
        *0H00
                0:09g0g#0i00K0
0e1
   0    UUS10U
pages-1.img:0n060`0distutils.50<0"0K0X>b00istutildatabase00&0c0<0"us00tk0&0whichd06"uй"u000"u@00&0c(>0"u0u0"up0"u
05"u000"u@00&0c00*"u0u0"u0<0"u0 /IZ_z50B"ucnxu0B"u0n"u00000000'SecretPassword'000"u0<0"u0004000shim_distutils
0<0"u0f0D000mysqler__"u0<0"u00`0000distutils.core0<0"u0jx00Y00coretlibule
pages-1.img:I0000{]0_distutilsle0<0"u
                                    00v00|00f0&0cdo_override0<0"u00mg00_10setuptoSecretPassword0<0"u0+DT51400arn
_dishostt0<0"u0f0D000nsure_lmysql0<0"u
```

**Figure 3.3:** Password in checkpoint files.

and the option to disable encryption will no longer be available, or can be accessed only with appropriate RBACs.

### Retrieving Kubernetes Secret by Its Name

Retrieving resources in Kubernetes, such as Secrets, necessitates making requests to the API. To enable this for Secrets, the checkpoint handler was updated to hold a reference to a `genericregistry.Store` instance. This store is specifically configured to operate on Secret resources, allowing the handler to fetch any Secret object by its name.

### 3.3.3 Parsing the Certificate in Kubelet

After the Kubernetes API server retrieves and validates the specified TLS Secret (confirming it's of type *kubernetes.io/tls* and exists in the correct namespace), the kubelet prepares this information for CRIU. The API server extracts the public certificate data, typically stored under the *tls.crt* key within the Secret, and includes this raw data in the container checkpoint request sent to the kubelet managing the target Pod.

Since CRIU's encryption mechanism requires the path to a file containing the X.509 certificate rather than the certificate content directly, the kubelet must bridge this gap. Upon receiving the request, the kubelet securely writes the certificate data to a temporary file on its

local filesystem. Critically, the kubelet must ensure this temporary file is reliably deleted immediately after the CRIU checkpoint operation concludes, regardless of success or failure, to maintain security.

### 3.3.4 Updating Container Runtime Interfaces and CRIU RPC

Propagating the encryption request requires updates cascading from the kubelet. The container runtime interface must be extended to accept the encryption flag and the path to the temporary certificate file. The container runtime implementation then passes these parameters down to `runc`.

The primary modification occurs within the RPC communication between `runc` and CRIU. Their shared protocol definition (*rpc.proto*) needs updating to include a boolean encrypt option and a string field designating the certificate file path. Consequently, `runc` must populate these fields when making the checkpoint RPC call. On the receiving end, CRIU's RPC service must be updated to read these new options, enabling its encryption feature and utilizing the specified certificate path when the encrypt flag is set. This ensures the user's encryption request and the associated key material are correctly handled by CRIU during the checkpoint process.

## 3.4 Final Request Path and Parameters

To request a checkpoint for a specific pod, a `POST` request should be made to the following endpoint:

```
https://<APISERVER_HOST>:
6443/api/v1/namespaces/<NAMESPACE>/pods/<POD>/checkpoint
```

In this URL: `<APISERVER_HOST>` represents the hostname or IP address of the Kubernetes API server. `<NAMESPACE>` is the Kubernetes namespace where the target Pod is located. `<POD>` is the name of the specific Pod to be checkpointed.

The parameters that can influence the behavior of the request are summarized in Table 3.2. They should be passed in the JSON format in the request body.

**Table 3.2:** Parameters for the checkpoint request.

| Name | Type | Default |
|---|---|---|
| leaveRunning | bool | true |
| encrypt | bool | true |
| encryptionSecret | string | "" |
| timeout | integer | 34 |
| container | string | "" |

# 4 Evaluation

This chapter details the evaluation process designed to assess the performance and functionality of the newly implemented Kubernetes-integrated checkpointing feature. A primary objective of this chapter is to understand the practical implications of integrating C/R capabilities directly within the Kubernetes ecosystem. Therefore, a key aspect of this evaluation is to compare the performance characteristics of this integrated approach against the baseline performance achieved when using the underlying tools directly.

## 4.1 Experiment Setup

The experiments were conducted on a node running Ubuntu 24.04 LTS, equipped with an Intel Xeon processor and 32 GB of RAM. This node is part of a testing cluster hosted in the center CERIT-SC[1]. Workloads were generated within containers using the `stress-ng` utility[28] to simulate memory-intensive application behavior. The specific versions of the key software components used in this evaluation are detailed in Table 3.1.

## 4.2 Kubernetes Checkpoint Overhead

This experiment aims to quantify the overhead introduced by the Kubernetes environment when initiating a container checkpoint. It is hypothesized that some overhead will exist due to the propagation of the checkpoint request through various Kubernetes and container runtime layers. A key question is whether this overhead is a fixed cost or if it scales with the resource footprint (e.g., memory usage) of the container being checkpointed.

To investigate this, a Pod utilizing a `stress-ng` container image was deployed. This setup allows for precise control over the number of worker processes and their CPU and memory consumption. An example Pod specification, configured to stress memory, is shown in Figure 4.1.

1. `https://docs.cerit.io/en/docs/platform/overview`

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: stress-ng-pod
spec:
  containers:
  - name: stress-ng
    image: ghcr.io/colinianking/stress-ng
    command: ["stress-ng"]
    args: ["--vm", "1", "--vm-bytes", "1G", "--vm-
       keep"]
  restartPolicy: Never
```

**Figure 4.1:** Specification of Pod used to perform the experiment.

The checkpoint performance for this Pod within Kubernetes was compared against checkpointing the same `stress-ng` container directly using the `ctr` command-line tool, which interacts with containerd. While CRIU is the underlying engine, it operates on processes rather than container abstractions directly. Thus, `ctr` provides the closest baseline for container-level checkpointing outside the full Kubernetes control plane, offering a more direct path to CRIU.

To evaluate performance under memory-intensive conditions, workloads were simulated using `stress-ng` configured to allocate varying amounts of RAM: 1 GB, 2 GB, 4 GB, and 8 GB. Figure 4.2 presents a comparison of the average checkpoint duration for these workloads, contrasting execution within a Kubernetes environment against direct `ctr` usage. Each data point represents the average of 5 independent runs.

As observed in Figure 4.2, the checkpoint duration exhibits a linear relationship with the container's RAM usage in both environments. This is an expected behavior, as CRIU spends a significant portion of its execution time dumping memory pages, an operation inherently proportional to the memory size (this can be confirmed by enabling CRIU's verbose logging).

As expected, the size of the generated checkpoint archives was primarily determined by the memory footprint of the containerized ap-

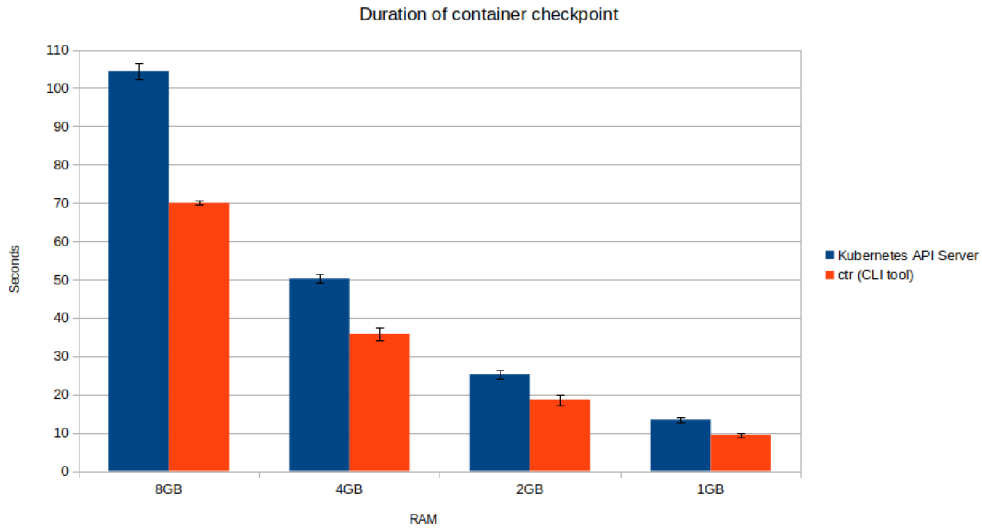


**Figure 4.2:** Checkpoint duration: Kubernetes vs. direct ctr.

plication. For example, containers running stressors that allocated and actively used 4 GB of RAM resulted in checkpoint archives measuring approximately 4.1 GB. This indicates that the archive size is largely composed of the dumped memory pages, with a minor overhead for additional metadata, process state, and file system information captured by CRIU.

Comparing the durations, checkpointing via Kubernetes consistently incurs a higher cost, approximately 30% longer than using `ctr` directly. This consistent percentage increase, scaling with the checkpoint task's complexity (memory size), suggests that the overhead is not merely a fixed latency from request propagation through Kubernetes components. If it were a fixed propagation delay, its relative impact would decrease for larger checkpoints.

To identify the source of this scaling overhead, the containerd component responsible for handling checkpoint requests from the CRI was investigated. Specifically, the **CheckpointContainer** function within `container_checkpoint_linux.go` file was profiled by instrumenting it to log timestamps before and after major internal function calls. Figure 4.3 presents a breakdown of the relative time spent in these

key operations within the **CheckpointContainer** function during a checkpoint.

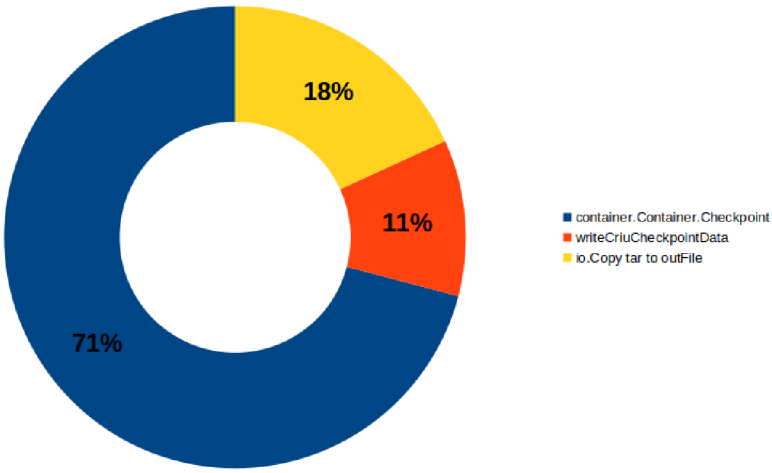


**Figure 4.3:** Relative duration of major operations in containerd CheckpointContainer function.

The profiling results, depicted in Figure 4.3, reveal that while the call to **container.Container.Checkpoint** (which delegates to `runc` and subsequently CRIU) accounts for the largest portion of time, two other operations contribute significantly to the total duration.

The `writeCriuCheckpointData` function parses the checkpoint image object returned by CRIU (via `runc` and the **container.Container.Checkpoint** interface) and writes its constituent files (memory dumps, page maps, etc.) to the filesystem.

A subsequent step involves packaging these checkpoint image files into a tar archive and copying it to the designated checkpoint location.

The ctr tool, when performing a checkpoint, typically creates the checkpoint image directly, without the intermediate filesystem dump and subsequent tarballing steps that containerd CRI implementation performs. Therefore, the additional 30% overhead observed when checkpointing through Kubernetes can be primarily attributed to these extra data handling steps within containerd: the explicit dumping of CRIU image files to an intermediate filesystem location and their subsequent aggregation into a tar archive.

## 4.3 Effect on Encryption on the Duration of Checkpoint Request

Another aspect of performance under investigation is the impact of encrypting the checkpoint image on the overall checkpoint duration. Encryption capabilities of CRIU were introduced in 2024 paper[17] which also provided initial overhead measurements. This section aims to quantify this overhead within the context of our Kubernetes-integrated checkpointing feature, assuming it leverages CRIU's native encryption.

The experimental setup mirrors that described in Section 4.2, focusing on memory-intensive workloads consuming 1 GB, 2 GB, 4 GB, and 8 GB of RAM. Checkpoints were performed using the Kubernetes integration, once with encryption enabled (utilizing the newly implemented functionality) and once without.

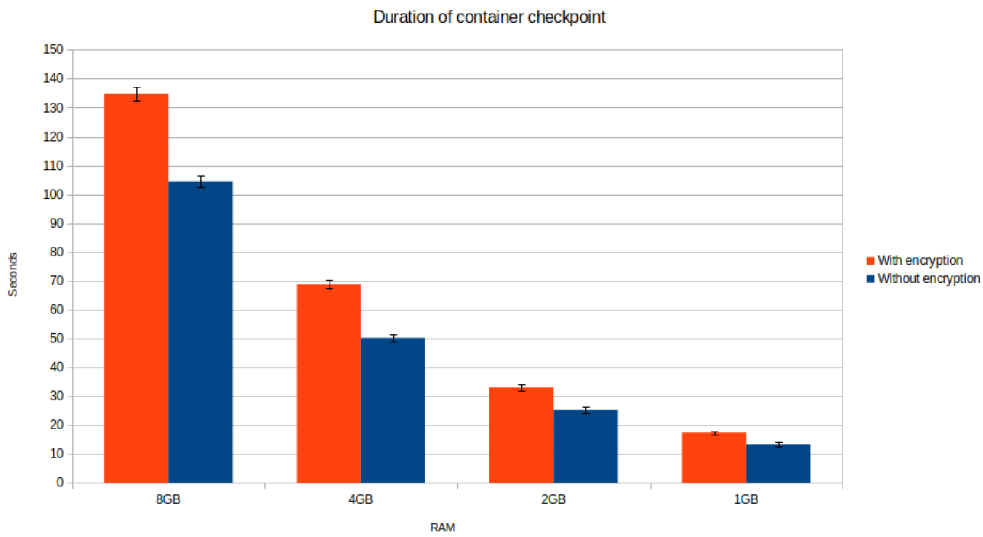


**Figure 4.4:** Impact of encryption on checkpoint duration.

The results presented in Figure 4.4 demonstrate that enabling encryption introduces a considerable performance overhead. On average, encrypted checkpoints took approximately 25% longer to complete compared to their unencrypted counterparts across the tested memory sizes. This overhead is attributed to the computational cost of

cryptographic operations performed by CRIU on the memory pages and other checkpoint image files during the dump process. The linear increase in absolute overhead with memory size is consistent with encrypting a larger volume of data.

# 5 Conclusion

This thesis aimed to enhance the integration of Checkpoint/Restore in Kubernetes by addressing its existing limitations and unlocking new capabilities for more effective container management. Although basic checkpointing functionality was previously available, its reliance on the node-local kubelet API and the absence of configuration options significantly limited its practical usefulness.

To address these challenges, this work introduced several key improvements. One of the most significant contributions was the migration of the checkpoint initiation mechanism from the limited kubelet API to the central Kubernetes API server. This shift allows checkpoint operations to be managed using standard Kubernetes tooling.

Additionally, this work introduced the ability to control the container's state after a checkpoint is taken. Users can now choose whether a container should continue running or be terminated post-checkpoint, providing essential flexibility for various C/R workflows. To further strengthen production readiness, support for checkpoint encryption was also implemented. This enhancement protects potentially sensitive data within checkpoint images, addressing a critical security concern.

The evaluation demonstrated the practicality of these enhancements. While the Kubernetes integration layer introduces an overhead of approximately 30% compared to direct interaction with the container runtime (`ctr`), this increase is primarily due to data handling within containerd CRI implementation and might be different for other container engines. The addition of encryption support further increased checkpoint duration by around 25%, reflecting the expected cost of cryptographic operations in exchange for improved security.

## 5.1 Future work

Building upon the enhancements introduced in this thesis, future work should focus on implementing comprehensive restore functionality at the Kubernetes API level. This advancement would enable seamless restoration of containers from checkpoint images, completing the cycle of Checkpoint/Restore.

To enhance usability, integrating direct support for checkpoint and restore operations within kubectl is essential. This integration would streamline user workflows by eliminating the need for low-level tools or manual interactions with the kubelet API.

Further refinement of pod lifecycle management post-checkpoint is also crucial. Developing more sophisticated mechanisms for graceful termination and automated resource cleanup would ensure smoother transitions and better resource utilization when containers are terminated after checkpointing.

Future work could enhance checkpoint integration security by updating the encryption mechanism to incorporate RBAC. This would ensure that only administrators are authorized to disable checkpoint encryption.

# Bibliography


1. *Kubernetes* [online]. Brno [visited on 2025-02-13]. Available from: https://kubernetes.io/.
2. *What is virtualization?* [online]. Brno [visited on 2025-02-19]. Available from: https://www.ibm.com/think/topics/virtualization/.
3. *Images* [online]. Brno [visited on 2025-02-20]. Available from: https://kubernetes.io/docs/concepts/containers/images/.
4. *Container Runtimes Explained* [online]. Brno [visited on 2025-02-18]. Available from: https://www.upwind.io/glossary/container-runtimes-explained.
5. *Open Container Initiative Runtime Specification* [online]. Brno [visited on 2025-02-18]. Available from: https://github.com/opencontainers/runtime-spec/blob/main/spec.md.
6. *containerd* [online]. Brno [visited on 2025-05-16]. Available from: https://github.com/containerd/containerd.
7. *Alternative container runtimes* [online]. Brno [visited on 2025-02-18]. Available from: https://docs.docker.com/engine/daemon/alternative-runtimes/.
8. *Container Runtime Interface (CRI)* [online]. Brno [visited on 2025-02-18]. Available from: https://kubernetes.io/docs/concepts/architecture/cri/.
9. *An update on container support on Google Cloud Platform* [online]. Brno [visited on 2025-02-20]. Available from: https://cloudplatform.googleblog.com/2014/06/an-update-on-container-support-on-google-cloud-platform.html.
10. *Cluster Architecture* [online]. Brno [visited on 2025-02-20]. Available from: https://kubernetes.io/docs/concepts/architecture/.

11. CLARK, Christopher; FRASER, Keir; HAND, Steven; HANSEN, Jacob Gorm; JUL, Eric; LIMPACH, Christian; PRATT, Ian; WARFIELD, Andrew. Live migration of virtual machines. In: *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. 2005, pp. 273–286.

12. CHAUDHARY, Shubham; RAMJEE, Ramachandran; SIVATHANU, Muthian; KWATRA, Nipun; VISWANATHA, Srinidhi. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–16.

13. SCHMIDT, Henri; REJIBA, Zeineb; EIDENBENZ, Raphael; FÖRSTER, Klaus-Tycho. Transparent Fault Tolerance for Stateful Applications in Kubernetes with Checkpoint/Restore. In: *2023 42nd International Symposium on Reliable Distributed Systems (SRDS)*. 2023, pp. 129–139. Available from DOI: 10.1109/SRDS60354.2023.00022.

14. REBER, Adrian. *Forensic container checkpointing in Kubernetes* [online]. Brno [visited on 2025-03-06]. Available from: https://kubernetes.io/blog/2022/12/05/forensic-container-checkpointing-alpha/.

15. GHARAIBEH, Taha; SEIDEN, Steven; ABOUELSAOUD, Mohamed; BOU-HARB, Elias; BAGGILI, Ibrahim. Don't, Stop, Drop, Pause: Forensics of CONtainer CheckPOINTs (ConPoint). In: *Proceedings of the 19th International Conference on Availability, Reliability and Security*. 2024, pp. 1–11.

16. *Configuration files* [online]. Brno [visited on 2025-03-16]. Available from: https://criu.org/Configuration_files.

17. STOYANOV, Radostin; REBER, Adrian; UENO, Daiki; CLAPIŃSKI, Michał; VAGIN, Andrei; BRUNO, Rodrigo. Towards Efficient End-to-End Encryption for Container Checkpointing Systems. In: *Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems*. Kyoto, Japan: Association for Computing Machinery, 2024, pp. 60–66. APSys '24. ISBN 9798400711053. Available from DOI: 10.1145/3678015.3680477.

18. *Pods* [online]. Brno [visited on 2025-05-17]. Available from: https://kubernetes.io/docs/concepts/workloads/pods/.

19. [*WIP*] *Add –checkpoint to drain* [online]. Brno [visited on 2025-05-18]. Available from: https://github.com/kubernetes/kubernetes/pull/97194.

20. *Deployments* [online]. Brno [visited on 2025-05-18]. Available from: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.

21. *Changelogs* [online]. Brno [visited on 2025-04-20]. Available from: https://criu.org/Changelogs.

22. *containerd/cmd/containerd-shim-runc-v2/runc/container.go at main · containerd/containerd* [online]. Brno [visited on 2025-05-18]. Available from: https://github.com/containerd/containerd/blob/main/cmd/containerd-shim-runc-v2/runc/container.go#L441.

23. *containerd/client/container_checkpoint_opts.go at main · containerd/containerd* [online]. Brno [visited on 2025-05-18]. Available from: https://github.com/containerd/containerd/blob/main/client/container_checkpoint_opts.go#L142.

24. *containerd/cmd/containerd-shim-runc-v2/process/init.go at main · containerd/containerd* [online]. Brno [visited on 2025-05-18]. Available from: https://github.com/containerd/containerd/blob/main/cmd/containerd-shim-runc-v2/process/init.go#L425.

25. *go-runc/runc.go at main · containerd/go-runc* [online]. Brno [visited on 2025-05-18]. Available from: https://github.com/containerd/go-runc/blob/main/runc.go#L598.

26. *runc/libcontainer/criu_linux.go at main · opencontainers/runc* [online]. Brno [visited on 2025-05-18]. Available from: https://github.com/opencontainers/runc/blob/main/libcontainer/criu_linux.go#L280.

27. *Images* [online]. Brno [visited on 2025-05-18]. Available from: https://criu.org/Images.

28. *ColinIanKing/stress-ng: This is the stress-ng upstream project git repository. stress-ng will stress test a computer system in various selectable ways. It was designed to exercise various physical subsystems of a computer as well as the various operating system kernel interfaces.* [online]. Brno [visited on 2025-05-18]. Available from: `https://github.com/ColinIanKing/stress-ng`.

# A Attachments

The `attachment.zip` archive, included with this thesis, contains all source code and configuration files which were part of this work. The top-level files and directories within this archive are detailed below:

- `build_kind_cluster.sh`: A shell script to automate the setup and configuration of a local Kubernetes cluster using Kind (Kubernetes in Docker).
- `counter-container/`: Directory containing the Dockerfile and source code for a sample counter application.
- `db-conn-container/`: Directory containing the Dockerfile and source code for a containerized application that demonstrates checkpoint of an application with an active database connection.
- `kube-deployments/`: Contains Kubernetes sample YAML manifest files.
- `README.md`: A Markdown file providing setup instructions.
- `containerd/`: Containerd source code.
- `criu/`: CRIU source code.
- `kind/`: Kind source code.
- `kubernetes/`: Kubernetes source code.
- `runc/`: Runc source code.