UNIVERSIDADE DO ESTADO DE SANTA CATARINA – UDESC
CENTRO DE CIÊNCIAS TECNOLÓGICAS – CCT
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA – PPGEEL

KELVIN CÉSAR DE ANDRADE

ARQUITETURA PARA MIGRAÇÃO DE APLICAÇÕES COM PRESERVAÇÃO DE ESTADO EM AMBIENTES EMBARCADOS

JOINVILLE
2026

KELVIN CÉSAR DE ANDRADE

ARQUITETURA PARA MIGRAÇÃO DE APLICAÇÕES COM PRESERVAÇÃO DE ESTADO EM AMBIENTES EMBARCADOS

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica do Centro de Ciências Tecnológicas da Universidade do Estado de Santa Catarina, como requisito parcial para a obtenção do grau de Mestre em Engenharia Elétrica.

Orientador: Prof. Dr. Douglas Wildgrube Bertol

JOINVILLE
2026

Ficha catalográfica elaborada pelo programa de geração automática da
Biblioteca Universitária Udesc,
com os dados fornecidos pelo(a) autor(a)

Andrade, Kelvin César de
Arquitetura para migração de aplicações com preservação de estado em ambientes embarcados / Kelvin César de Andrade. -- Joinville, 2026.
91 p.

Orientador: Prof. Dr. Douglas Wildgrube Bertol.
Dissertação (Mestrado) -- Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Engenharia Elétrica, Joinville, 2026.

1. Migração de aplicações. 2. Preservação de estado. 3. CRIU. 4. Dispositivos embarcados. 5. K3s. I. Wildgrube Bertol, Douglas. II. Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Engenharia Elétrica. III. Título.

KELVIN CÉSAR DE ANDRADE

ARQUITETURA PARA MIGRAÇÃO DE APLICAÇÕES COM PRESERVAÇÃO DE ESTADO EM AMBIENTES EMBARCADOS

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica do Centro de Ciências Tecnológicas da Universidade do Estado de Santa Catarina, como requisito parcial para a obtenção do grau de Mestre em Engenharia Elétrica.

Orientador: Prof. Dr. Douglas Wildgrube Bertol

BANCA EXAMINADORA:

Dr. Douglas Wildgrube Bertol
Universidade do Estado de Santa Catarina (UDESC)

Membros:

Dr. Hylson Vescovi Netto
Instituto Federal Catarinense (IFC)

Dr. Rafael Rodrigues Obelheiro
Universidade do Estado de Santa Catarina (UDESC)

Joinville, 26 de agosto de 2026

Aos meus pais, pelas bases do que sou hoje.
E à minha esposa, por todo o apoio e
companheirismo durante essa jornada.

AGRADECIMENTOS

Agradeço aos meus pais, Julio e Susana, por sempre me incentivarem a estudar e por todo o suporte que me deram ao longo do meu desenvolvimento.

À minha esposa, Sabrina, pela paciência, incentivo e compreensão em todos os momentos em que precisei me ausentar para me dedicar a esta pesquisa.

A todos os amigos e colegas que conheci durante essa jornada.

Ao meu orientador, Prof. Dr. Douglas Wildgrube Bertol, pela confiança depositada em mim ao aceitar me orientar. Agradeço também pelo apoio durante todo o processo de construção deste projeto de pesquisa, inclusive no início, quando ainda não sabíamos como resolver o problema que tínhamos em mãos.

Por fim, agradeço à Universidade do Estado de Santa Catarina (UDESC) e aos membros da banca avaliadora pelas contribuições a este trabalho.

"Aquilo que alguém *tem em si mesmo* é, pois, o mais essencial para uma vida feliz."

(Schopenhauer, 2014, p. 18).

RESUMO

A orquestração de contêineres em sistemas ciberfísicos esbarra na dificuldade de migrar aplicações com estado, que não toleram reinicializações sem perda de contexto operacional. Para endereçar essa restrição, este trabalho propõe o ShiftPod, uma arquitetura de migração transparente para a aplicação e integrada de forma nativa ao plano de controle do Kubernetes. O sistema combina um shim customizado e um operador distribuído para coordenar a captura, transferência e restauração da aplicação em execução por meio do CRIU. Avaliada em um cluster de dispositivos embarcados, a solução permitiu mapear o tempo de indisponibilidade sob diferentes cargas de memória e cenários. Os resultados demonstram que a arquitetura viabiliza a mobilidade de estado em hardware de baixo poder computacional, alcançando tempos médios de interrupção entre 1,3 e 3,3 segundos, além de mostrar que, acima de certo tamanho do estado da aplicação, migrar no mesmo nó passa a ser mais rápido que migrar entre nós distintos. O estudo atesta a viabilidade do mecanismo para aplicações ciberfísicas tolerantes a curtas indisponibilidades.

Palavras-chave: Migração de aplicações; Preservação de estado; CRIU; Dispositivos embarcados; K3s.

ABSTRACT

Container orchestration in cyber-physical systems is hindered by the difficulty of migrating stateful applications, which cannot tolerate restarts without the loss of operational context. To address this limitation, this work proposes ShiftPod, an application-transparent migration architecture natively integrated into the Kubernetes control plane. The system combines a custom shim and a distributed operator to coordinate the capture, transfer, and restoration of the execution state via CRIU. Evaluated on a cluster of embedded devices, the solution enabled the mapping of downtime across different memory workloads and scenarios. The results demonstrate that the architecture makes state mobility feasible on resource-constrained hardware, achieving average interruption times between 1.3 and 3.3 seconds, while also showing that, above a certain application state size, migrating within the same node becomes faster than migrating across distinct nodes. This study attests to the viability of the mechanism for cyber-physical applications tolerant to brief interruptions.

Keywords: Application migration; State preservation; CRIU; Embedded devices; K3s.

LISTA DE ILUSTRAÇÕES

Figura 1 – Arquitetura do Kubernetes 24
Figura 2 – Interação entre os componentes do Kubernetes no uso do containerd como *runtime* padrão 27
Figura 3 – Processo de *checkpoint* e *restore* com CRIU 30
Figura 4 – Arquitetura detalhada e fluxo de interação entre os componentes de migração 44
Figura 5 – Diagrama da máquina de estados do processo de migração 45
Figura 6 – Estrutura de integração do shim do ShiftPod com o containerd 48
Figura 7 – Fluxo do processo de *checkpoint* no nó fonte, do ponto de vista do shim . . 53
Figura 8 – Fluxo do processo de restauração no nó de destino, do ponto de vista do shim 54
Figura 9 – Coordenação do manager no nó fonte durante o *checkpoint* 59
Figura 10 – Coordenação do manager no nó destino durante a restauração 60
Figura 11 – Pipeline de transmissão em fluxo contínuo do *checkpoint* entre os nós 61
Figura 12 – Topologia de conectividade do ambiente experimental 63
Figura 13 – Linha do tempo das métricas de temporização durante uma migração 66
Figura 14 – Indisponibilidade t_{da} na migração entre nós distintos. Cada caixa agrega entre 25 e 30 amostras (Seção 6.4), com compressão habilitada e padrões linear e aleatório. 70
Figura 15 – Decomposição da indisponibilidade por fase nos cenários entre nós, compressão habilitada e padrão linear. Barras representam a média das 30 amostras e somam t_{di}, cujo intervalo de confiança de 95% da média é indicado pela barra de erro no topo. 71
Figura 16 – Redução do tamanho do arquivo de *checkpoint* com zstd L1 em função do tamanho de memória, por padrão de conteúdo 72
Figura 17 – Duração da transferência de rede em função do tamanho de memória, para os cenários entre nós. As bandas indicam intervalos de confiança de 95%. . . . 73
Figura 18 – Indisponibilidade t_{da} na migração no mesmo nó. Cada caixa agrega entre 25 e 30 amostras (Seção 6.4), com compressão habilitada. 75
Figura 19 – Decomposição da indisponibilidade por fase nos cenários de mesmo nó, compressão habilitada e padrão linear. Barras representam a média das 30 amostras e somam t_{di}, cujo intervalo de confiança de 95% da média, calculado com distribuição t de Student, é indicado pela barra de erro no topo. 75
Figura 20 – Indisponibilidade t_{da} por tipo de migração em função do tamanho de memória, com compressão habilitada e padrão linear 77
Figura 21 – Distribuição de t_{da} por direção de migração entre nós, compressão habilitada e desabilitada, após a substituição do cartão SD do N_1 80

LISTA DE TABELAS

Tabela 1 – Campos de especificação do recurso ContainerMigration 56
Tabela 2 – Fases do ciclo de vida do recurso ContainerMigration 56
Tabela 3 – Métricas de status do recurso ContainerMigration 57
Tabela 4 – Mensagens do serviço gRPC do manager 58
Tabela 5 – Especificações do ambiente experimental 63
Tabela 6 – Resumo da matriz de experimentos 67
Tabela 7 – Resultados da calibração de `CHUNK_SIZE_KB`, sem compressão, 256 MB, 10 amostras 68
Tabela 8 – Decomposição da indisponibilidade por fase nos cenários entre nós, compressão habilitada, padrão linear. As quatro fases somam t_{di} e agregam 30 amostras por linha. O t_{da} é a indisponibilidade percebida pela aplicação e agrega n amostras. 71
Tabela 9 – Efeito da compressão para 256 MB. Médias de 28 a 30 amostras por cenário. 74
Tabela 10 – Decomposição da indisponibilidade por fase nos cenários de mesmo nó, compressão habilitada, padrão linear. As três fases somam t_{di} e agregam 30 amostras por linha. O t_{da} é a indisponibilidade percebida pela aplicação e agrega n amostras. 76
Tabela 11 – Utilização de *Central Processing Unit* (CPU) do conjunto de nós durante as janelas de migração, por cenário 76
Tabela 12 – Medianas de t_{da} por direção de migração entre nós, agregadas por configuração 78
Tabela 13 – Tratamento das chamadas do serviço de tarefas pelo shim do ShiftPod . . . 91

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
ARM	*Advanced RISC Machine*
CNCF	*Cloud Native Computing Foundation*
CNI	*Container Networking Interface*
CPU	*Central Processing Unit*
CRD	*Custom Resource Definition*
CRI	*Container Runtime Interface*
CRIU	*Checkpoint/Restore In Userspace*
eMMC	*embedded MultiMediaCard*
gRPC	*Google Remote Procedure Call*
HPC	*High Performance Computing*
GPU	*Graphics Processing Unit*
HTTP	*Hypertext Transfer Protocol*
IHM	Interface Homem-Máquina
I/O	*Input/Output*
IoT	Internet das Coisas
IP	*Internet Protocol*
JSON	*JavaScript Object Notation*
NTP	*Network Time Protocol*
NVMe	*Non-Volatile Memory Express*
OCI	*Open Container Initiative*
OPC UA	*OPC Unified Architecture*
PCIe	*Peripheral Component Interconnect Express*
PID	*Process Identifier*
PLC	Controlador Lógico Programável
RAM	*Random Access Memory*
RPC	*Remote Procedure Call*
SSH	*Secure Shell*
STDERR	*Standard Error*
STDIN	*Standard Input*

STDOUT	*Standard Output*
TCP	*Transmission Control Protocol*
TI	Tecnologia da Informação
TO	Tecnologia Operacional
ttrpc	*Tiny Transport Remote Procedure Call*
UID	*User Identifier*
UTS	*Unix Time-Sharing System*
vPLC	Controlador Lógico Programável Virtual

LISTA DE SÍMBOLOS

%	Por cento
GB	Gigabyte
GHz	Gigahertz
KB	Kilobyte
MB	Megabyte
MB/s	Megabytes por segundo
ms	Milissegundo
s	Segundo
s_{comp}	Tamanho do artefato de checkpoint após compressão
s_{raw}	Tamanho do artefato de checkpoint sem compressão
t_{da}	Duração de indisponibilidade percebida pela aplicação de benchmark
$t_{\text{checkpoint}}$	Duração do checkpoint CRIU (congelamento e escrita em disco)
t_{di}	Duração da janela de migração medida pela infraestrutura, do congelamento à retomada do contêiner
t_{k3s}	Parcela residual da janela de migração não atribuível às fases medidas
t_{restore}	Duração da restauração CRIU no nó destino
t_{transfer}	Duração da transferência do artefato entre nós

SUMÁRIO

1 INTRODUÇÃO 17
1.1 PROBLEMA DE PESQUISA 18
1.2 OBJETIVOS 18
1.2.1 Objetivos específicos 19
1.3 JUSTIFICATIVA 19
1.4 ORGANIZAÇÃO DO TRABALHO 19
2 CONCEITOS FUNDAMENTAIS 21
2.1 ARQUITETURA DO KERNEL LINUX 21
2.1.1 Espaço de usuário e espaço do *kernel* 21
2.1.2 Criação de processos: fork e exec 21
2.1.3 Grupos de controle (*cgroups*) 22
2.1.4 Namespaces 22
2.2 COMPUTAÇÃO EM CONTÊINERES 23
2.3 KUBERNETES 23
2.3.1 Arquitetura 23
2.3.2 Abstrações 25
2.3.3 Operadores 26
2.3.4 Arquitetura de execução dos contêineres 26
2.4 K3S 28
2.5 SERVIÇOS COM ESTADO E SEM ESTADO 28
2.6 CHECKPOINT/RESTORE IN USERSPACE (CRIU) 29
2.6.1 Modos de migração suportados pelo CRIU 30
2.7 FERRAMENTAS DE DESENVOLVIMENTO DE SISTEMAS 31
2.7.1 Linguagens de programação 31
2.8 PROTOCOLOS DE CHAMADA REMOTA DE PROCEDIMENTO 32
2.8.1 gRPC 32
2.8.2 ttrpc 32
2.9 SÍNTESE DO CAPÍTULO 32
3 ESTADO DA ARTE 34
3.1 CHECKPOINT E RESTAURAÇÃO DE CONTÊINERES 34
3.2 MIGRAÇÃO DE CONTÊINERES EM AMBIENTES DE BORDA 35
3.3 MIGRAÇÃO COM ESTADO EM KUBERNETES 36
3.4 CONTROLADORES INDUSTRIAIS CONTEINERIZADOS 37
3.5 COMPARAÇÃO E POSICIONAMENTO 38
3.6 SÍNTESE DO CAPÍTULO 39

4 MODELAGEM E ARQUITETURA PROPOSTA 40
4.1 PRINCÍPIOS DE PROJETO 40
4.2 COMPONENTES DA ARQUITETURA 41
4.2.1 Plataforma de orquestração (k3s) 41
4.2.2 Mecanismo de preservação de estado (CRIU) 41
4.2.3 Manager do ShiftPod 42
4.2.4 Shim do ShiftPod 42
4.3 FLUXO DE MIGRAÇÃO DE CARGAS DE TRABALHO 42
4.3.1 Máquina de estados da migração 43
4.4 RESTRIÇÕES TÉCNICAS E PRESSUPOSTOS 45
4.5 SÍNTESE DO CAPÍTULO 46
5 DESENVOLVIMENTO 47
5.1 IMPLEMENTAÇÃO DO SHIM 47
5.1.1 Integração com o ecossistema do containerd 48
5.1.2 Registro do shim no containerd 49
5.1.3 Estado em memória 50
5.1.4 Processo de *checkpoint* 51
5.1.5 Processo de restauração 52
5.1.6 Configuração e habilitação da migração 54
5.2 IMPLEMENTAÇÃO DO MANAGER E DO CRD 55
5.2.1 Definição do recurso customizado (CRD) 55
5.2.2 Interface de comunicação 57
5.2.3 Coordenação da migração pelo manager 58
5.2.4 Transferência de *checkpoint* entre nós 61
6 TESTES E RESULTADOS 62
6.1 CONFIGURAÇÃO DO AMBIENTE EXPERIMENTAL 62
6.2 APLICAÇÃO DE BENCHMARK 63
6.3 INSTRUMENTAÇÃO E MÉTRICAS COLETADAS 64
6.4 DEFINIÇÃO DA MATRIZ DE EXPERIMENTOS 66
6.5 CALIBRAÇÃO DO TAMANHO DE BLOCO DE TRANSMISSÃO 68
6.6 MIGRAÇÃO ENTRE NÓS DISTINTOS 69
6.6.1 Indisponibilidade total observada da aplicação 69
6.6.2 Decomposição por fase 70
6.6.3 Efeito da compressão e tempo de transferência 72
6.7 MIGRAÇÃO NO MESMO NÓ 74
6.7.1 Indisponibilidade e decomposição por fase 74
6.7.2 Comparação com migração entre nós 77

6.8 ASSIMETRIA ENTRE DIREÇÕES **78**
6.9 DISCUSSÃO DOS RESULTADOS **80**
7 CONCLUSÃO 82
7.1 SÍNTESE DAS CONTRIBUIÇÕES **82**
7.2 PRINCIPAIS RESULTADOS **82**
7.3 LIMITAÇÕES **83**
7.4 TRABALHOS FUTUROS **83**
7.5 DECLARAÇÃO DE USO DE INTELIGÊNCIA ARTIFICIAL GENERATIVA **84**
REFERÊNCIAS 85
GLOSSÁRIO 89
APÊNDICE A – EXEMPLOS DE RECURSOS DO SHIFTPOD 90
APÊNDICE B – INTERFACE DO SHIM NA ESPECIFICAÇÃO *RUNTIME V2* 91

1 INTRODUÇÃO

A automação industrial vem incorporando tecnologias originalmente concebidas para o domínio da Tecnologia da Informação (TI). Um dos vetores dessa mudança é a evolução do hardware embarcado, que passou a oferecer capacidade computacional suficiente para hospedar aplicações antes restritas a servidores dedicados. Somado às demandas de digitalização associadas ao movimento conhecido como Indústria 4.0 (Lasi *et al.*, 2014), esse deslocamento do processamento para junto dos ativos físicos estabelece uma camada intermediária entre o chão de fábrica e a TI corporativa, no que se consolidou como o paradigma de Computação de Borda (*Edge Computing*) (Qiu *et al.*, 2020).

O termo borda é empregado aqui em sentido restrito. Designa os dispositivos que processam dados junto à sua fonte, sob restrições de *Central Processing Unit* (CPU), memória e armazenamento impostas por custo e por consumo energético. A infraestrutura de TI corporativa fica fora desse recorte, pois ocupa a outra extremidade da camada e não compartilha tais restrições.

Diferentemente das aplicações tipicamente associadas à computação em nuvem, como serviços *web*, as aplicações industriais estão sujeitas a restrições próprias do domínio, entre elas a sensibilidade temporal e a segurança operacional. Por essa razão, as tecnologias herdadas da TI precisam ser adaptadas e validadas antes de serem empregadas no domínio da automação (Sollfrank *et al.*, 2021). Essa transposição tem sido viabilizada pela conteinerização e por plataformas de orquestração, como o Kubernetes, que permitem encapsular a aplicação junto de suas dependências e coordenar sua execução de forma isolada sobre um conjunto de nós (Burns *et al.*, 2022).

Uma aplicação depende de bibliotecas específicas, de suas versões e de arquivos de configuração. O contêiner empacota esse conjunto em uma imagem única e imutável, o que dispensa reproduzi-lo manualmente em cada dispositivo e permite que aplicações com requisitos distintos coexistam no mesmo nó, sem conflito de versões de biblioteca (Sollfrank *et al.*, 2021). Atualização e reversão passam a substituir a unidade inteira, em vez de modificar o sistema de arquivos de um dispositivo em operação.

Nesse movimento de virtualização, aplicações de controle industrial passam a ser implementadas como software executado sobre sistemas operacionais de propósito geral, beneficiando-se das mesmas ferramentas de implantação, observabilidade e atualização já consolidadas no domínio da TI. O Controlador Lógico Programável Virtual (vPLC) é uma das manifestações desse modelo, no qual a lógica de um controlador lógico programável passa a ser executada em software em dispositivos de uso geral, com a possibilidade de consolidar, em um mesmo nó, múltiplas aplicações de controle que tradicionalmente exigiriam um Controlador Lógico Programável (PLC) físico cada (Perez *et al.*, 2022).

O ferramental herdado da nuvem, no entanto, foi originalmente concebido para tratar as cargas de trabalho como entidades sem estado, projetadas para serem reiniciadas em qualquer

nó disponível sem maiores consequências. Aplicações industriais raramente compartilham essa propriedade. Tais aplicações executam ciclicamente, em janelas que não toleram reinicialização sem perda de continuidade, e mantêm temporizadores, variáveis e máquinas de estado que não podem ser reconstruídas a partir da imagem do contêiner (Tošić, 2024; Koziolek; Burger; Peedikayil, 2024).

A possibilidade de deslocar tais aplicações entre nós, sem perder o estado em execução, abre cenários operacionais relevantes para a automação industrial. Tolerância a falhas é um deles. A iminência de degradação ou falha de um nó pode ser respondida com a transferência preventiva da aplicação para um par saudável, antes que o processo controlado seja afetado. A manutenção planejada também se beneficia, já que correções no hardware ou no sistema operacional passam a ser aplicadas sem parada de máquina. O balanceamento de carga é outra aplicação possível, na qual algoritmos externos avaliam a utilização de recursos ou o consumo de energia e realizam migrações coordenadas.

Para sustentar esses cenários, a técnica de migração de aplicações com estado captura, transfere e restaura o estado em execução de um processo para outro nó. O *Checkpoint/Restore In Userspace* (CRIU) é hoje a base mais consolidada para esse tipo de operação em sistemas Linux, sendo capaz de serializar memória, descritores de arquivo, conexões e demais recursos do espaço de usuário (CRIU Project, 2025a).

Nó significa, ao longo do texto, um dispositivo físico distinto, de modo que migrar entre nós implica mudança de máquina. O que reúne esses dispositivos em um *cluster* é a conectividade de rede entre eles, e não a proximidade física. Homogêneo, por sua vez, refere-se à arquitetura do processador, condição necessária para restaurar em um nó o estado serializado em outro, conforme detalhado na Seção 4.4.

1.1 PROBLEMA DE PESQUISA

Diante desse quadro, esta dissertação investiga o seguinte problema: de que forma é possível realizar a migração de uma aplicação com estado entre nós embarcados homogêneos, sem comprometer a continuidade da aplicação migrada e respeitando as restrições de CPU, memória e armazenamento típicas desses dispositivos?

1.2 OBJETIVOS

O objetivo principal deste trabalho é propor, implementar e avaliar uma arquitetura de migração de aplicações com estado entre dispositivos embarcados homogêneos, capaz de preservar o estado em execução durante o deslocamento entre nós, integrada de forma nativa ao orquestrador e transparente para a aplicação migrada.

1.2.1 Objetivos específicos

São objetivos específicos deste trabalho:

a) Revisar o estado da arte em migração de aplicações com estado, com ênfase em trabalhos que empregam CRIU, conteinerização e Kubernetes em ambientes de borda ou com recursos restritos;
b) Modelar a arquitetura de um sistema de migração de aplicações com estado, cuja operação seja transparente para a aplicação, utilizando componentes nativos de um ecossistema de orquestração;
c) Implementar o sistema proposto e validá-lo em dispositivos embarcados reais, coletando métricas de desempenho e indisponibilidade durante a migração;
d) Caracterizar experimentalmente o desempenho da migração nos cenários de mesmo nó e entre nós distintos, decompondo o tempo de indisponibilidade em fases de captura, transferência e restauração do estado;
e) Identificar e discutir os gargalos observados no hardware embarcado, avaliar a simetria entre direções de migração e caracterizar o efeito da transferência de rede sobre o tempo total da operação.

1.3 JUSTIFICATIVA

Os cenários descritos anteriormente dependem de um mecanismo capaz de preservar o estado das aplicações em movimento entre nós. Em ambientes de nuvem, esse mecanismo tem sido alvo de investigação ativa, com trabalhos recentes explorando otimizações para a migração com estado em orquestradores Kubernetes (Koziolek; Burger; Peedikayil, 2024; Calagna *et al.*, 2025; Zhang *et al.*, 2025).

Em dispositivos embarcados, com restrições de processamento, memória e rede, o comportamento dessas técnicas permanece pouco documentado (Soussi; Gür; Stiller, 2024). Os resultados apresentados nesta dissertação visam preencher parte dessa lacuna. Os testes experimentais conduzidos em um *cluster* composto por dispositivos embarcados fornecem base quantitativa sobre tempos de indisponibilidade e simetria entre direções de migração, material útil tanto para o uso da técnica em aplicações reais quanto como referência para algoritmos de balanceamento de carga que a empreguem como mecanismo de execução.

1.4 ORGANIZAÇÃO DO TRABALHO

Este trabalho está estruturado da seguinte forma:

- O Capítulo 2 apresenta os fundamentos teóricos necessários para compreender os conceitos de contêineres, orquestração com Kubernetes e k3s, *checkpoint/restore* com CRIU e os protocolos de comunicação utilizados;

- O Capítulo 3 apresenta o estado da arte em migração de aplicações com estado, posicionando a contribuição deste trabalho em relação à literatura;
- O Capítulo 4 apresenta a modelagem e a arquitetura da solução proposta, com seus componentes, fluxo de migração e máquina de estados;
- O Capítulo 5 detalha o desenvolvimento da solução, abordando a implementação do shim, do manager e da comunicação entre nós;
- O Capítulo 6 apresenta os resultados obtidos e a análise de desempenho da solução nos cenários experimentais avaliados;
- Por fim, o Capítulo 7 traz as conclusões gerais do trabalho e sugestões para pesquisas futuras.

2 CONCEITOS FUNDAMENTAIS

Este capítulo apresenta as ferramentas e conceitos fundamentais utilizados no desenvolvimento deste trabalho. A base teórica aqui estabelecida sustenta a leitura do estado da arte, no capítulo seguinte, e a compreensão da arquitetura proposta.

2.1 ARQUITETURA DO KERNEL LINUX

O *kernel* é o núcleo do sistema operacional Linux e executa no nível mais privilegiado do processador. Ele é responsável pela gestão direta do hardware, incluindo CPU, memória e dispositivos de *Input/Output* (I/O), além de arbitrar o acesso a esses recursos pelas aplicações (Love, 2010).

2.1.1 Espaço de usuário e espaço do *kernel*

O sistema é dividido fundamentalmente em dois espaços de memória distintos, visando garantir segurança e estabilidade. No espaço do *kernel* (*kernel space*) executam-se o núcleo do sistema operacional e os *drivers* de dispositivo, com privilégios totais, de modo que falhas nessa região podem comprometer a integridade de todo o sistema. No espaço de usuário (*user space*) executam-se as aplicações convencionais com privilégios restritos, sem acesso direto ao hardware, devendo solicitar serviços ao *kernel* por meio de uma interface estrita conhecida como chamadas de sistema (*system calls*) (Love, 2010).

Esta separação é vital para a estabilidade de sistemas industriais, pois assegura que uma falha em uma aplicação de controle, como uma Interface Homem-Máquina (IHM) travada, não afete o sistema operacional como um todo, permitindo que outros processos críticos continuem operando.

2.1.2 Criação de processos: fork e exec

Nos sistemas Unix, a criação de processos ocorre de forma distinta da adotada pela maioria dos sistemas operacionais. Em vez de utilizar uma única operação para criar um novo processo, carregar um executável em um novo espaço de endereçamento e iniciar sua execução, o Unix separa essas etapas em duas funções: `fork()` e `exec()`. A primeira, `fork()`, cria um processo filho como uma cópia da tarefa atual, diferindo do processo pai principalmente em seu identificador de processo *Process Identifier* (PID) e em alguns recursos e estatísticas que não são herdados. A segunda, `exec()`, carrega um novo executável no espaço de endereçamento do processo e inicia sua execução. A combinação `fork()` seguida de `exec()` equivale, portanto, ao mecanismo único de criação de processos oferecido pela maioria dos sistemas operacionais (Love, 2010).

2.1.3 Grupos de controle (*cgroups*)

Os *cgroups* (grupos de controle) são um recurso do *kernel* do Linux que permite organizar processos em grupos hierárquicos, possibilitando limitar, contabilizar e isolar o uso de recursos de hardware para esses grupos (Love, 2010).

Em aplicações industriais, os *cgroups* contribuem para limitar a interferência entre cargas e reduzir variações temporais, embora não garantam, isoladamente, o determinismo temporal. É possível, por exemplo, criar um grupo para processos de "Controle Crítico" e garantir a eles uma parcela majoritária da CPU, enquanto processos como os de "Monitoramento" ficam restritos aos recursos remanescentes. Sem os *cgroups*, um processo secundário poderia consumir excessivamente a CPU, causando latência indesejada na aplicação principal do sistema, como uma malha de controle.

Entre os principais subsistemas, ou controladores, dos *cgroups*, o controlador de CPU garante fatias de tempo de processamento e define prioridades, o controlador de memória limita o uso de *Random Access Memory* (RAM) e de memória virtual (*swap*), e o controlador de PIDs limita o número total de processos que podem ser criados, prevenindo ataques de negação de serviço por exaustão de recursos.

2.1.4 Namespaces

Enquanto os *cgroups* limitam quanto de cada recurso um grupo de processos pode consumir, os *namespaces* delimitam o que é visível a ele. Um *namespace* envolve um recurso global do sistema em uma abstração que dá aos processos pertencentes a ele uma instância própria desse recurso, compartilhada entre esses processos e invisível aos demais (The Linux Kernel Community, 2024a).

Esse mecanismo é a base dos contêineres (The Linux Kernel Community, 2024a), que de forma preliminar, podem ser entendidos como processos executados em *namespaces* próprios e sob limites de *cgroups*, conceito detalhado na Seção 2.2. Dos oito tipos de *namespace* definidos pelo *kernel*, os mais relevantes para este trabalho são:

- *namespace* de PID: isola a árvore de identificadores de processo, de modo que um processo dentro de um contêiner pode ter o PID 1, como se fosse o processo de inicialização do sistema, ainda que no *host* possua um PID distinto;
- *namespace* de rede: fornece ao contêiner sua própria pilha de rede, com interfaces, tabelas de roteamento e regras de *firewall* isoladas do *host* e dos demais contêineres;
- *namespace* de montagem: permite que o contêiner tenha seu próprio sistema de arquivos raiz e pontos de montagem, independentes do sistema hospedeiro;
- *namespace* de *Unix Time-Sharing System* (UTS): segrega identificadores de sistema, como *hostname* e nome de domínio;

- *namespace* de tempo: isola o relógio monotônico observado pelo processo. Esse relógio, diferentemente do relógio do sistema, avança de forma estritamente crescente a partir de um ponto arbitrário no passado e não é afetado por ajustes externos, como sincronização via *Network Time Protocol* (NTP) ou mudanças de fuso. O isolamento possibilita a aplicação de um deslocamento por *namespace*, mecanismo utilizado na restauração de um contêiner para reaplicar, ao relógio do processo restaurado, o valor registrado no instante do *checkpoint* (The Linux Kernel Community, 2024b).

A combinação de *cgroups* (limites) e *namespaces* (isolamento) constitui a base tecnológica para a implementação de contêineres no Linux.

2.2 COMPUTAÇÃO EM CONTÊINERES

Observa-se uma tendência no setor de automação para a incorporação da virtualização de software através do conceito de conteinerização em sistemas de controle distribuído (Sollfrank *et al.*, 2021). A virtualização por meio de contêineres permite o encapsulamento da aplicação e de suas dependências em unidades leves, autossuficientes, modularizadas e escaláveis.

O mecanismo de isolamento de recursos utiliza as ferramentas do *kernel* do Linux supracitadas, *cgroups* e *namespaces*. Diferente das máquinas virtuais tradicionais, os contêineres compartilham o mesmo *kernel* do sistema hospedeiro, o que resulta em menor sobrecarga. Aplicações com requisitos temporais podem ser executadas em ambientes conteinerizados quando o sistema operacional oferece escalonamento com garantias de tempo real. Nessas condições, Sollfrank *et al.* (2021) demonstram que o atraso adicionado pela conteinerização permanece dentro dos limites exigidos por aplicações de controle de tempo real brando (*soft real-time*).

2.3 KUBERNETES

O Kubernetes é uma plataforma de orquestração de contêineres criada originalmente pelo Google e atualmente mantida pela *Cloud Native Computing Foundation* (CNCF), sendo uma das soluções mais utilizadas em ambientes de computação em nuvem e um dos projetos de código aberto mais populares do mundo (Silverthorne; Hendrick; CNCF, 2025). Sua função primordial é automatizar o processo de implantação, escalonamento e gerenciamento de aplicações em contêineres em um *cluster* de máquinas (Burns *et al.*, 2022).

Se um nó físico falha, o Kubernetes detecta o evento e automaticamente reagenda as cargas de trabalho afetadas para outros nós saudáveis, garantindo a continuidade do processo.

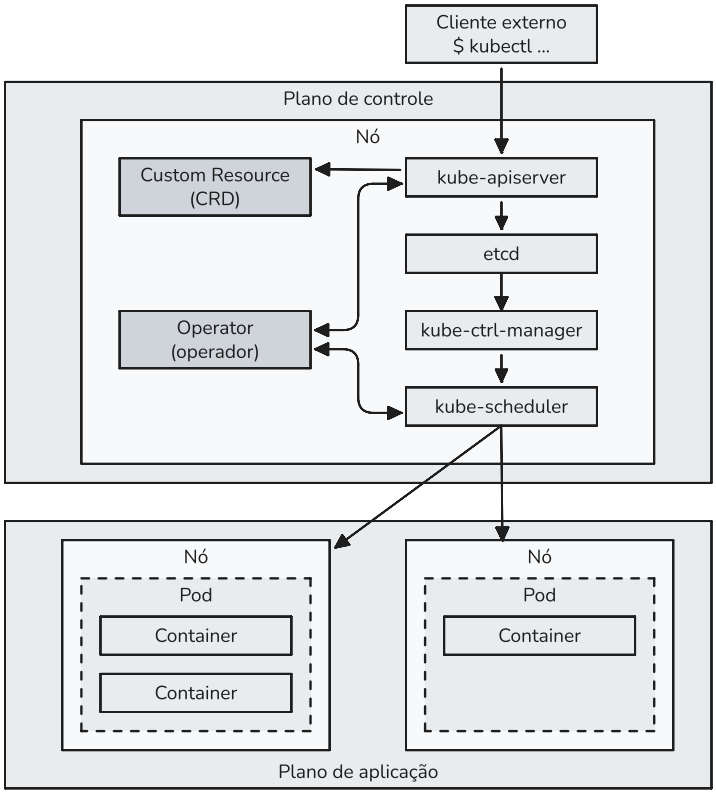
2.3.1 Arquitetura

A arquitetura do Kubernetes é dividida entre plano de controle (*control plane*) e nós de computação (*worker nodes*), conforme ilustrado na Figura 1. O plano de controle executa os

componentes necessários para gerenciar o *cluster*, que é uma coleção de unidades de processamento. Em contrapartida, os nós de trabalho executam as aplicações definidas pelo usuário. Implantações que exigem alta disponibilidade replicam os componentes do plano de controle em múltiplos nós dedicados a essa função.

Um *cluster* Kubernetes depende de sistemas de apoio, como um sistema de armazenamento persistente e um registro de imagens de contêineres. O primeiro provê abstração de armazenamento de dados para os contêineres, permitindo a integração com o *cluster*, enquanto o segundo armazena as imagens necessárias à execução das aplicações.

Figura 1 – Arquitetura do Kubernetes



Fonte: Adaptado de (Dobies; Wood, 2020)

A operação do *cluster* apoia-se em um modelo declarativo. O usuário registra o estado desejado por meio da *Application Programming Interface* (API) do orquestrador, e controladores em execução contínua comparam esse estado com o estado observado, atuando para reconciliar as diferenças (Burns *et al.*, 2022). Os componentes que implementam esse modelo distribuem-se entre o plano de controle e os nós de trabalho. No plano de controle destacam-se:

- kube-apiserver: componente central que expõe a API do Kubernetes, atuando como única porta de entrada para comandos de gerenciamento;
- etcd: banco de dados chave-valor consistente e de alta disponibilidade, empregado como armazenamento de apoio para todos os dados do *cluster*, tanto o estado desejado quanto o estado observado (Dobies; Wood, 2020);

- kube-scheduler: atribui novos Pods a nós específicos, com base em requisitos de recursos, como CPU e memória, e em políticas de afinidade;
- kube-controller-manager: executa, em um único processo, os controladores nativos do orquestrador. Cada controlador responde por um tipo de recurso, como Deployment, ReplicaSet ou Node, e atua continuamente para reconciliar as instâncias desse tipo.

Nos nós de trabalho (*worker nodes*), que são as máquinas físicas ou virtuais onde as aplicações de usuário efetivamente são executadas, dois componentes são essenciais:

- kubelet: agente que recebe do kube-apiserver as especificações de Pods atribuídas ao nó e garante que os contêineres descritos nessas especificações estejam em execução e saudáveis, interagindo com o *runtime* de contêineres;
- kube-proxy: mantém as regras de rede no nó, permitindo a comunicação com os Pods a partir de dentro ou de fora do *cluster*.

2.3.2 Abstrações

O Kubernetes gerencia o sistema através de uma série de objetos abstratos. Os mais relevantes para este trabalho são:

- Pod: unidade básica de trabalho e agendamento, formada por um ou mais contêineres que compartilham recursos como armazenamento e o *namespace* de rede (Burns *et al.*, 2022). Por compartilharem a mesma pilha de rede, os contêineres de um Pod comunicam-se diretamente pela interface de *loopback*, cujo endereço *Internet Protocol* (IP) recebe tradicionalmente o nome `localhost`. Cada Pod recebe um endereço de rede próprio, atribuído por um *plugin* em conformidade com a *Container Networking Interface* (CNI);
- ReplicaSet: garante que um número especificado de réplicas de Pods esteja sempre em execução, assegurando a disponibilidade contínua da aplicação em caso de falhas;
- DaemonSet: gerencia a execução de uma réplica de Pod em todos os nós, ou em um subconjunto, do *cluster*. É essencial para aplicações de infraestrutura, como agentes de monitoramento e o manager de migração proposto neste trabalho;
- Deployment: estende as funcionalidades do ReplicaSet, gerenciando atualizações de versão e reversões de forma controlada;
- *Custom Resource Definitions* (CRDs): mecanismo de extensão que permite registrar tipos próprios de recurso na API do orquestrador. Uma vez declarado, o recurso personalizado passa a ser armazenado no mesmo banco de dados que guarda os objetos nativos, o etcd no Kubernetes ou, por padrão, o SQLite no K3s (Seção 2.4), e manipulado pelas mesmas ferramentas e mecanismos de reconciliação dos objetos nativos, sem exigir modificação

do orquestrador. É sobre esse mecanismo que a arquitetura proposta define o recurso de migração descrito no Capítulo 4;

- RuntimeClass: nomeia uma configuração de *runtime* de contêineres disponível nos nós do *cluster*, associando um nome de uso na API a um identificador declarado na configuração do gerenciador de contêineres (Kubernetes Authors, 2025d). Um Pod seleciona essa configuração pelo campo `runtimeClassName` de sua especificação, e o kubelet repassa a escolha ao gerenciador no momento da criação. Esse mecanismo permite que *runtimes* distintos coexistam no mesmo *cluster*, atribuídos por carga de trabalho, e é o que viabiliza a adoção seletiva do shim desenvolvido neste trabalho, descrito no Capítulo 5.

2.3.3 Operadores

Os Operadores (*operators*) representam um padrão do ecossistema Kubernetes voltado à automação da gestão de recursos complexos, como bancos de dados e aplicações distribuídas com estado, cuja operação envolve cópia de segurança e restauração do estado, atualização de versão acompanhada de mudanças de esquema e eleição de líder entre réplicas. Sua proposta é codificar o conhecimento operacional humano em software, automatizando tarefas de criação, atualização e manutenção (Dobies; Wood, 2020; Kubernetes Authors, 2025a).

Um Operador combina dois elementos. O primeiro é o CRD, que declara o tipo de recurso e os campos que descrevem seu estado desejado. O segundo é o controlador, que observa as instâncias desse recurso e executa as ações necessárias para que o estado do sistema corresponda ao que foi declarado. É esse par que permite estender o orquestrador com lógica própria sem alterar seus componentes nativos, e é o padrão adotado pelo componente de coordenação desenvolvido neste trabalho.

2.3.4 Arquitetura de execução dos contêineres

A execução dos contêineres envolve a interação entre diversos componentes. O kubelet se comunica com o *runtime* de contêineres por meio da *Container Runtime Interface* (CRI), interface padronizada que permite a interoperabilidade entre diferentes implementações de *runtime*. Atualmente, o containerd é o mais utilizado e expõe essa interface de forma nativa, como um serviço *Google Remote Procedure Call* (gRPC) em soquete Unix local (Rosso *et al.*, 2021).

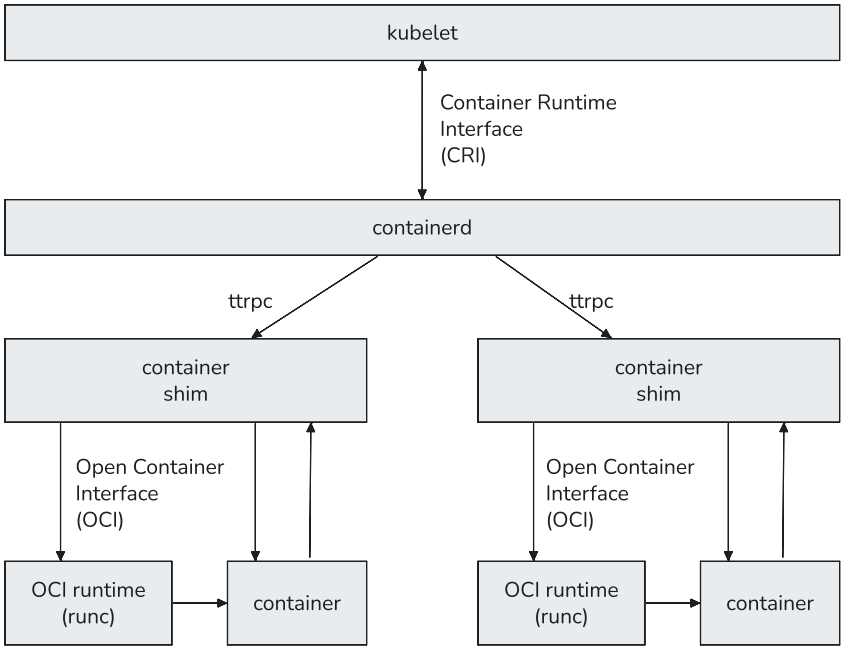
O containerd, por sua vez, não executa diretamente os contêineres. Essa tarefa é delegada ao shim e ao runc. Em software, o termo *shim* designa uma camada intermediária que intercepta as chamadas trocadas entre dois componentes e as adapta, sem exigir modificação de nenhum deles. No containerd, o shim é um processo que se encontra entre o containerd e o runc, garantindo que cada contêiner tenha um processo independente para manter seus *namespaces*, bem como para redirecionar *Standard Input* (STDIN), *Standard Output* (STDOUT) e *Standard Error* (STDERR). A comunicação entre o containerd e cada shim é feita pelo protocolo *Tiny Transport Remote*

Procedure Call (ttrpc), variante enxuta do gRPC discutida na Seção 2.8.2, sobre um soquete Unix dedicado ao contêiner. Como o shim, e não o containerd, é o processo pai do contêiner, o containerd pode ser reiniciado, seja por atualização ou por falha, sem que os contêineres em execução sejam encerrados, restabelecendo a comunicação com os shims existentes ao retornar (Containerd Authors, 2024).

O runc é a implementação de referência do *runtime* da *Open Container Initiative* (OCI) (Open Container Initiative, 2024). Ele fornece a interface de baixo nível para criar, executar e gerenciar contêineres segundo a especificação OCI. Através dele, ocorre a preparação do ambiente de execução: configuração de *cgroups*, isolamento de *namespaces* e inicialização do processo principal do contêiner. O shim encaminha os comandos do containerd para o runc, que executa operações como criação, execução, parada e remoção do contêiner. Além disso, uma das grandes vantagens do runc é a sua integração com o CRIU (Seção 2.6) para realizar migração com estado de contêineres.

A interação entre esses componentes pode ser visualizada na Figura 2, que mostra como kubelet, containerd, shim e runc se relacionam na execução de contêineres em Kubernetes. A figura representa o caminho padrão de execução, anterior a qualquer extensão introduzida neste trabalho. O shim ali representado é o shim padrão do containerd, e o contêiner mostrado corresponde ao contêiner de aplicação de um Pod.

Figura 2 – Interação entre os componentes do Kubernetes no uso do containerd como *runtime* padrão



Fonte: Elaborado pelo autor (2026).

2.4 K3S

Embora o Kubernetes seja o padrão de orquestração, sua complexidade pode inviabilizar o uso em cenários com restrições computacionais, como dispositivos embarcados. O K3s, desenvolvido pela SUSE/Rancher, é uma distribuição leve e certificada do Kubernetes projetada para atender a essas demandas.

Na terminologia do K3s, o nó que executa o plano de controle é denominado nó servidor, e os nós de trabalho são denominados nós agentes. Essa nomenclatura é adotada no restante do texto sempre que a descrição se referir ao ambiente experimental.

O K3s adota uma abordagem de simplificação estrutural, reduzindo o consumo de memória e CPU ao integrar diversos componentes em um único binário compacto (Rancher, 2019). Diferentemente do Kubernetes tradicional, que utiliza o etcd como armazenamento, o K3s adota por padrão o SQLite como banco de dados embarcado, o que facilita a instalação em dispositivos de recursos limitados. O SQLite não admite *clusters* com múltiplos nós servidores, e o etcd embarcado é selecionado quando o K3s é configurado para formar ou integrar um *cluster* desse tipo (K3s Project Authors, 2026b). Como o ambiente experimental deste trabalho conta com um único nó servidor (Seção 6.1), o estado do *cluster*, incluindo os recursos personalizados, é persistido em SQLite.

A conectividade entre Pods também é provida por padrão. O K3s inclui o Flannel, provedor de rede de camada 3 que implementa a CNI, como *plugin* de rede padrão, e permite desabilitá-lo para instalar outra implementação, como Canal, Calico ou Cilium (K3s Project Authors, 2026a). Neste trabalho manteve-se o Flannel na configuração padrão, que atendeu aos experimentos sem necessidade de substituição.

Por essas características, o projeto declara como principais alvos de uso os cenários de computação de borda, Internet das Coisas (IoT) e ambientes de desenvolvimento, destacando sua simplicidade e leveza. O K3s é escrito em Go, e os requisitos mínimos declarados pelo projeto são de dois núcleos e 2 GB de memória RAM para o nó servidor, e de um núcleo e 512 MB de RAM para os nós agentes (Rancher, 2019).

2.5 SERVIÇOS COM ESTADO E SEM ESTADO

A capacidade de tolerar falhas e migrar cargas de trabalho depende intrinsecamente da natureza da aplicação, que pode ser sem estado (*stateless*) ou com estado (*stateful*) (Calagna *et al.*, 2024).

Serviços sem estado são aqueles que não retêm, entre execuções sucessivas, nenhuma informação interna necessária ao seu funcionamento. Qualquer instância pode, portanto, ser encerrada e recriada em outro nó sem perda de contexto operacional. Para essas aplicações, a replicação é uma solução simples e eficaz para alcançar alta disponibilidade e tolerância a falhas, já que não há necessidade de sincronização de estado. Exemplos incluem aplicações CPU-*bound* ou GPU-*bound*, como reconhecimento de objetos (Cacciabue *et al.*, 2024). O Kubernetes,

por exemplo, fornece mecanismos nativos para gerenciar o ciclo de vida e a recuperação de aplicações sem estado (Dobies; Wood, 2020).

Em contraste, serviços com estado mantêm um contexto interno crítico para sua operação. A migração destas aplicações em cenários de borda exige a transferência integral deste estado para garantir a continuidade do serviço (Calagna *et al.*, 2024).

Para uma migração bem-sucedida, o seguinte conjunto de informações de estado deve ser transferido ao nó de destino (Calagna *et al.*, 2024):

- o contexto de CPU, incluindo registradores, a estrutura da árvore de processos e os *namespaces* associados;
- o conteúdo da memória, ou seja, as páginas alocadas na memória principal;
- os soquetes de rede estabelecidos;
- os descritores de arquivos abertos.

A sincronização do estado é o maior desafio para aplicações com estado. Ainda assim, apesar da tendência de desenvolver microsserviços sem estado, microsserviços com estado seguem amplamente empregados, em razão da complexidade de refatorar aplicações monolíticas legadas, e alguns serviços utilitários essenciais continuarão a depender de estado mesmo que as implementações sem estado se tornem predominantes (Calagna *et al.*, 2024).

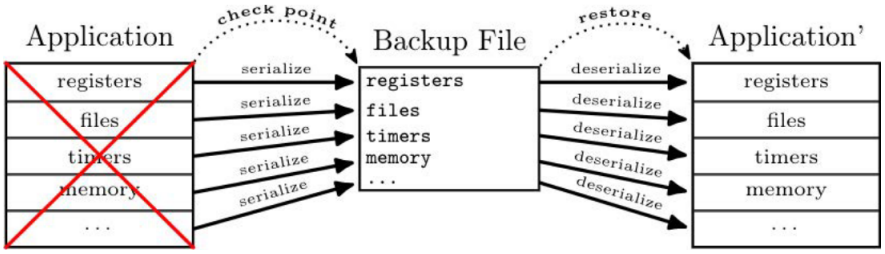
2.6 CHECKPOINT/RESTORE IN USERSPACE (CRIU)

O CRIU é uma ferramenta que permite suspender a execução de uma aplicação (ou contêiner) e persistir seu estado interno em disco (*Checkpoint*). Este estado pode ser utilizado posteriormente para restaurar a aplicação exatamente no ponto de interrupção (*Restore*) (CRIU Project, 2025a), e tem papel fundamental no contexto de migração de aplicações com estado (*stateful*).

Antes de iniciar o procedimento de *checkpoint*, é necessário garantir que o estado do processo não sofrerá alterações, por exemplo, abertura de novos arquivos, criação de soquetes ou outras modificações. Para isso, o processo é imobilizado por meio da chamada de sistema `ptrace` ou do *cgroups freezer* (CRIU Project, 2025b). Na prática, não apenas o processo principal precisa ser congelado, mas também toda a sua árvore de processos filhos, garantindo a consistência do estado. Após paralisar a árvore de processos, o CRIU escreve as informações coletadas em arquivos de imagem, como mostrado na Figura 3. Esses dados incluem praticamente todo o conteúdo encontrado em `/proc/<PID>/*`, como descritores de arquivos, parâmetros de *pipes*, memória mapeada e estados do processador.

A captura da memória é realizada através da injeção de um código parasita no espaço de endereçamento do processo alvo (CRIU Project, 2025d). Utilizando `ptrace`, este código

Figura 3 – Processo de *checkpoint* e *restore* com CRIU



Fonte: Venkatesh *et al.* (2019)

coordena a cópia das páginas de memória para *buffers* e, posteriormente, para os arquivos de imagem, utilizando chamadas eficientes como `vmsplice` (CRIU Project, 2025c).

Um desafio crítico é a preservação de conexões *Transmission Control Protocol* (TCP) estabelecidas. O CRIU soluciona este problema utilizando o recurso `TCP_REPAIR` do *kernel* Linux. Este mecanismo permite colocar o soquete em um estado que possibilita a leitura e escrita direta de seus parâmetros internos (como números de sequência e filas), permitindo a reconstrução transparente da conexão no momento da restauração (CRIU Project, 2025e).

Enquanto a árvore de processos permanece congelada, nenhuma requisição destinada à aplicação é atendida. Os pacotes que chegam ao nó ainda podem ser recebidos pelo *kernel* e acumulados nos *buffers* dos soquetes, mas não são lidos pelo processo e se perdem quando o contêiner de origem é encerrado após o *checkpoint*. Na configuração adotada neste trabalho, o CRIU não preserva conexões TCP: conexões cujo estabelecimento ainda não foi concluído são ignoradas (opção `skip-in-flight`) (CRIU Project, 2026a), e conexões estabelecidas são restauradas no estado fechado, mantendo-se apenas os soquetes em escuta (CRIU Project, 2026b). Além disso, a instância restaurada em outro nó recebe um novo endereço IP, atribuído pelo *plugin* CNI desse nó. As requisições recebidas durante a migração são, portanto, descartadas, e cabe aos clientes restabelecer a conexão com a instância restaurada.

Apesar de suas capacidades, o CRIU não é capaz de realizar *checkpoint/restore* de todas as aplicações. Processos que dependem fortemente de recursos externos, como dispositivos físicos, drivers específicos ou sistemas gráficos, geralmente não podem ser migrados, pois parte do estado não está disponível no espaço do usuário. Da mesma forma, a restauração entre arquiteturas ou plataformas diferentes pode ser inviável, já que estruturas internas de registradores de CPU e drivers variam conforme o ambiente de execução (CRIU Project, 2025f).

2.6.1 Modos de migração suportados pelo CRIU

A partir do mecanismo básico de *checkpoint* e *restore* descrito nas seções anteriores, o CRIU pode ser empregado em três condições distintas de migração entre nós, que se diferenciam pela ordem e pelo paralelismo entre o congelamento do processo e a transferência das páginas de memória (CRIU Project, 2026d).

A migração a frio (*cold migration*), modalidade adotada nesta dissertação, congela o

processo, serializa todo o seu estado em arquivos de imagem, transfere-os para o nó de destino e somente então restaura a execução. A janela de indisponibilidade percebida pela aplicação corresponde à soma dos tempos de criação do *checkpoint*, transferência e restauração. Trata-se da modalidade mais simples de implementar e da que apresenta maior indisponibilidade para estados volumosos.

No modelo de migração ao vivo com pré-cópia (*live migration with pre-copy*), o CRIU transfere as páginas de memória para o nó de destino em iterações sucessivas, enquanto o processo continua em execução no nó de origem, utilizando o mecanismo *page-server* (CRIU Project, 2026e). Cada iteração identifica as páginas modificadas desde a anterior e retransmite apenas esse subconjunto. Quando o volume de páginas modificadas se estabiliza, o processo é congelado para o envio das páginas restantes e, em seguida, restaurado no destino. A janela de indisponibilidade reduz-se ao tempo dessa fase final, em troca de maior volume total transmitido pela rede.

Já na migração ao vivo com pós-cópia (*live migration with post-copy*), o processo é restaurado no nó de destino antes da transferência completa das páginas de memória, que passam a ser obtidas sob demanda à medida que o processo as acessa. O mecanismo utiliza o recurso `userfaultfd` do *kernel* Linux, que entrega faltas de página ao espaço de usuário, e a funcionalidade *lazy-pages* do CRIU, que atende essas faltas por meio de um processo em segundo plano conectado ao nó de origem (CRIU Project, 2026c). A indisponibilidade percebida é mínima, mas a aplicação fica sensível à latência da rede durante o período em que páginas remotas ainda podem ser acessadas.

2.7 FERRAMENTAS DE DESENVOLVIMENTO DE SISTEMAS

A implementação de infraestrutura de alto desempenho requer ferramentas que conciliem eficiência, segurança e suporte a concorrência.

2.7.1 Linguagens de programação

Duas linguagens foram empregadas na implementação dos componentes desenvolvidos nesta dissertação, escolhidas em função das características exigidas por cada subsistema.

A linguagem Go, desenvolvida pelo Google, destaca-se pela simplicidade e pelo suporte nativo a concorrência por meio de *goroutines*, sendo a linguagem predominante no ecossistema de contêineres e de computação em nuvem (Donovan; Kernighan, 2015). Por essa razão, foi adotada na implementação do shim de contêineres, que se integra diretamente ao containerd e reaproveita as bibliotecas oficiais do projeto.

A linguagem Rust, por sua vez, é orientada à segurança e ao desempenho, garantindo segurança de memória em tempo de compilação sem depender de *garbage collector*, característica que a torna adequada a componentes críticos de infraestrutura nos quais latência e estabilidade

são prioritárias (Rust Foundation, 2025; Klabnik; Nichols; Krycho, 2025). Foi a linguagem adotada para o manager distribuído, responsável pela coordenação da migração entre nós.

2.8 PROTOCOLOS DE CHAMADA REMOTA DE PROCEDIMENTO

Chamadas de procedimento remoto (*Remote Procedure Call* (RPC)) são um paradigma de comunicação entre processos no qual um programa invoca uma função em outro processo ou máquina como se fosse uma chamada local, abstraindo os detalhes de transporte e serialização. No contexto deste trabalho, dois protocolos RPC são empregados: o gRPC para comunicação entre componentes distintos do *cluster*, e o ttrpc para comunicação interna entre o containerd e os shims no mesmo nó.

2.8.1 gRPC

O gRPC é um framework de alto desempenho para chamadas de procedimento remoto, desenvolvido pelo Google e mantido pela CNCF. Ele utiliza o protocolo *Hypertext Transfer Protocol* (HTTP)/2 para transporte, permitindo multiplexação e *streaming* bidirecional, e Protocol Buffers (Protobuf) para serialização de dados. A serialização binária do Protobuf resulta em mensagens mais compactas e processamento mais rápido em comparação a formatos textuais como *JavaScript Object Notation* (JSON), características essenciais para a eficiência da comunicação em sistemas distribuídos (gRPC Contributors, 2023).

2.8.2 ttrpc

O ttrpc é uma variante do gRPC projetada para comunicação dentro de um único nó, em que a sobrecarga do HTTP/2 é desnecessária. Ele reutiliza a mesma definição de serviços em Protobuf do gRPC, mas substitui o transporte HTTP/2 por soquetes Unix, eliminando o custo de estabelecimento de conexão TCP e reduzindo o consumo de memória. Essas características tornam o ttrpc adequado para processos de baixo nível com ciclo de vida efêmero, como os shims do containerd. É por meio do ttrpc que o containerd se comunica com cada shim para invocar operações do ciclo de vida dos contêineres (Containerd Authors, 2024).

2.9 SÍNTESE DO CAPÍTULO

Este capítulo apresentou os fundamentos técnicos sobre os quais a solução proposta é construída. Foram revisados os mecanismos do *kernel* Linux que sustentam a conteinerização, em particular *cgroups*, *namespaces* e a criação de processos via `fork` e `exec`, bem como a abstração de contêineres e o *runtime* runc, a orquestração com k3s e a distinção entre serviços sem estado e com estado. Em seguida, foram detalhados o CRIU e seus três regimes de migração, a frio, com pré-cópia e com pós-cópia, as linguagens Go e Rust adotadas na implementação, e os protocolos gRPC e ttrpc usados para comunicação entre componentes.

Esses conceitos formam o vocabulário técnico necessário para o capítulo seguinte, no qual a literatura sobre migração com estado é analisada. São também os elementos combinados na arquitetura proposta no Capítulo 4, que reúne shim do containerd, CRIU e k3s para realizar a migração com estado de contêineres entre nós embarcados.

3 ESTADO DA ARTE

A migração ao vivo de contêineres com preservação de estado tornou-se objeto de pesquisa ativa nos últimos anos, motivada pela consolidação de contêineres Linux como unidade padrão de implantação em plataformas distribuídas e pela disseminação de orquestradores leves em ambientes de borda e industriais. O levantamento de Soussi, Gür e Stiller (2024) cataloga as principais técnicas empregadas na área, em particular pré-cópia, pós-cópia e transferência incremental, e situa o uso do CRIU como caminho dominante para migração com estado (*stateful*), mecanismo descrito no Capítulo 2. Nesse contexto, preservar o estado significa que o conteúdo da memória da aplicação permanece íntegro ao longo da transferência entre nós, sem que seu código precise ser modificado.

Os trabalhos analisados a seguir foram organizados em quatro frentes complementares ao posicionamento adotado por esta dissertação, dedicadas respectivamente às primitivas de *checkpoint* e restauração, à migração em ambientes de borda, à integração com Kubernetes e aos controladores industriais conteinerizados. A síntese ao final do capítulo identifica a lacuna específica que este trabalho se propõe a preencher.

As referências foram levantadas por busca no Google Scholar, a partir de expressões como *container live migration*, *stateful container migration*, *CRIU checkpoint restore*, *edge container migration* e *task offloading*. Foram priorizados os trabalhos que tratam de migração de contêineres com preservação de estado, com ênfase nos aplicados a ambientes de borda ou industriais.

3.1 CHECKPOINT E RESTAURAÇÃO DE CONTÊINERES

O CRIU é a tecnologia central para migração com estado de processos Linux sem modificação das aplicações. Durante o *checkpoint*, o processo alvo é congelado, e seu espaço de endereçamento virtual, descritores de arquivo, conexões de rede e demais recursos do sistema operacional são serializados em um conjunto de arquivos de *checkpoint*. Em seguida, o processo pode ser encerrado. A restauração reconstrói o processo a partir desses arquivos em qualquer nó com arquitetura de hardware compatível.

Pickartz *et al.* (2016) apresentaram uma das primeiras implementações de migração de contêineres com CRIU em contexto de *High Performance Computing* (HPC). Os autores desenvolveram um *driver* para contêineres com suporte tanto a migração a frio quanto a migração ao vivo, esta última por meio do mecanismo de *page-server* do CRIU, que transmite o conteúdo de memória diretamente ao nó destino através de uma fase de pré-cópia iterativa. O trabalho demonstrou que a abordagem é tecnicamente viável e estabeleceu a base conceitual para implementações posteriores, ainda no período anterior à consolidação do containerd e do Kubernetes como padrões do ecossistema.

Tošić (2024) conduziu estudo empírico sobre a viabilidade do CRIU para migração de aplicações em tempo de execução em cenários de computação em borda. Os experimentos

mostram que o tempo de *checkpoint* é proporcional ao tamanho da memória alocada pelo contêiner, ao passo que a utilização de CPU durante *checkpoint* e restauração não escala com esse mesmo tamanho. Apesar dessa independência em relação à memória, o autor conclui que a CPU permanece como fator limitante na implantação de *checkpointing* em dispositivos de borda com recursos restritos. O estudo observa ainda que aplicações que exigem alta disponibilidade e operação contínua não se mostram candidatas viáveis ao *checkpointing*, ao passo que microsserviços e aplicações sem estado global permanecem adequados, pois a ausência de estado global reduz a memória alocada e, com ela, o impacto do congelamento do processo durante a captura de estado. Essa observação delimita o problema tratado nesta dissertação, já que as aplicações de controle industrial pertencem justamente à classe de operação contínua. Contudo, o estudo avalia estados na ordem de 1,5 GB em um cenário sem integração com orquestrador. O Capítulo 6 caracteriza a indisponibilidade em faixas de estado uma ordem de grandeza menores e decompõe sua origem por fase, o que permite delimitar quais regimes de operação industrial a técnica alcança e quais permanecem fora de seu alcance.

3.2 MIGRAÇÃO DE CONTÊINERES EM AMBIENTES DE BORDA

A migração de contêineres em ambientes de borda é problema reconhecido há quase uma década. Em Ma, Yi e Li (2017), propôs-se um sistema de transferência de contêineres Docker entre servidores de borda conforme clientes móveis se deslocam, anterior à consolidação do Kubernetes como padrão de orquestração, fora do contexto de hardware embarcado e de aplicações industriais. Os trabalhos seguintes adotam Kubernetes ou hardware restrito como pressuposto e empregam CRIU diretamente como mecanismo de preservação de estado.

Das e Sidhanta (2021) propuseram LIMOCE, algoritmo leve para migração ao vivo de contêineres em *clusters* de borda com restrição de recursos. A solução implementa transferência de estado via CRIU sem dependência de orquestradores externos e foi avaliada sob diferentes cargas e condições de rede, incluindo cenários com falhas de nós. O contexto de restrição de recursos coincide com o abordado nesta dissertação.

Calagna *et al.* (2024) propuseram COAT, solução para migração com estado de microsserviços em borda que preserva conexões TCP ativas com clientes durante a migração. A estratégia combina pré-cópia iterativa com fase final de *Stop&Copy*, empregando Podman (gerenciador de contêineres) e CRIU. Em sequência, Calagna *et al.* (2025) integraram COAT ao *framework* MOSE, que coordena a migração e seleciona automaticamente o instante ótimo para a fase final, reportando reduções de *downtime* de até 77% em comparação com as estratégias de pré-cópia iterativa e migração a frio adotadas como referência na literatura. Ambos os trabalhos operam em servidores de borda x86, sem integração com Kubernetes, e têm como alvo microsserviços de rede. Esta dissertação, por sua vez, situa-se em hardware embarcado ARM64, com integração nativa ao orquestrador.

3.3 MIGRAÇÃO COM ESTADO EM KUBERNETES

O próprio projeto Kubernetes incorporou uma primitiva de *checkpoint* de contêineres pela proposta de melhoria intitulada *Forensic Container Checkpointing* (Kubernetes SIG Node, 2024), experimental na versão 1.25 e promovida a beta na versão 1.30, quando passou a ser habilitada por padrão pelo *feature gate* `ContainerCheckpoint` (Reber, 2022). A API exposta pelo kubelet aciona a chamada `CheckpointContainer` da CRI no *runtime* de contêineres, que recorre ao CRIU para serializar o processo em disco no nó de origem, sem interromper a aplicação.

Como indica sua denominação, a proposta destina-se à análise forense, e a restauração foi dela deliberadamente excluída. O documento registra que, na primeira implementação, não se pretende oferecer suporte a restauração no kubelet, devendo esta ocorrer fora do Kubernetes (Kubernetes SIG Node, 2024). O orquestrador oferece assim a captura do estado, mas não o caminho de retorno, que exige converter manualmente o artefato em imagem OCI, publicá-la em um registro e criar um pod que a referencie. Permanecem ausentes a coordenação entre nós, a transferência do artefato e a transparência para a aplicação, elementos que distinguem um mecanismo de *checkpoint* de um sistema de migração.

O reconhecimento dessa lacuna pelo próprio projeto é recente. Em janeiro de 2026, anunciou-se a criação do grupo de trabalho Checkpoint/Restore do Kubernetes, dedicado a integrar as funcionalidades de *checkpoint* e restauração ao orquestrador e a aproximar as comunidades do Kubernetes e do CRIU (Stoyanov *et al.*, 2026). Entre os casos de uso declarados pelo grupo está a migração de pods entre nós para balanceamento de carga e manutenção, sem interrupção das cargas de trabalho, que corresponde ao problema tratado nesta dissertação. A existência do grupo confirma que essa capacidade permanece indisponível de forma nativa no orquestrador, o que motiva as propostas revisadas a seguir.

Schmidt (2023) propôs e avaliou dois operadores Kubernetes para tolerância a falhas de aplicações com estado. O primeiro operador, denominado C/R, utiliza *checkpoints* periódicos via CRIU, enquanto o segundo, denominado *Pull State*, depende de uma API própria da aplicação. A abordagem integra CRIU ao ecossistema Kubernetes sem modificação das aplicações, utilizando CRDs para gerenciar o ciclo de vida dos *checkpoints*. O escopo, porém, é a recuperação de falhas no mesmo *cluster*, sem transferência de *checkpoint* entre nós físicos distintos. Essa limitação diferencia a proposta da migração entre nós tratada nesta dissertação.

Cacciabue *et al.* (2024) abordaram o *offloading* de tarefas de robótica móvel sobre ROS2 em *clusters* Kubernetes federados, executando duas instâncias da mesma tarefa em paralelo, uma ativa no robô com k3s e outra em modo de espera na nuvem com Kubernetes padrão, e comutando entre elas por políticas de rede. A abordagem é declaradamente limitada a serviços sem estado, com migração com estado listada como trabalho futuro. Esse ponto pendente é precisamente o que se endereça nesta dissertação, no mesmo contexto de k3s em sistemas embarcados.

Poggiani *et al.* (2024) propuseram migração ao vivo de pods Kubernetes entre *clusters* distintos em sistemas *edge serverless*. O *checkpoint* é solicitado pela API de *checkpoint* do

kubelet, descrita no início desta seção, que recorre ao CRIU por meio do *runtime* de contêineres. A transferência entre *clusters* apoia-se no Liqo, ferramenta de federação leve de *clusters* Kubernetes (Iorio *et al.*, 2023), empregada para estabelecer o *peering* entre os *clusters* de origem e de destino. A solução opera em Kubernetes padrão, sem modificações de *kernel* ou da API do orquestrador. A principal distinção em relação a esta dissertação está na dependência de uma camada de federação para mover o pod entre dois *clusters*, enquanto a abordagem aqui proposta migra contêineres entre nós de um mesmo *cluster* k3s. Além disso, a avaliação de Poggiani *et al.* (2024) utiliza servidores x86, em contraste com a plataforma ARM64 embarcada adotada neste trabalho.

O trabalho mais próximo em abrangência técnica é o de Zhang *et al.* (2025), que propõe KubeSPT para migração ao vivo de pods com estado em Kubernetes entre nós distintos, combinando CRIU com pré-cópia iterativa, restauração sob demanda das páginas de memória e preservação de conexões TCP ativas durante a transferência. Os experimentos em servidores x86 com CPUs Intel Xeon, 64 GB de memória DDR4 e rede de 3 Gbps reportam reduções de *downtime* entre 86% e 93% em comparação com os métodos atuais de reagendamento do Kubernetes. A obtenção desses ganhos depende, contudo, de modificações no *kernel* Linux, e o domínio de aplicação corresponde a serviços de propósito geral em hardware servidor x86. Esta dissertação, por sua vez, atua em hardware ARM64 efetivamente embarcado, opera inteiramente em espaço de usuário por meio do shim do containerd e tem como alvo aplicações industriais.

3.4 CONTROLADORES INDUSTRIAIS CONTEINERIZADOS

Sollfrank *et al.* (2021) avaliaram empiricamente a adequação do Docker para aplicações de automação industrial distribuídas e sensíveis ao tempo em Raspberry Pi. Os resultados indicam que a conteinerização introduz sobrecarga controlada, com *jitter* dentro de limites aceitáveis para controle em tempo real brando em sistemas ciberfísicos. O trabalho sustenta a premissa de que contêineres em hardware ARM embarcado são viáveis para aplicações industriais, premissa adotada também nesta dissertação.

Koziolek, Burger e Peedikayil (2024) propuseram um mecanismo de migração ao vivo de vPLCs entre nós em *clusters* Kubernetes, no qual a transferência do estado interno da aplicação ocorre via *OPC Unified Architecture* (OPC UA) entre dois pods. Os autores reportam a transferência de 100.000 variáveis em menos de 42 ms, demonstrando a aplicabilidade do padrão de Operadores (*Operators*) do Kubernetes a sistemas de controle industrial. A proposta converge com este trabalho no uso de CRDs para orquestração, mas difere no mecanismo de transferência de estado. Em Koziolek, Burger e Peedikayil (2024), o compilador ou *runtime* do PLC deve expor uma interface OPC UA com o estado interno da aplicação, o que acopla a migração à implementação específica do controlador. A solução aqui apresentada, por outro lado, utiliza CRIU para capturar o estado do processo em nível de sistema operacional, sem modificação na aplicação ou no *runtime*.

3.5 COMPARAÇÃO E POSICIONAMENTO

O Quadro 1 sintetiza os trabalhos levantados segundo cinco dimensões relevantes para o posicionamento desta dissertação. São elas o uso de CRIU como mecanismo de preservação de estado, a integração com Kubernetes, a operação em hardware embarcado ou de borda com restrição de recursos, a migração entre nós físicos distintos e a preservação de estado de aplicação. O contexto de automação industrial não figura entre as dimensões comparadas por não constituir critério de classificação dos trabalhos, mas sim a motivação que orienta a escolha das demais dimensões, conforme discutido na Seção 3.4. A marcação *Parc.* indica atendimento parcial da dimensão, e *N/I* indica que o trabalho não informa o dado necessário para a classificação.

Quadro 1 – Comparação entre os trabalhos relacionados e a proposta desta dissertação

Trabalho	**CRIU**	**Kubernetes**	**Embarcado**	**Multinó**	**Stateful**
(Pickartz *et al.*, 2016)	Sim	Não	Não	Sim	Sim
(Tošić, 2024)	Sim	Não	N/I	Parc.	Sim
(Ma; Yi; Li, 2017)	Não	Não	Não	Sim	Sim
(Das; Sidhanta, 2021)	Sim	Não	Sim	Sim	Sim
(Calagna *et al.*, 2024)	Sim	Não	Não	Sim	Sim
(Calagna *et al.*, 2025)	Sim	Não	Não	Sim	Sim
(Schmidt, 2023)	Sim	Sim	Não	Não	Sim
(Cacciabue *et al.*, 2024)	Não	Sim	Sim	Sim	Não
(Poggiani *et al.*, 2024)	Sim	Sim	Não	Sim	Sim
(Zhang *et al.*, 2025)	Sim	Sim	Não	Sim	Sim
(Sollfrank *et al.*, 2021)	Não	Não	Sim	Não	Não
(Koziolek; Burger; Peedikayil, 2024)	Não	Sim	Não	Sim	Sim
Este trabalho	**Sim**	**Sim (k3s)**	**Sim**	**Sim**	**Sim**

Fonte: O autor (2026)

A análise do Quadro 1 revela que nenhum trabalho existente combina simultaneamente todas as dimensões abordadas nesta dissertação. Cabe registrar que essa lacuna não decorre da ausência de primitivas no orquestrador, mas do escopo com que foram concebidas, conforme discutido na Seção 3.3. Os trabalhos aqui revisados, assim como esta dissertação, constroem sobre o CRIU a camada de coordenação, transferência e restauração que o orquestrador não fornece. Em Poggiani *et al.* (2024), o CRIU é acionado pela API de *checkpoint* exposta pelo kubelet. Nas demais propostas, e também nesta dissertação, ele é acionado diretamente pelo componente responsável pela migração. Os trabalhos que utilizam CRIU em conjunto com Kubernetes (Schmidt, 2023; Poggiani *et al.*, 2024; Zhang *et al.*, 2025) operam em hardware x86 de alto desempenho. Os que operam sob restrição de recursos, por sua vez, prescindem do orquestrador, como em Das e Sidhanta (2021), ou restringem-se a serviços sem estado, como em Cacciabue *et al.* (2024). No campo da automação, Sollfrank *et al.* (2021) demonstram a viabilidade de contêineres em Raspberry Pi, mas não tratam de migração com preservação de estado, e Koziolek, Burger e Peedikayil (2024) abordam a migração de controladores industriais, porém dispensam o CRIU.

Cabe delimitar o alcance desta contribuição. O foco em aplicações industriais orientou as escolhas de projeto e a definição do ambiente experimental, mas a validação apresentada nos capítulos seguintes emprega uma aplicação de *benchmark* sintética, concebida para tornar o estado em memória controlável e a medição reprodutível. A avaliação em controladores industriais reais, sobre hardware certificado para o chão de fábrica, permanece fora do escopo desta dissertação.

3.6 SÍNTESE DO CAPÍTULO

A contribuição deste trabalho articula quatro escolhas de projeto que respondem às lacunas identificadas na seção anterior. A primeira é a opção por uma arquitetura canônica, montada a partir de componentes consolidados do ecossistema Linux e Kubernetes, em particular o shim do containerd, o CRIU e o k3s. Com isso, evitam-se novos orquestradores, camadas de federação ou *patches* de *kernel*. A segunda escolha é preservar o estado da aplicação no nível do sistema operacional, por meio do CRIU. Essa decisão dispensa modificações na aplicação, no *runtime* do controlador ou no *kernel* do sistema operacional. A terceira é a validação experimental em hardware ARM64 efetivamente embarcado, condição rara na literatura recente, na qual a maioria dos trabalhos opera em servidores x86 com folga de memória, rede e armazenamento. A quarta é a coleta sistemática de métricas, que estabelece um *baseline* reprodutível para o desempenho da migração nessa classe de hardware. Essa coleta abrange tempo de congelamento e duração de transferência sob diferentes tamanhos de carga e padrões de uso de memória, contemplando tanto cargas lineares quanto cargas com alocação aleatória.

Os mecanismos recorrentes nos trabalhos analisados e também adotados nesta dissertação foram detalhados no Capítulo 2. O próximo capítulo apresenta a modelagem e a arquitetura que os combinam.

4 MODELAGEM E ARQUITETURA PROPOSTA

Este capítulo detalha as decisões de projeto e a arquitetura do sistema desenvolvido para viabilizar a migração transparente de aplicações com estado em ambientes industriais baseados em contêineres. O objetivo central consiste em prover uma solução que permita a realocação de cargas de trabalho entre nós de processamento distintos, minimizando o tempo de indisponibilidade e preservando o contexto de execução da aplicação, como variáveis de processo e estados internos.

A arquitetura fundamenta-se na extensão das capacidades do orquestrador Kubernetes através do padrão de Operadores. O sistema integra a técnica do CRIU ao ciclo de vida dos contêineres, orquestrando a captura, transferência e restauração do estado de forma automatizada.

4.1 PRINCÍPIOS DE PROJETO

O desenvolvimento da solução foi orientado por princípios que garantem sua aplicabilidade em cenários de convergência entre TI e Tecnologia Operacional (TO). Essa convergência ocorre quando as práticas e ferramentas de implantação e gerenciamento consolidadas na TI passam a operar também sobre os equipamentos de chão de fábrica (Perez *et al.*, 2022), historicamente realizados com controladores dedicados e redes de campo proprietárias (Sollfrank *et al.*, 2021).

A transparência operacional constitui o primeiro desses princípios: a migração deve ocorrer sem a necessidade de instrumentação, recompilação ou modificação do código-fonte da aplicação industrial. Este requisito é fundamental para garantir o suporte a sistemas legados ou proprietários, nos quais o acesso ao código-fonte é frequentemente restrito. O sistema de orquestração deve, portanto, abstrair completamente a complexidade da migração em relação à aplicação hospedada.

A redução da indisponibilidade orienta as decisões de projeto, ainda que não constitua um requisito rígido a ser satisfeito. Em sistemas de controle, interrupções prolongadas podem provocar perda de sincronismo com o processo físico ou acionar mecanismos de proteção, como temporizadores de vigilância (*watchdogs*). A arquitetura busca, portanto, encurtar o intervalo entre a suspensão do processo no nó de origem e sua retomada no nó de destino, sendo a faixa efetivamente alcançada o que delimita as classes de aplicação atendidas.

A aderência aos padrões do Kubernetes determina que a solução se apoie nas APIs nativas do orquestrador, em vez de introduzir ferramentas de gerenciamento paralelas. A migração passa assim a ser operada pelas mesmas interfaces já em uso no *cluster*, como o `kubectl`, o que reduz o custo de adoção em ambientes operacionais estabelecidos.

Por fim, o desacoplamento entre mecanismo e decisão separa duas responsabilidades de escopos distintos, ambas executadas em espaço de usuário. O shim do ShiftPod opera no escopo de um contêiner, realizando a captura e a restauração do estado no nó em que reside. O manager do ShiftPod opera no escopo do *cluster*, determinando quando migrar e coordenando os nós

envolvidos. Como o shim não depende do manager para manter o contêiner em execução, uma falha na camada de coordenação não interrompe as aplicações já em operação.

4.2 COMPONENTES DA ARQUITETURA

A arquitetura compreende um conjunto de módulos de software distribuídos que interagem em diferentes camadas do sistema, desde o nível de orquestração do *cluster* até a manipulação de processos no espaço de usuário.

4.2.1 Plataforma de orquestração (k3s)

Para viabilizar a implantação em ambientes industriais, que frequentemente possuem restrições de hardware e energia, como cenários de computação de borda, optou-se pela utilização do k3s. Esta distribuição certificada do Kubernetes reduz a complexidade operacional ao empacotar os serviços essenciais em um único binário e ao admitir o SQLite como armazenamento de estado, alternativa ao etcd mais adequada às restrições de memória e de escrita em disco desses dispositivos (Rancher, 2019).

O k3s fornece a camada base de gerenciamento de recursos, sendo responsável pelo agendamento dos Pods e pela manutenção da rede entre Pods, o que permite a comunicação contínua entre as aplicações independentemente do nó em que executam.

4.2.2 Mecanismo de preservação de estado (CRIU)

Enquanto o Kubernetes soluciona nativamente a tolerância a falhas para aplicações sem estado através de reinicialização, ele carece de mecanismos para a persistência da memória volátil. O CRIU preenche esta lacuna funcional, atuando como o motor de persistência de estado.

Na arquitetura proposta, o CRIU é acionado para realizar a serialização do estado da carga de trabalho. O processo envolve a captura da árvore de processos, registradores da CPU, páginas de memória modificadas, descritores de arquivos e *buffers* de rede, persistindo-os em disco. Inversamente, a ferramenta reconstrói estes elementos para retomar a execução a partir do ponto em que foi interrompida.

Dentre os regimes de migração descritos na Seção 2.6.1, adota-se o de migração a frio, no qual o processo é congelado, o estado é integralmente serializado e transferido, e somente então restaurado no nó de destino. A escolha deve-se à sua simplicidade de implementação, uma vez que dispensa a coordenação iterativa de páginas modificadas exigida pela pré-cópia e o atendimento remoto de faltas de página exigido pela pós-cópia. Por serem estritamente sequenciais, as fases desse regime determinam a indisponibilidade percebida pela aplicação, o que fundamenta a decomposição por fase adotada na análise dos resultados.

4.2.3 Manager do ShiftPod

O manager do ShiftPod atua como o controlador primário da lógica de migração. Construído segundo o padrão de Operadores descrito na Seção 2.3.3 e implantado como um DaemonSet, este componente assegura a presença de uma instância de gerenciamento em cada nó do *cluster*, estabelecendo uma malha de controle distribuída. Cabe a ele coordenar o ciclo de vida das migrações, monitorando as requisições registradas em CRDs e ordenando a sequência de eventos entre os nós envolvidos. Cabe-lhe também gerir os artefatos de *checkpoint*, o que abrange o armazenamento local e a transferência dos arquivos de imagem. Por fim, é o manager que estabelece os canais de comunicação gRPC empregados na sincronização de estado e no transporte de dados entre nós.

4.2.4 Shim do ShiftPod

Para viabilizar a migração transparente, foi desenvolvida uma extensão ao componente shim, aqui denominada shim do ShiftPod. Na pilha de execução do nó, o kubelet delega a execução ao gerenciador de contêineres (containerd) via CRI, o containerd despacha os eventos de ciclo de vida ao shim via ttrpc, e o shim aciona o runc, executor compatível com OCI. É essa posição que permite ao shim do ShiftPod interceptar os eventos antes que cheguem ao runc. Ao receber uma requisição de criação de contêiner, ele consulta o manager local sobre a existência de estado pendente de restauração. Havendo estado disponível, a rotina padrão de inicialização é substituída pelo procedimento de restauração via CRIU. De forma análoga, ao interceptar um comando de encerramento dirigido a um contêiner marcado para migração, o shim suspende o encaminhamento do sinal e executa o procedimento de *checkpoint*, assegurando a persistência do estado antes do término do processo.

4.3 FLUXO DE MIGRAÇÃO DE CARGAS DE TRABALHO

A migração de uma aplicação compreende uma sequência coordenada de operações destinadas a preservar o estado da aplicação durante a transição entre nós. A Figura 4 ilustra a interação entre os componentes do sistema.

O processo tem início quando o orquestrador determina o encerramento de um Pod em execução, seja por políticas de balanceamento ou intervenção manual. O shim do ShiftPod no nó de origem intercepta o comando de encerramento encaminhado pelo containerd, identificando a carga de trabalho como marcada para migração, e aciona o CRIU para suspender a execução da aplicação e serializar seu estado de memória e descritores de sistema em armazenamento temporário. Concluído o *checkpoint*, o shim notifica o manager local, que assume a custódia dos artefatos gerados. O k3s, ao constatar a finalização do Pod original, agenda uma nova instância no nó de destino. Nesse nó, o shim do ShiftPod intercepta a requisição de criação e consulta o manager local sobre a disponibilidade de estado pendente. Confirmada a disponibilidade, o manager do nó de destino estabelece conexão direta com o nó de origem e realiza a transferência

dos arquivos de imagem via gRPC. Após a recepção dos dados, o shim invoca o CRIU em modo de restauração, recriando o processo com seu contexto original preservado.

A escolha do nó de destino não cabe ao ShiftPod. O Pod substituto é criado pelo controlador de ReplicaSet e posicionado pelo escalonador do k3s segundo seus critérios nativos, como recursos disponíveis, afinidades e tolerâncias, exatamente como ocorreria sem o sistema proposto. Os componentes desenvolvidos neste trabalho apenas reagem à decisão já tomada, de modo que o destino pode ser qualquer nó elegível do *cluster*, inclusive o próprio nó de origem.

A diferença em relação ao comportamento padrão do k3s está no destino dos eventos de ciclo de vida, e não na substituição do orquestrador ou na modificação da aplicação. No caminho padrão, o encerramento de um Pod encaminha um sinal ao processo, o contêiner é destruído e a memória volátil é descartada, de modo que o Pod seguinte inicia a partir da imagem, no estado definido pelo desenvolvedor, e a persistência fica a cargo da própria aplicação. Na arquitetura proposta, esses mesmos eventos são interceptados pelo shim do ShiftPod, de forma que o encerramento passa a disparar o *checkpoint* antes do término do processo, e a criação passa a disparar a restauração quando existe estado associado. O processo retoma então a execução a partir do ponto em que foi suspenso, com memória e descritores de arquivo restaurados pelo CRIU, sem que a aplicação precise implementar sincronização própria com armazenamento externo.

4.3.1 Máquina de estados da migração

A consistência do processo distribuído é assegurada por uma máquina de estados finitos, gerenciada pelo manager e refletida no CRD `ContainerMigration`. A Figura 5 apresenta os estados e transições do sistema.

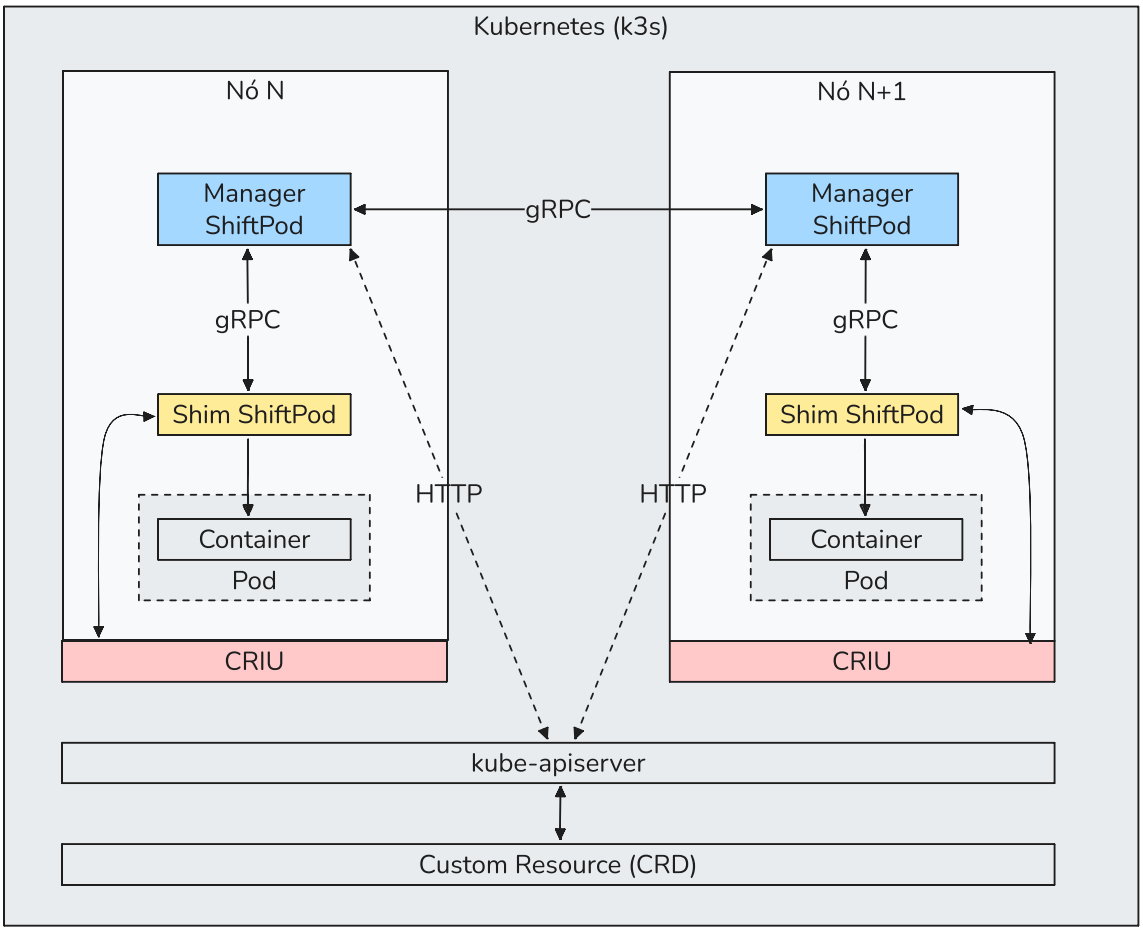
O estado inicial, denominado `Preparing`, é criado pelo manager do nó de origem no momento em que o shim intercepta o comando de encerramento, sendo registrado antes da conclusão do *checkpoint* sem bloquear essa operação. Sua função é antecipar ao nó de destino a possibilidade de uma migração em curso, evitando que o escalonador (kube-scheduler) inicie o contêiner com estado inicial antes que os artefatos estejam disponíveis, o que caracterizaria uma condição de corrida com o ciclo de vida do Pod.

Ao término bem-sucedido do *checkpoint* no shim do nó de origem, o estado transita para `Checkpointed`. O manager atualiza o CRD com o caminho onde os artefatos gerados pelo CRIU estão armazenados, registrando também métricas de duração das etapas do procedimento e o instante de congelamento do processo. A partir desse estado, o recurso torna-se disponível para ser reivindicado por qualquer nó do sistema, inclusive o nó de origem.

Quando um nó de destino reserva exclusivamente o recurso para restauração, o estado transita para `Claimed`, prevenindo condições de corrida em cenários com múltiplos candidatos. A transferência dos arquivos de imagem via gRPC ocorre durante esta fase, que se encerra quando todos os dados estão disponíveis localmente no nó de destino.

O estado `Restoring` indica que os arquivos de *checkpoint* já se encontram no nó de

Figura 4 – Arquitetura detalhada e fluxo de interação entre os componentes de migração

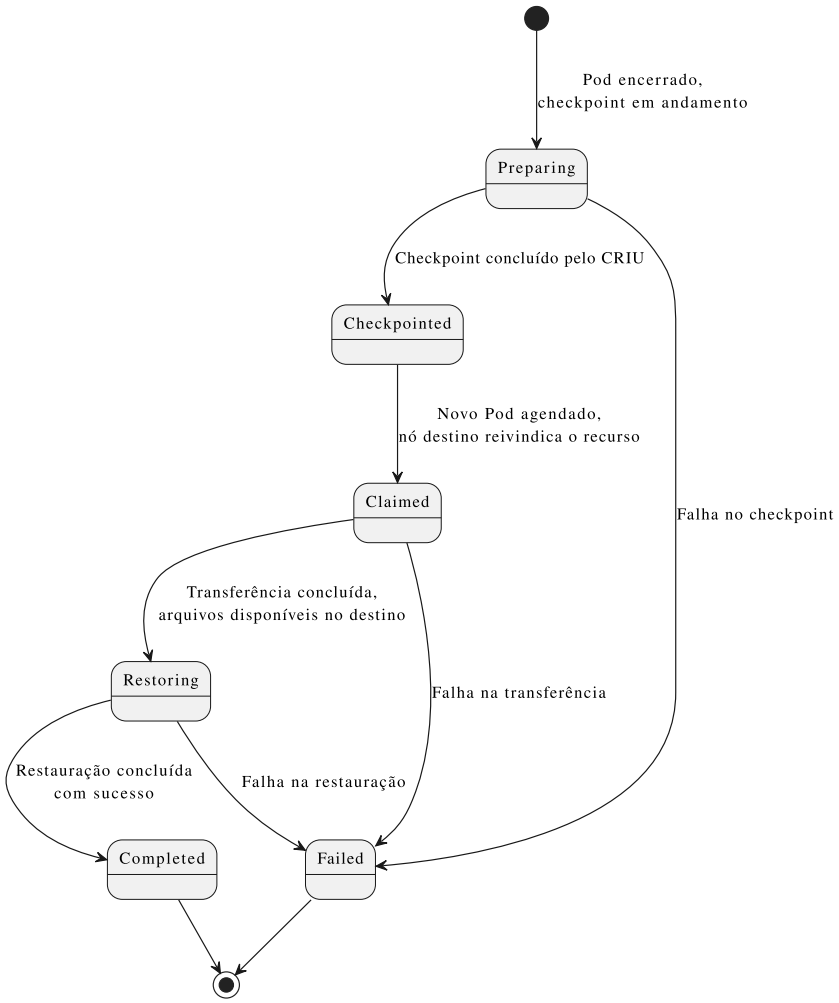


Fonte: Elaborado pelo autor (2026).

destino e que o shim invocou o executor de contêineres em modo de restauração, recriando o processo com seu contexto original. Em migrações no mesmo nó, esta fase é atingida diretamente após `Claimed`, sem transferência de dados.

O estado `Completed` sinaliza o sucesso da restauração no destino. O CRD é atualizado com métricas consolidadas de todo o processo e os artefatos intermediários são removidos. Em caso de falha durante o processo, o sistema transita para o estado `Failed`, que preserva registros do motivo do erro para fins de diagnóstico.

Figura 5 – Diagrama da máquina de estados do processo de migração



Fonte: Elaborado pelo autor (2026).

4.4 RESTRIÇÕES TÉCNICAS E PRESSUPOSTOS

A implementação de migração em nível de sistema impõe requisitos arquiteturais específicos que delimitam o escopo de aplicabilidade da solução.

A homogeneidade da arquitetura de CPU constitui a primeira restrição. Como o CRIU serializa o conteúdo dos registradores do processador, a migração só é viável entre nós da mesma família de arquitetura, condição satisfeita pela plataforma *Advanced RISC Machine* (ARM)64 homogênea adotada neste trabalho. A transferência de estado entre um nó ARM64 e um servidor x86-64 não é suportada.

A compatibilidade de versão do *kernel* representa outro pressuposto relevante. Embora o contêiner isole o espaço de usuário, ele compartilha o *kernel* do sistema hospedeiro. Pequenas diferenças de versão geralmente são toleradas, mas recursos específicos do *kernel*, como determinadas extensões de CPU ou módulos de rede, devem estar disponíveis no nó de destino para que a restauração ocorra com êxito.

Por fim, assume-se que a rede de gerenciamento dispõe de largura de banda suficiente

e baixa latência para viabilizar a transferência das imagens de memória dentro dos requisitos temporais da aplicação em execução.

4.5 SÍNTESE DO CAPÍTULO

Este capítulo apresentou a modelagem da solução proposta, partindo dos princípios de projeto para os componentes que os materializam, o fluxo de migração coordenado pela máquina de estados do recurso `ContainerMigration` e as restrições de arquitetura de CPU, de *kernel* e de rede que delimitam sua aplicabilidade.

O capítulo seguinte trata da implementação concreta dos componentes aqui descritos, e os capítulos subsequentes apresentam os experimentos que avaliam o tempo de congelamento e a duração de transferência da migração na plataforma embarcada adotada.

5 DESENVOLVIMENTO

Com base na modelagem apresentada no Capítulo 4, este capítulo detalha o desenvolvimento do sistema distribuído de migração de contêineres com estado. A implementação atende aos pressupostos ali descritos e integra-se ao Kubernetes de forma canônica, valendo-se apenas de seus mecanismos nativos de extensão.

O sistema desenvolvido neste trabalho é denominado ShiftPod. O nome reflete os dois conceitos centrais da proposta, o deslocamento de estado entre nós de processamento e a unidade de implantação do Kubernetes sobre a qual essa operação atua. O ShiftPod é composto por dois componentes implantados em cada nó do *cluster*. O shim do ShiftPod é responsável pela interceptação do ciclo de vida dos contêineres e pela criação e restauração de *checkpoints*. O manager do ShiftPod é responsável pela coordenação da migração entre nós e pelo gerenciamento do estado distribuído. Essas denominações são usadas ao longo do texto para distinguir os dois componentes desenvolvidos neste trabalho do shim padrão do containerd e dos demais controladores nativos do orquestrador. O código-fonte do ShiftPod, que inclui o shim, o manager, o instalador e os scripts de execução e análise dos experimentos, está disponível em repositório público (Andrade, 2026).

5.1 IMPLEMENTAÇÃO DO SHIM

A implementação do shim foi desenvolvida em Go, escolha fundamentada em seu alto desempenho, no gerenciamento eficiente de memória e na maturidade do ecossistema integrado ao Kubernetes.

O papel fundamental desta implementação é atuar como processo intermediário persistente no ciclo de vida dos contêineres. Diferentemente do containerd, que opera no gerenciamento de alto nível, e do runc, que atua como processo transitório na interação com o sistema operacional, o shim permanece em execução de forma individual para cada Pod.

Dado que o ecossistema utiliza o k3s, a implementação beneficia-se do containerd como *runtime* de contêineres padrão da distribuição, e em particular de sua arquitetura modular de shims. Essa modularidade permite a troca do shim padrão por uma versão customizada, possibilitando a interceptação e a modificação do comportamento de eventos padronizados entre o gerenciador (containerd) e o executor (runc), de maneira integrada ao ecossistema Kubernetes.

O shim do ShiftPod não reimplementa o shim padrão, mas o reutiliza por composição, em que o pacote Go do shim do runc é importado como biblioteca e permanece responsável pela execução efetiva de todas as operações, enquanto a camada desenvolvida neste trabalho acrescenta comportamento antes ou depois da delegação, apenas nas chamadas de interesse. Preserva-se, assim, a semântica original do ciclo de vida do contêiner.

Ao implementar um shim próprio, é possível interceptar chamadas que definem o ciclo de vida do contêiner, sobrescrevendo interfaces já estabelecidas pelo shim padrão do containerd. O evento `Create` prepara o ambiente antes da execução do runc, permitindo a injeção de

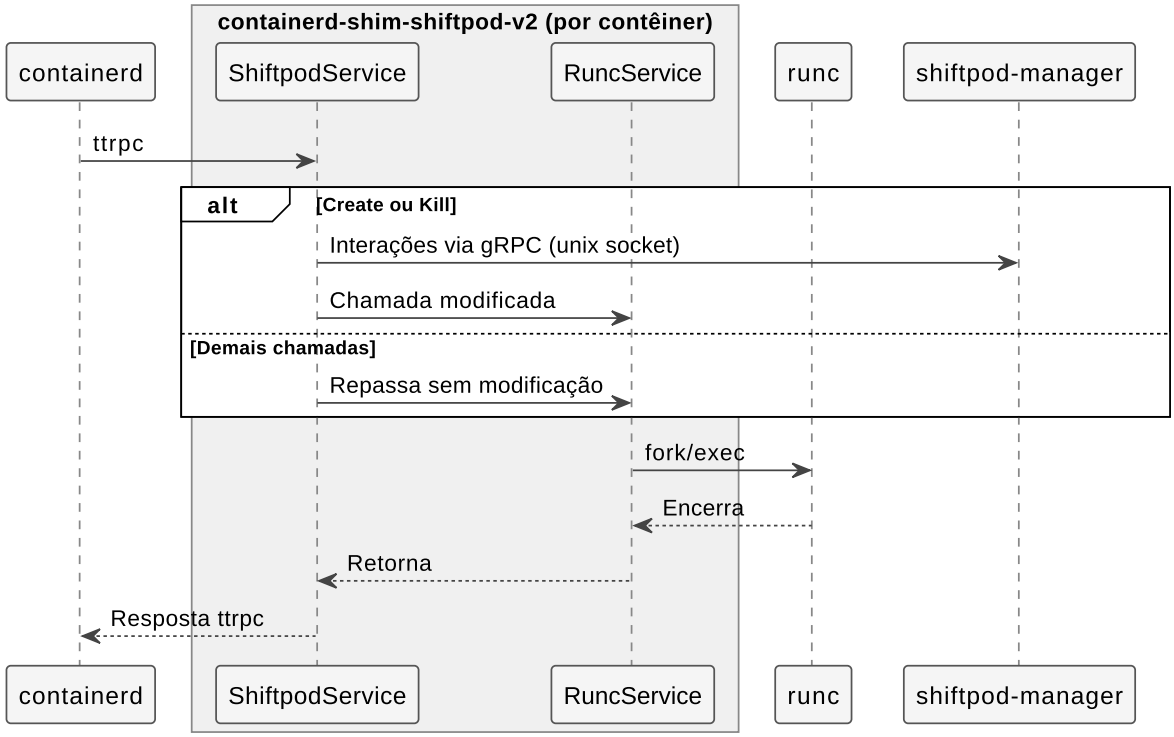
configurações adicionais de *cgroups* ou volumes. O evento `Start` é disparado no momento em que o processo do contêiner é iniciado no *namespace* isolado. O evento `Kill` recebe o sinal de encerramento enquanto o processo ainda está em execução, possibilitando processamento prévio à finalização. O evento `Exec` intercepta comandos executados em contêineres em operação, e o evento `Stats` possibilita a coleta de telemetria customizada diretamente do shim, sem dependência exclusiva das métricas padrão do containerd.

Essa abordagem permite que a solução implemente regras de negócio específicas, atuando como um *middleware* de baixo nível na infraestrutura de contêineres. Das dezessete chamadas do serviço de tarefas, apenas `Create` e `Kill` são interceptadas e condicionam o comportamento subsequente, enquanto `Start` e `Delete` recebem processamento posterior à delegação, para o registro temporal da restauração e a liberação do estado em memória. As demais são repassadas integralmente ao serviço do runc. O Apêndice B detalha o tratamento de cada chamada.

5.1.1 Integração com o ecossistema do containerd

A integração do shim do ShiftPod ao ecossistema do containerd é realizada por meio da especificação *Runtime v2* (Containerd Authors, 2024), que define as interfaces e o mecanismo de inicialização padrão para shims. O binário resultante, `containerd-shim-shiftpod-v2`, é estruturado em duas camadas que envolvem o shim do ShiftPod e o shim padrão do runc, conforme ilustrado na Figura 6.

Figura 6 – Estrutura de integração do shim do ShiftPod com o containerd



Fonte: Elaborado pelo autor (2026).

O binário expõe um servidor ttrpc ao containerd, e as requisições são interceptadas pelo `ShiftpodService`, que encapsula o `RuncService`, implementação importada da biblioteca do shim padrão do runc. Nesse fluxo, o `ShiftpodService` aplica a lógica necessária para as operações de *checkpoint* e *restore* antes de delegar a execução ao `RuncService`, que por sua vez invoca o executável `runc` via *fork/exec*.

O runc é um processo transitório, que executa a operação solicitada e encerra, propagando o resultado de volta ao containerd pela mesma cadeia.

5.1.2 Registro do shim no containerd

Para que o containerd reconheça e utilize o shim do ShiftPod, duas configurações devem ser realizadas no *cluster*: o registro do *runtime* no arquivo de configuração do containerd e a criação de um recurso RuntimeClass (Seção 2.3.2) que associa esse *runtime* aos Pods que devem utilizá-lo.

No contexto do k3s, a configuração do containerd é gerida pelo arquivo de modelo `/var/lib/rancher/k3s/agent/etc/containerd/config.toml.tmpl`, que estende o modelo base do k3s. A declaração do *runtime* customizado deve ser acrescentada a esse arquivo em cada nó do *cluster*, conforme o Código 5.1.

Código 5.1 – Declaração do *runtime* ShiftPod no containerd (k3s)

```
{{ template "base" . }}

[plugins."io.containerd.grpc.v1.cri".containerd.runtimes."
   shiftpod"]
  runtime_type = "io.containerd.shiftpod.v2"
  pod_annotations = [
    "shiftpod/enable-migration"
  ]
```

O containerd busca o shim no `PATH` com base no `runtime_type`. No caso do Código 5.1, requer o binário `containerd-shim-shiftpod-v2` no sistema (geralmente em `/usr/local/bin/`).

O campo `pod_annotations` declara quais anotações do Pod devem ser propagadas pela CRI até a especificação OCI do contêiner, tornando-as acessíveis ao shim no momento do `Create`. A anotação listada, `shiftpod/enable-migration`, é a mesma reconhecida pelo shim para habilitação do comportamento de migração. Sem essa declaração na configuração do containerd, a anotação presente no manifesto do Pod não chegaria ao shim, e o comportamento de migração permaneceria inativo para todos os contêineres.

A segunda configuração é apresentada no Código 5.2. O campo `handler` é o identificador que o Kubernetes utiliza para selecionar a entrada correspondente na configuração do containerd, e ao receber uma requisição de criação de Pod com `runtimeClassName: shiftpod-class`, o

Kubernetes instrui o kubelet a utilizar o *runtime* identificado como `shiftpod`, que por sua vez invoca o binário `containerd-shim-shiftpod-v2`.

Código 5.2 – Recurso RuntimeClass do ShiftPod

```
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  name: shiftpod-class
handler: shiftpod
```

5.1.3 Estado em memória

No modelo *Runtime v2* do containerd, um único processo de shim é instanciado por Pod, e não por contêiner. Isso implica que o mesmo processo recebe eventos de ciclo de vida para todos os contêineres pertencentes àquele Pod: o contêiner *sandbox*, responsável por estabelecer o *namespace* de rede compartilhado pelo Pod no modelo CRI, e os contêineres de aplicação declarados na especificação do recurso Kubernetes. Essa arquitetura exige que o shim mantenha estado indexado por identificador de contêiner, de modo a correlacionar corretamente os eventos de `Create`, `Start` e `Kill` de cada contêiner de forma independente.

O shim mantém em memória um conjunto mínimo de informações necessárias para essa correlação. Essas informações são organizadas na estrutura `ShiftpodService`, que centraliza o estado compartilhado entre as chamadas recebidas pelo servidor ttrpc.

O componente principal desse estado é um mapa de contêineres (*hashmap*), indexado pelo identificador de contêiner. Para cada contêiner gerenciado pelo shim, são registradas a configuração extraída das anotações do Pod, o caminho do diretório de *checkpoint* calculado a partir do *User Identifier* (UID) do Pod e as informações de rastreamento da restauração. Estas últimas compreendem um sinalizador que indica se o contêiner foi iniciado a partir de um *checkpoint*, o instante em que a restauração teve início e o identificador do contêiner de origem. Esses campos são preenchidos durante o `Create`, quando um *checkpoint* é localizado, e consumidos durante o `Start`, quando o contêiner retoma a execução e o tempo total de restauração é calculado e reportado ao manager.

Adicionalmente, um mapa efêmero armazena o nome do recurso `ContainerMigration` obtido junto ao manager no início do processo de *checkpoint*. Esse nome é necessário na notificação final, enviada após a conclusão da captura de estado, e é mantido neste mapa porque a obtenção e o uso ocorrem em contextos de execução distintos: uma *thread* assíncrona realiza a consulta enquanto o CRIU executa a captura em primeiro plano. A entrada correspondente é removida do mapa imediatamente após o uso, de modo que ele permanece vazio durante a operação normal do sistema.

O acesso ao mapa de contêineres é protegido por um *mutex* dedicado, enquanto um segundo *mutex* serializa as operações de *checkpoint* e restauração, impedindo que um `Kill` inicie

uma nova captura enquanto um `Create` ainda está processando a restauração de outro contêiner no mesmo processo de shim.

5.1.4 Processo de *checkpoint*

O processo de *checkpoint* é acionado pela interceptação da chamada `Kill`, disparada pelo containerd quando o Kubernetes solicita o encerramento de um Pod. Ao receber essa chamada, o shim verifica inicialmente se o *checkpoint* está habilitado para o contêiner em questão, consultando a configuração extraída das anotações do Pod. Caso o sinal recebido seja `SIGKILL`, o *checkpoint* é omitido e o sinal é repassado diretamente ao runc, pois esse sinal não pode ser interceptado pelo processo-alvo e não permite o intervalo necessário para a captura de estado. Apenas ao receber `SIGTERM`, sinal com que o Kubernetes inicia o período de encerramento gracioso do Pod, e confirmando junto ao runc que o contêiner ainda está em execução, o shim prossegue com o procedimento de *checkpoint*.

O shim identifica, pela anotação do Pod, se o contêiner está habilitado para a captura, e separa, pelo sinal recebido, o encerramento forçado do gracioso. O que não lhe é acessível é o desfecho deste último. A interface do containerd não indica se o Pod será reagendado em outro nó ou removido em definitivo, e essa decisão é tomada pelo plano de controle depois que o sinal já foi emitido. Todo encerramento gracioso de um Pod habilitado produz, assim, um *checkpoint*, ainda que nenhum Pod substituto venha a reivindicá-lo.

Antes de acionar o CRIU, o shim coleta métricas de estado do contêiner diretamente do sistema operacional. O uso de memória é obtido a partir da hierarquia de *cgroups* do contêiner, enquanto a contagem de processos e *threads* é lida de `/proc/<pid>/task`. A quantidade de descritores de arquivo abertos é obtida via `/proc/<pid>/fd`, e as conexões TCP ativas no *namespace* de rede são consultadas em `/proc/<pid>/net/tcp`. Nesses caminhos, `<pid>` designa o PID do processo inicial do contêiner, expresso na numeração do nó e não na do *namespace* isolado, obtido junto ao runc pela chamada `State`. Essas informações são enviadas posteriormente como parte dos metadados do *checkpoint*.

Uma decisão de projeto crítica diz respeito ao momento de criação do CRD de migração no Kubernetes. Em uma implementação direta, esse recurso seria criado após a conclusão do *checkpoint*, quando o manager já dispõe dos metadados completos. Contudo, essa abordagem introduz uma condição de corrida. A solicitação de encerramento que dá origem ao `SIGTERM` registra no servidor de API o instante a partir do qual o Pod é considerado em terminação (Kubernetes Authors, 2025b), e o Kubernetes deixa de contabilizá-lo entre as réplicas ativas nesse mesmo instante, ainda que o processo permaneça em execução durante todo o período de encerramento gracioso (Kubernetes Authors, 2026). Constatada a diferença em relação ao número desejado de réplicas, o controlador de ReplicaSet agenda imediatamente um Pod substituto (Kubernetes Authors, 2025c). O novo Pod é iniciado no nó destino enquanto o *checkpoint* ainda está em andamento no nó fonte, e o shim desse nó consulta o manager em busca de um CRD correspondente antes mesmo de ele ter sido criado. Como o *checkpoint* demora

tipicamente cerca de 800 ms, a janela de corrida é suficientemente longa para que a consulta do nó destino ocorra antes da criação do CRD, impedindo a restauração de estado.

Para eliminar essa janela, o shim notifica o manager de forma assíncrona antes mesmo de solicitar ao runc a captura de estado do contêiner. Essa notificação, denominada `PrepareCheckpoint`, instrui o manager a criar o CRD imediatamente com fase `Preparing`, sinalizando que um *checkpoint* está em andamento. A criação do CRD ocorre em paralelo com a captura de estado realizada pelo CRIU, operação de duração significativamente maior, conforme ilustrado na Figura 7.

O *checkpoint* é solicitado ao runc por meio de sua interface nativa de captura de estado, que aciona o CRIU internamente. O CRIU congela o processo e captura as páginas de memória, os descritores de arquivo e a árvore de processos, gravando os arquivos de imagem em diretório alocado em *tmpfs* para maximizar o desempenho de escrita e garantir a limpeza automática em caso de reinicialização do nó.

Após a conclusão bem-sucedida da captura de estado, o shim envia a notificação `NotifyCheckpoint` ao manager com os metadados finais (duração do *checkpoint*, tamanho do diretório gerado e métricas de estado coletadas anteriormente), que atualiza o CRD para a fase `Checkpointed`, liberando o nó destino para iniciar a transferência e a restauração. O shim então repassa o sinal `SIGTERM` original ao runc, permitindo que o ciclo de vida do contêiner prossiga normalmente sob a perspectiva do containerd.

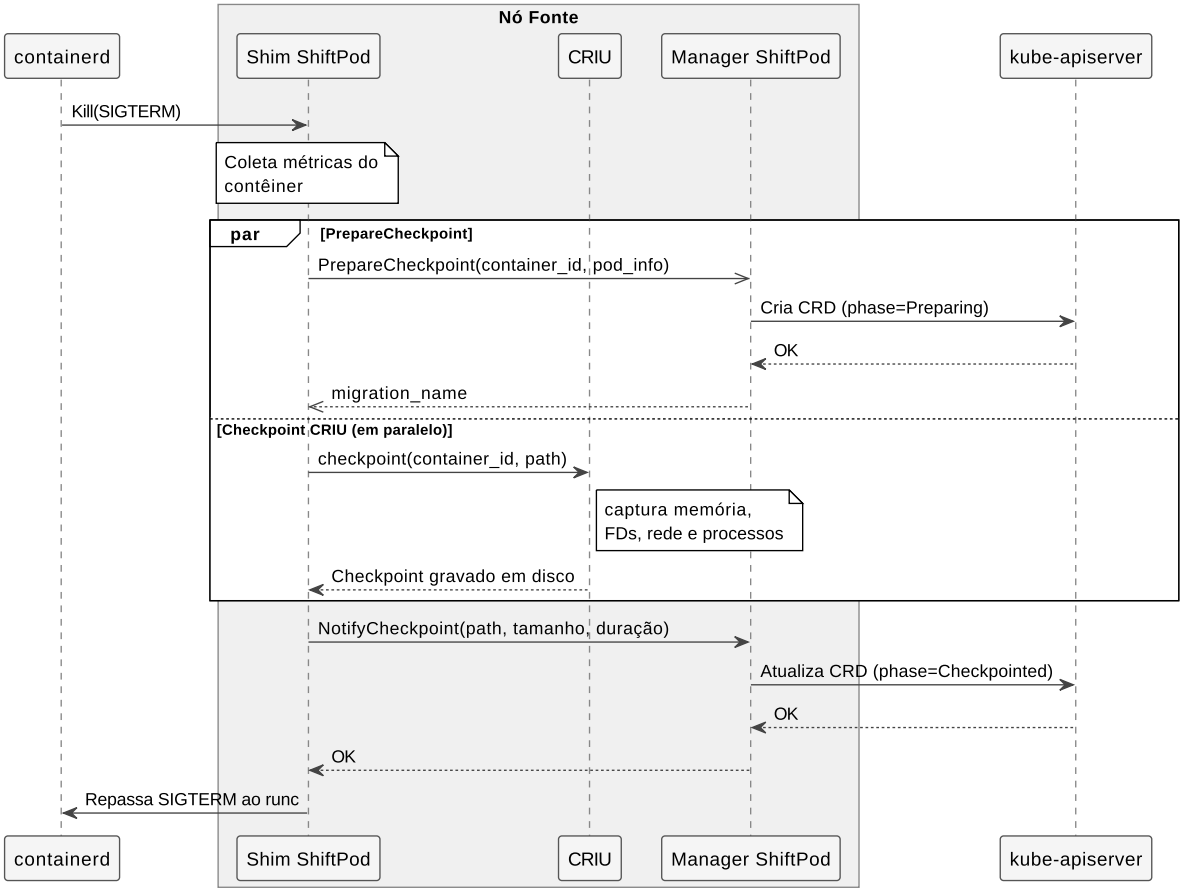
5.1.5 Processo de restauração

A restauração de um contêiner a partir de *checkpoint* é iniciada pela interceptação da chamada `Create`. Ao receber essa chamada, o shim lê o arquivo `config.json`, que contém a especificação OCI do contêiner, para extrair as anotações e identificar seu tipo. Contêineres do tipo *sandbox*, responsáveis pela infraestrutura de rede do Pod no modelo CRI, são tratados de forma transparente, com a chamada repassada diretamente ao runc sem nenhuma lógica adicional.

Para contêineres de aplicação com migração habilitada, o shim consulta o manager via soquete Unix, informando o nome do Pod, o nome do contêiner e o *namespace* Kubernetes. O manager busca um CRD `ContainerMigration` compatível com o Pod em questão. Caso nenhum CRD seja encontrado após as tentativas de busca, o manager responde sinalizando ausência de migração e o shim repassa a chamada `Create` diretamente ao runc, iniciando o contêiner do zero sem qualquer intervenção. Esse retorno ao fluxo padrão de criação garante que a lógica de restauração seja completamente transparente para o containerd e para o Kubernetes.

Quando um CRD é localizado, pode ocorrer que ele ainda esteja na fase `Preparing`, indicando que o *checkpoint* no nó de origem ainda não foi concluído. Nesse caso, o manager aguarda ativamente a transição para a fase `Checkpointed` antes de prosseguir, consultando periodicamente o CRD até que o estado esperado seja atingido ou o tempo limite seja excedido. Uma vez confirmado o *checkpoint*, o manager reivindica o CRD e obtém do nó de origem os arquivos correspondentes. Essa obtenção é representada de forma condensada na Figura 8, por

Figura 7 – Fluxo do processo de *checkpoint* no nó fonte, do ponto de vista do shim



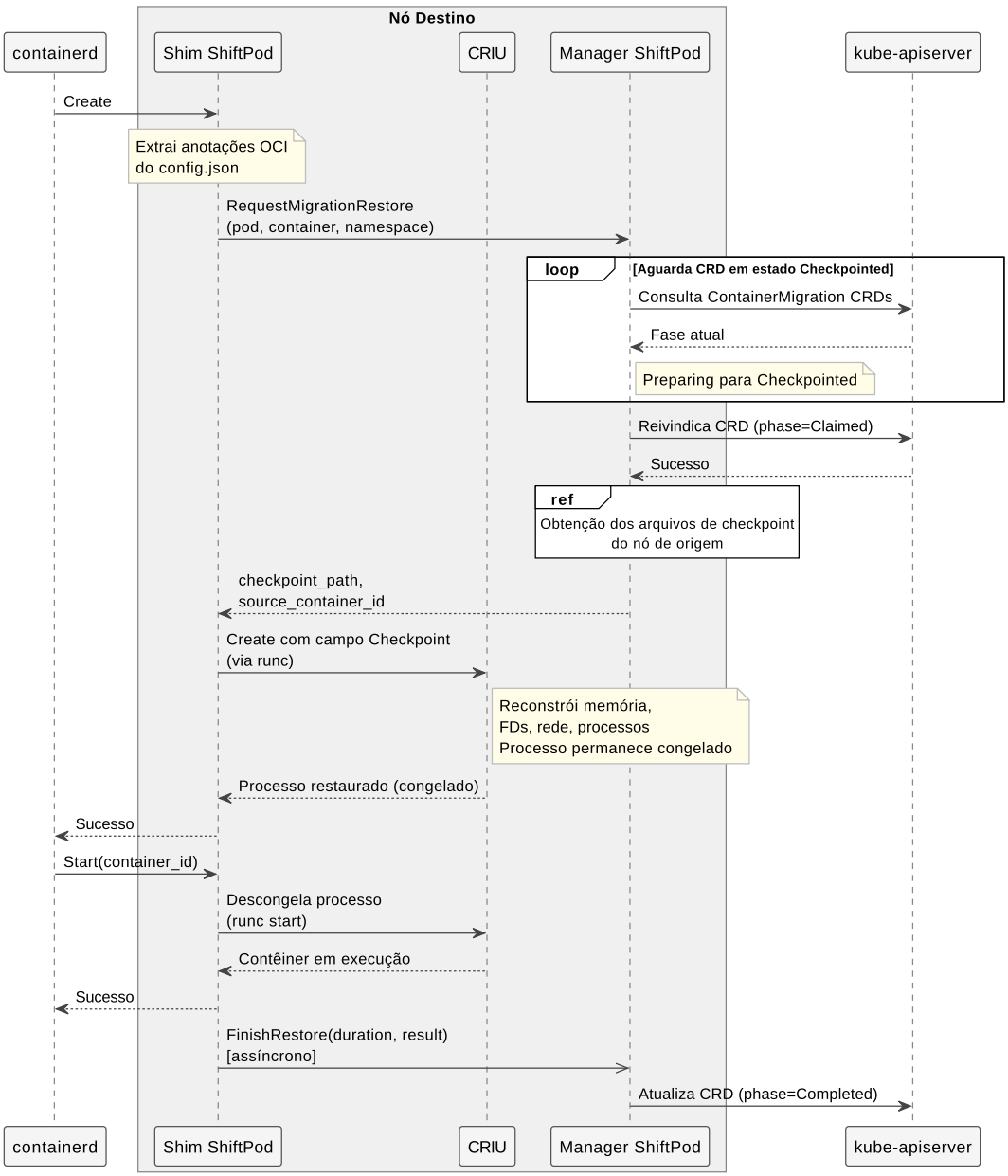
Fonte: Elaborado pelo autor (2026).

não pertencer ao ponto de vista do shim, e é detalhada na Seção 5.2.3, que trata da coordenação pelo manager. Concluída essa etapa, o manager devolve ao shim o caminho do diretório local de *checkpoint* e o identificador do contêiner de origem.

Com essa informação, o shim constrói uma requisição `CreateTaskRequest` incluindo o campo `Checkpoint`, instruindo o runc a invocar o CRIU no modo de restauração. Nessa etapa, o CRIU reconstrói o estado do processo, incluindo páginas de memória e descritores de arquivo, deixando-o em estado congelado, sem que a execução seja retomada imediatamente.

A retomada da execução ocorre apenas na chamada subsequente `Start`, quando o containerd instrui o shim a iniciar o processo. Nesse momento, o contêiner é descongelado e volta a executar a partir do estado capturado no nó de origem. O shim registra o instante de descongelamento e calcula a duração total da restauração, notificando o manager com esses dados temporais e o resultado da operação por meio da chamada `FinishRestore`, que é processada de forma assíncrona, encerrando o fluxo ilustrado na Figura 8.

Figura 8 – Fluxo do processo de restauração no nó de destino, do ponto de vista do shim



Fonte: Elaborado pelo autor (2026).

5.1.6 Configuração e habilitação da migração

O shim determina o comportamento de migração de cada contêiner por meio de anotações presentes na especificação OCI do Pod, propagadas pelo containerd no momento da criação. A anotação `shiftpod/enable-migration` habilita o comportamento completo de migração, em que o shim realiza o *checkpoint* no encerramento e consulta o manager em busca de *checkpoint* disponível na criação. Na ausência dela, o contêiner segue o ciclo de vida padrão do containerd.

Além dessa anotação, o shim lê anotações padrão do CRI do Kubernetes para extrair o contexto de execução. São eles: o tipo do contêiner, que distingue o *sandbox* da aplicação, o nome e o *namespace* do Pod e o identificador único do Pod, utilizado como nome do diretório de

checkpoint. O sistema suporta atualmente um único contêiner de aplicação por Pod.

Do ponto de vista de quem implanta a aplicação, a habilitação da migração para um Pod requer apenas a inclusão da anotação na especificação do recurso Kubernetes, sem necessidade de alteração na imagem do contêiner ou em configurações do nó. O campo `runtimeClassName` instrui o Kubernetes a utilizar a RuntimeClass `shiftpod-class`, que associa o contêiner ao shim do ShiftPod em vez do shim padrão do containerd, conforme o Código 5.3.

Código 5.3 – Exemplo de manifesto de Pod com migração habilitada

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    shiftpod/enable-migration: "true"
spec:
  runtimeClassName: shiftpod-class
```

5.2 IMPLEMENTAÇÃO DO MANAGER E DO CRD

O componente de coordenação apresentado no Capítulo 4 é o manager do ShiftPod. Ele realiza o padrão de Operadores do ecossistema Kubernetes, que combina um CRD com um controlador de reconciliação (Seção 2.3.3). A distinção entre os dois termos é de nível de descrição. Operador designa o padrão arquitetural, ao passo que manager nomeia o artefato de software que o realiza neste trabalho. Por concisão, emprega-se também CRD tanto para a definição do tipo, registrada uma única vez na instalação do sistema, quanto para as instâncias criadas a cada migração, já que o contexto distingue os dois usos.

O manager do ShiftPod é implantado como DaemonSet no *cluster* k3s, garantindo que uma instância esteja presente em todos os nós participantes da migração. A escolha de Rust como linguagem de implementação fundamenta-se nas garantias de segurança de memória, no consumo reduzido de recursos em tempo de execução e no desempenho previsível, propriedades especialmente adequadas ao contexto de dispositivos embarcados com recursos limitados. O componente opera dois servidores gRPC concorrentes, gerenciados por um *runtime* assíncrono, que expõem o mesmo serviço sobre transportes distintos, um soquete Unix local para comunicação com o shim do mesmo nó e uma porta TCP para comunicação entre managers em nós distintos.

5.2.1 Definição do recurso customizado (CRD)

O recurso ContainerMigration não é nativo do Kubernetes. Trata-se de um tipo definido nesta dissertação e registrado no *cluster* durante a instalação, pelo mecanismo de extensão descrito na Seção 2.3.3, passando a ser tratado pelo servidor de API como os tipos nativos. Ele

constitui o mecanismo central de coordenação da migração no ShiftPod. Ao utilizar a API do Kubernetes como base de dados distribuída, elimina-se a necessidade de um banco de dados externo para coordenação entre nós, aproveitando os mecanismos nativos de consistência e observabilidade do *cluster*. O manager utiliza a biblioteca kube-rs (kube-rs Authors, 2024) para criar, atualizar e monitorar esses recursos. Os campos de especificação do recurso estão descritos na Tabela 1.

O campo `pod_template_hash` merece atenção especial. Ao criar um Deployment, o Kubernetes gera internamente um ReplicaSet responsável por manter o número desejado de réplicas. O ReplicaSet adiciona automaticamente o rótulo (*label*) `pod-template-hash` a todos os Pods que cria, derivado do *hash* do modelo definido no Deployment. Como o Pod substituto é criado pelo mesmo ReplicaSet, ele recebe o mesmo valor de *hash* que o Pod original. O sistema aproveita essa propriedade como mecanismo de correlação: ao iniciar uma restauração, o manager consulta o rótulo do novo Pod via API do Kubernetes e usa o *hash* para localizar o recurso de migração correspondente, sem necessidade de comunicação direta entre os nós.

Tabela 1 – Campos de especificação do recurso ContainerMigration

Campo	**Descrição**
`pod_template_hash`	*Hash* do modelo do Pod. Usado pelo nó destino para localizar o CRD de migração correspondente
`source_node`	Nó de origem da migração
`source_pod`	Pod de origem
`target_node`	Nó destino. Preenchido durante a restauração
`target_pod`	Pod destino. Preenchido durante a restauração
`containers`	Lista de contêineres a migrar. Possui os campos `name`, `id`, `image_server` (endereço do manager do nó fonte, utilizado pelo nó destino para requisitar a transferência) e `checkpoint_path`

Fonte: Elaborado pelo autor (2026).

O status do recurso é controlado pelo campo `phase`, que define a fase atual da migração conforme o ciclo de vida descrito no Capítulo 4. As fases estão descritas na Tabela 2.

Tabela 2 – Fases do ciclo de vida do recurso ContainerMigration

Fase	**Significado**
`Preparing`	Inicialização. *Checkpoint* em andamento no nó fonte
`Checkpointed`	*Checkpoint* concluído e disponível
`Claimed`	Nó destino reservou o recurso. Transferência em andamento
`Restoring`	Arquivos recebidos. *Restore* em andamento
`Completed`	Migração concluída com sucesso
`Failed`	Migração encerrada com falha

Fonte: Elaborado pelo autor (2026).

Além da fase, o status acumula métricas ao longo do ciclo de migração. Todos os campos são opcionais e omitidos da serialização quando não preenchidos, conforme a Tabela 3.

Tabela 3 – Métricas de status do recurso ContainerMigration

Campo	Descrição
`checkpoint_duration_ms`	Duração da captura de estado pelo CRIU em milissegundos
`checkpoint_size_bytes`	Tamanho total do diretório de *checkpoint*
`freeze_timestamp_ms`	Instante de congelamento do contêiner no nó fonte
`memory_usage_bytes`	Uso de memória do contêiner antes do congelamento
`process_count`	Número de processos e *threads* ativos
`open_files_count`	Descritores de arquivo abertos
`tcp_connections_count`	Conexões TCP ativas no *namespace* de rede
`transfer_duration_ms`	Duração total da transferência de *checkpoint* entre os nós
`compressed_size_bytes`	Volume de dados transferido após compressão
`restore_duration_ms`	Duração da restauração no nó destino
`total_duration_ms`	Soma das durações de *checkpoint*, transferência e restauração
`total_downtime_ms`	Intervalo entre o congelamento no nó fonte e a retomada no nó destino

Fonte: Elaborado pelo autor (2026).

Um exemplo completo do recurso em estado final é apresentado no Apêndice A.

5.2.2 Interface de comunicação

O manager expõe um serviço gRPC definido pelo contrato `ManagerService`, acessível por dois canais distintos. O canal local utiliza um soquete Unix, por onde o shim do mesmo nó envia notificações de *checkpoint* e solicita coordenação de restauração. O canal externo utiliza TCP na porta 9090, por onde managers de outros nós requisitam a transferência do *checkpoint*. Ambos os canais compartilham a mesma definição de serviço em Protocol Buffers, de modo que o manager não distingue a origem da chamada pelo protocolo, apenas pelo contexto da mensagem recebida.

A transferência de *checkpoint* entre managers utiliza *streaming* do gRPC, em que o manager do nó de destino abre uma chamada `PullImage` com o manager fonte, que responde com uma sequência de blocos comprimidos. Esse modelo evita materializar o arquivo completo em memória e permite iniciar a descompressão no destino enquanto o envio ainda está em andamento.

A Tabela 4 descreve as mensagens disponíveis no serviço, organizadas por fase do ciclo de migração.

Tabela 4 – Mensagens do serviço gRPC do manager

Mensagem	Descrição
`PrepareCheckpoint`	Sinaliza ao manager que um *checkpoint* está prestes a iniciar. O manager cria o CRD em fase `Preparing`, garantindo sua disponibilidade antes da conclusão da captura de estado
`NotifyCheckpoint`	Notifica a conclusão da captura de estado com caminho, duração, tamanho e métricas de estado. O manager atualiza o CRD para `Checkpointed`
`RequestMigrationRestore`	Solicita ao manager que localize um CRD compatível, aguarde o *checkpoint* e o reivindique. Retorna o caminho local do *checkpoint* e o identificador do contêiner de origem
`PullImage`	Transfere o diretório de *checkpoint* do manager fonte para o manager destino em *streaming* comprimido
`FinishRestore`	Notifica a conclusão da restauração com duração e resultado. O manager consolida as métricas, atualiza o CRD para `Completed` ou `Failed` e remove os arquivos locais

Fonte: Elaborado pelo autor (2026).

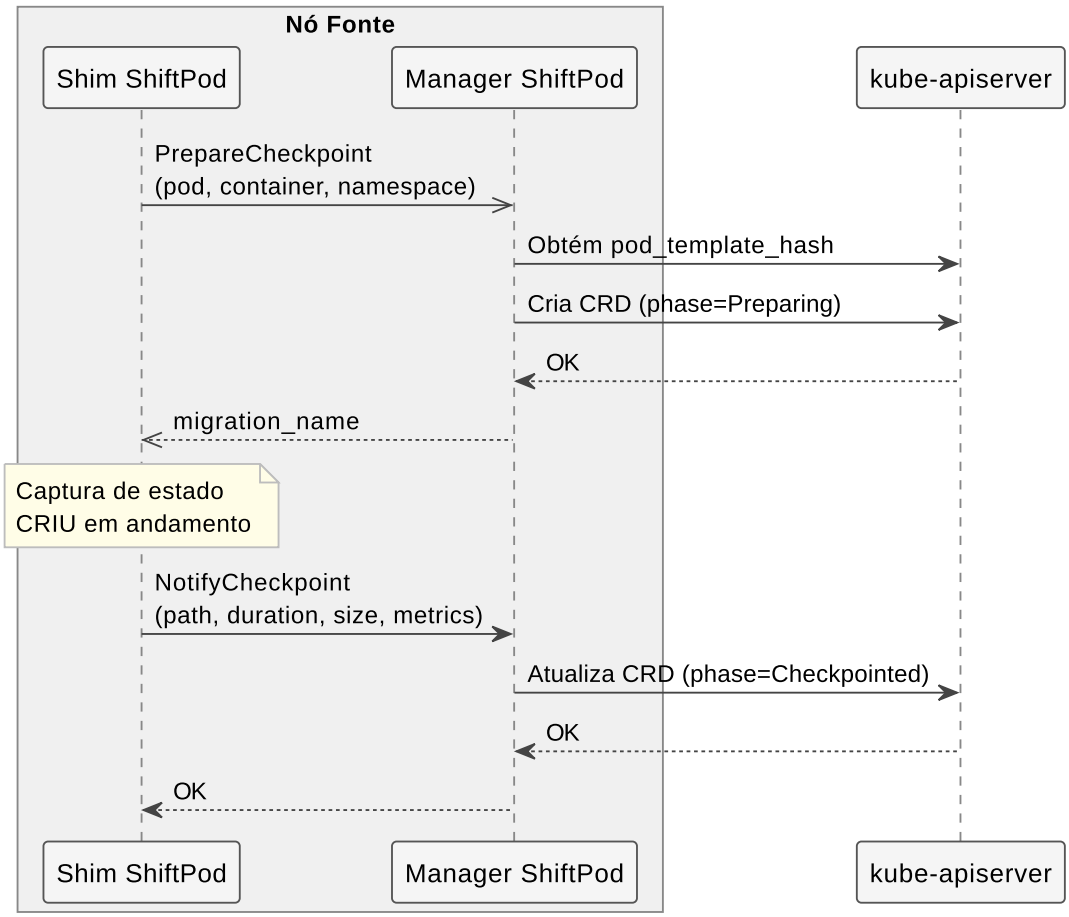
5.2.3 Coordenação da migração pelo manager

No nó fonte, ao receber a mensagem `PrepareCheckpoint`, o manager consulta a API do k3s para obter o *hash* do Pod e cria o CRD imediatamente em fase `Preparing`. Ao receber `NotifyCheckpoint`, enviada pelo shim após a conclusão da captura de estado, o manager atualiza o CRD para `Checkpointed` com as métricas coletadas. O fluxo está ilustrado na Figura 9.

No nó destino, ao receber `RequestMigrationRestore`, o manager obtém o *hash* do novo Pod e busca um CRD compatível por meio de consultas com recuo exponencial (*exponential backoff*), atendendo à janela entre a criação do Pod e a publicação do CRD pelo nó fonte. São elegíveis apenas os recursos ainda não reivindicados por outro nó, e a busca é abandonada após o número máximo de tentativas, caso em que o contêiner é criado do zero. Se o CRD selecionado estiver em fase `Preparing`, o manager observa o recurso na API do k3s até a transição para `Checkpointed`, respeitando um limite de cinco segundos.

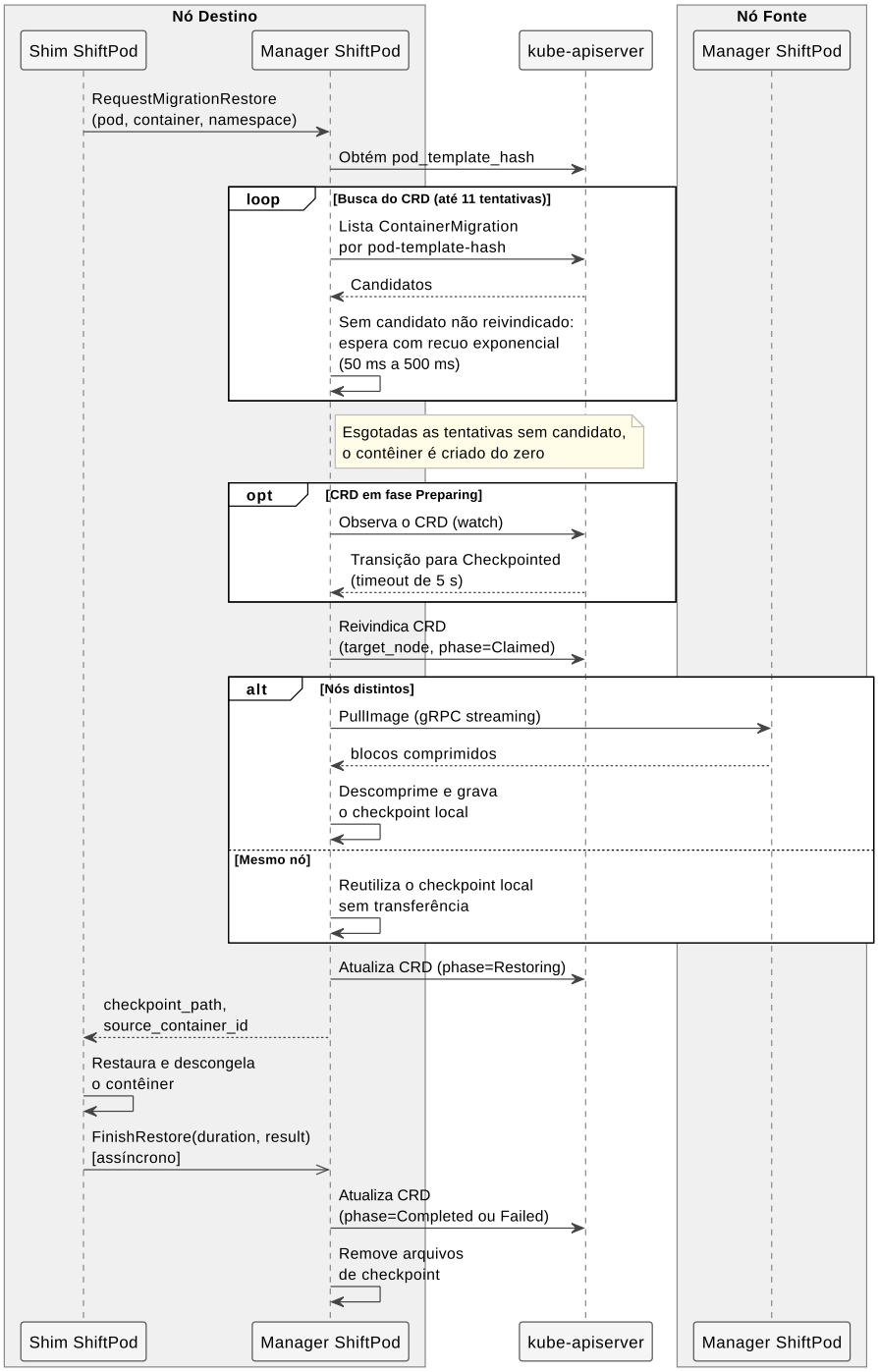
A reivindicação é feita diretamente na API do k3s, com o preenchimento dos campos `target_node` e `target_pod` e a transição para `Claimed`. A operação é idempotente, de modo que se outro nó já tiver reivindicado o mesmo CRD, a operação retorna indicando ausência de migração disponível para o nó atual. Concluída a reivindicação, o *checkpoint* é transferido em *streaming* do manager fonte, ou utilizado diretamente se a migração ocorrer no mesmo nó. Após o contêiner descongelar, `FinishRestore` consolida as métricas e fecha o CRD. O fluxo está ilustrado na Figura 10.

Figura 9 – Coordenação do manager no nó fonte durante o *checkpoint*



Fonte: Elaborado pelo autor (2026).

Figura 10 – Coordenação do manager no nó destino durante a restauração



Fonte: Elaborado pelo autor (2026).

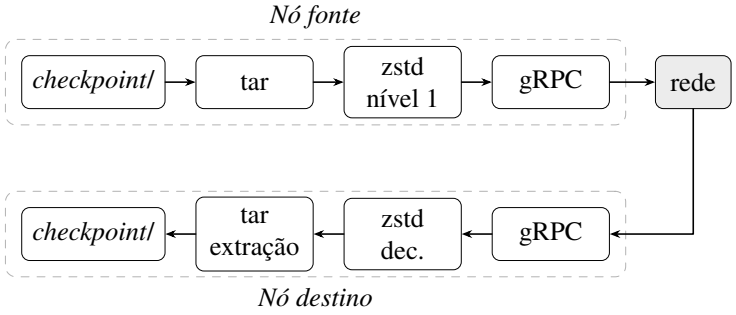
5.2.4 Transferência de *checkpoint* entre nós

A transferência do *checkpoint* entre os nós do sistema, orquestrada pelo manager, emprega um *pipeline* de transmissão em fluxo contínuo (*streaming*) que evita a criação de arquivos temporários, reduzindo o uso de disco e a latência total da operação.

No nó fonte, o diretório de *checkpoint* é empacotado em formato tar, consolidando os múltiplos arquivos de imagem gerados pelo CRIU em um único fluxo de dados contínuo para transmissão. Quando a compressão está habilitada, esse fluxo é processado antes do envio por um codificador Zstandard (zstd), algoritmo de compressão sem perdas voltado a alta velocidade de codificação (Collet; Kucherawy, 2021). O esforço de codificação é ajustável em uma escala de 1 a 22, do mais rápido ao de maior taxa de compressão, e adotou-se o nível 1, priorizando velocidade em detrimento da taxa. Os dados resultantes são segmentados em blocos de tamanho configurável pelo parâmetro `CHUNK_SIZE_KB` (padrão de 512 KB) e transmitidos como mensagens gRPC ao manager do nó destino. O primeiro bloco sinaliza o modo de compressão adotado. Ao término da transmissão, o diretório de *checkpoint* é removido automaticamente do nó fonte.

No nó destino, os blocos recebidos são encaminhados a um decodificador que, caso a compressão esteja ativa, aplica a descompressão zstd antes de extrair o conteúdo diretamente no diretório de destino em disco. O fluxo completo opera sem materializar qualquer arquivo intermediário, conforme ilustrado na Figura 11.

Figura 11 – Pipeline de transmissão em fluxo contínuo do *checkpoint* entre os nós



Fonte: Elaborado pelo autor (2026).

A adoção de zstd nível 1 justifica-se pelo perfil do ambiente experimental. O enlace Gigabit Ethernet opera entre 70 e 85 MB/s em transferências grandes, conforme medido no Capítulo 6, e constitui o gargalo predominante, enquanto a CPU dos Raspberry Pi permanece ociosa. Nesse cenário, a compressão reduz o volume transmitido e o tempo de transferência, desde que o estado apresente padrões repetitivos, como variáveis de processo e tabelas de controle. Para estados de alta entropia, isto é, cujo conteúdo não apresenta redundância explorável pela codificação (Shannon, 1948), como sequências pseudoaleatórias, a taxa aproxima-se da unidade e a codificação deixa de compensar, caso em que a compressão pode ser desabilitada pelo parâmetro `COMPRESS`.

6 TESTES E RESULTADOS

Este capítulo apresenta a avaliação experimental do ShiftPod em um ambiente embarcado composto por dois nós Raspberry Pi 4. A análise é orientada pelas seguintes questões experimentais, cujas respostas são consolidadas na Seção 6.9:

a) Como o tamanho do estado da aplicação afeta a duração de cada fase da migração?
b) Em que condições a compressão zstd L1 reduz o tempo de transferência de forma relevante?
c) Quais os limiares de indisponibilidade produzidos pelo sistema nos cenários avaliados, e o que implicam para a viabilidade da migração em contexto embarcado?
d) Qual a diferença de desempenho entre a migração entre nós e a restauração no mesmo nó, e o que essa diferença revela sobre a contribuição relativa de cada fase?

Para respondê-las, foram conduzidos experimentos em quatro cenários, variando o tamanho do estado, a configuração de compressão e o padrão de conteúdo da memória.

6.1 CONFIGURAÇÃO DO AMBIENTE EXPERIMENTAL

Os experimentos foram conduzidos com o k3s formado por um conjunto de dois nós Raspberry Pi 4 Model B, interligados diretamente via interface Gigabit Ethernet. Há uma única interface cabeada por nó, por onde passa todo o tráfego do *cluster* e da migração. O acesso administrativo e a coleta dos resultados são feitos por Wi-Fi, em rede separada, e por isso não interferem nas medições de transferência. Cada nó dispõe de processador ARM Cortex-A72 quad-core a 1,8 GHz (arquitetura AArch64) e 8 GB de RAM. O armazenamento em ambos os nós é um cartão microSD WD Purple SD101 de 32 GB, escolhido após a identificação de um problema de saturação de escrita em cartões mais antigos, descrito na Seção 6.8.

O sistema operacional instalado é o Fedora Linux 43 (IoT Edition), com *kernel* 6.17.12-300.fc43.aarch64. A edição IoT do Fedora foi adotada por oferecer um perfil mínimo adequado com módulos *kernel* exigidos pelo CRIU: suporte a *namespaces* de usuário, *cgroups* v2, userfaultfd e *checkpoint/restore* de processos com soquetes TCP.

O orquestrador utilizado é o k3s versão v1.32.3+k3s1, com o N_1 atuando como nó servidor e o N_2 como nó agente. O CRIU, versão 4.2, já está disponível no repositório oficial do Fedora 43 e, portanto, já se encontrava instalado por padrão em ambos os nós. A sincronização de tempo entre os nós é realizada por NTP, e o desvio de relógio medido entre eles foi inferior a 0,2 ms, valor negligível para as métricas de temporização empregadas. A Tabela 5 consolida as especificações do ambiente experimental para fins de reprodutibilidade.

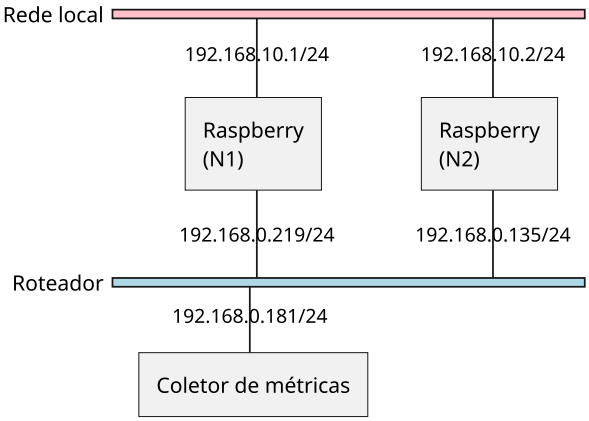
A topologia de conectividade do ambiente experimental é apresentada na Figura 12.

Tabela 5 – Especificações do ambiente experimental

Parâmetro	N_1 **(servidor)**	N_2 **(agente)**
Modelo	Raspberry Pi 4 Model B	Raspberry Pi 4 Model B
CPU	ARM Cortex-A72 quad-core, 1,8 GHz	ARM Cortex-A72 quad-core, 1,8 GHz
RAM	8 GB	8 GB
Armazenamento	MicroSD WD Purple SD101 32 GB	MicroSD WD Purple SD101 32 GB
Sistema operacional	Fedora 43 IoT Edition	Fedora 43 IoT Edition
Kernel	6.17.12-300.fc43.aarch64	6.17.12-300.fc43.aarch64
k3s	1.32.3+k3s1 (servidor)	1.32.3+k3s1 (agente)
CRIU	4.2	4.2
Responsabilidade	Nó servidor	Nó agente

Fonte: Elaborado pelo autor (2026).

Figura 12 – Topologia de conectividade do ambiente experimental



Fonte: Elaborado pelo autor (2026).

6.2 APLICAÇÃO DE BENCHMARK

Para isolar o comportamento do sistema de migração das características de uma aplicação específica, desenvolveu-se uma aplicação de benchmark dedicada. A aplicação é escrita em Go e expõe um servidor HTTP com duas interfaces, uma para coleta das métricas acumuladas pela aplicação de benchmark, e outra para zerar os contadores entre amostras consecutivas.

Ao ser inicializada, a aplicação aloca um bloco contíguo de memória de tamanho configurável por parâmetro de linha de comando (8, 16, 32, 64, 128 ou 256 MB). O bloco é preenchido segundo um de dois padrões controlados:

Padrão linear (*linear*): cada byte recebe o valor `byte(i % 256)`, resultando em uma sequência periódica altamente compressível. Esse padrão modela cargas com estado estruturado, como contadores, *buffers* de protocolo e tabelas de dados gerais.

Padrão aleatório (*random*): cada byte é preenchido com um valor pseudo-aleatório uniforme, produzindo um bloco de entropia máxima praticamente incompressível. Esse padrão representa o pior caso para compressão, modelando dados como chaves criptográficas, imagens binárias ou amostras de sensores com alta variabilidade.

Internamente, a aplicação mantém um temporizador interno com intervalo de 1 ms que detecta suspensões do processo. Quando a diferença entre observações consecutivas supera um limiar de 50 ms, o intervalo é registrado como momento do último congelamento, métrica que captura a indisponibilidade percebida pela aplicação isolando ruídos externos ao processo de migração. O temporizador não é um detector de falhas, e sim a resolução da amostragem, que define o menor intervalo observável. O limiar tampouco é um tempo limite de espera, e serve apenas para separar as suspensões atribuíveis à migração das pausas de escalonamento do sistema operacional.

Em versões preliminares, utilizou-se verificação de integridade por *hash* do bloco de memória, porém, esse mecanismo foi removido da versão final por introduzir sobrecarga e interferir na medição de desempenho. Os resultados reportados atestam, portanto, a conclusão da restauração e a duração de cada fase, e não a integridade do estado restaurado. A preservação do conteúdo de memória e dos descritores de arquivo é capacidade do CRIU, verificada em versões preliminares mas não medida nos experimentos apresentados. Conexões TCP não foram avaliadas e, na configuração adotada, não são preservadas (Seção 2.6).

6.3 INSTRUMENTAÇÃO E MÉTRICAS COLETADAS

O sistema de instrumentação coleta métricas temporais a partir de duas fontes: o shim, que registra a duração de cada fase interna da migração, e a aplicação de benchmark descrita na Seção 6.2, que mede a indisponibilidade efetivamente percebida pelo processo. As durações de cada fase são denotadas por $t_{\text{checkpoint}}$, t_{transfer} e t_{restore}, correspondendo respectivamente ao *checkpoint* CRIU, à transferência do artefato entre nós e à restauração CRIU no nó destino. O tamanho do artefato é registrado com e sem a compressão, denotados por s_{raw} e s_{comp}.

A grandeza central da análise é t_{da}, o tempo de indisponibilidade percebido pela aplicação de benchmark. Essa métrica é obtida a partir do temporizador interno da aplicação, descrito na Seção 6.2. Entre duas observações consecutivas, o intervalo supera o limiar de 50 ms sempre que o processo foi suspenso pelo mecanismo de *checkpoint*, e a maior dessas diferenças durante o ciclo de migração é registrada como t_{da}. Por ser aferida no próprio processo, t_{da} corresponde ao intervalo em que a aplicação efetivamente deixou de estar disponível, e não a partir de instantes observados externamente pelo orquestrador. É essa proximidade com a paralisação percebida pelo próprio processo que a torna a métrica apropriada para avaliar o impacto da migração sobre a execução da aplicação.

Nas migrações entre nós distintos, as duas observações que delimitam esse intervalo são registradas em máquinas diferentes, já que a aplicação é suspensa em um nó e retomada em

outro. A medida depende, portanto, da sincronização de relógio entre eles.

Além de t_{da}, o manager registra t_{di}, a janela de indisponibilidade delimitada pela própria infraestrutura de migração. Seu início é marcado pelo shim do nó de origem no instante em que a chamada `Kill` é interceptada, antes que o *checkpoint* CRIU comece, e seu término pelo shim do nó destino quando o contêiner restaurado volta a executar. Por construção, t_{di} contém integralmente as três fases medidas, o que não ocorre com t_{da}. Parte do *checkpoint* CRIU é consumida na coleta da árvore de processos e na preparação dos arquivos de imagem, período em que o processo ainda executa e, portanto, ainda registra observações no temporizador interno. Como consequência, t_{di} supera t_{da}, e a diferença entre as duas medidas corresponde à preparação da migração que a aplicação não chega a perceber.

A decomposição por fase é definida sobre t_{di}, de modo que todos os termos provenham da mesma instrumentação e a soma das parcelas reproduza exatamente a janela decomposta. Define-se então t_{k3s} como a parcela residual de t_{di} não atribuível ao *checkpoint*, à transferência ou à restauração:

$$t_{\mathrm{k3s}} = t_{di} - t_{\mathrm{checkpoint}} - t_{\mathrm{transfer}} - t_{\mathrm{restore}} \tag{1}$$

onde $t_{\mathrm{transfer}} = 0$ para migrações no mesmo nó. Nesse intervalo ocorrem etapas do orquestrador não observadas diretamente pelo shim entre o encerramento do pod de origem e o início da restauração CRIU no destino, como o agendamento, a remoção do pod encerrado e a preparação do novo pod pelo kubelet até a chamada `Create`. Como essas etapas são parcialmente concorrentes às fases medidas, t_{k3s} não corresponde ao tempo total de coordenação do orquestrador, e sim à fração dele que permanece no caminho depois de descontadas as fases instrumentadas. Por construção, tampouco permite separar individualmente as etapas que o compõem. A Figura 13 ilustra a sequência e os pontos de coleta de cada métrica ao longo do tempo de uma migração entre nós distintos.

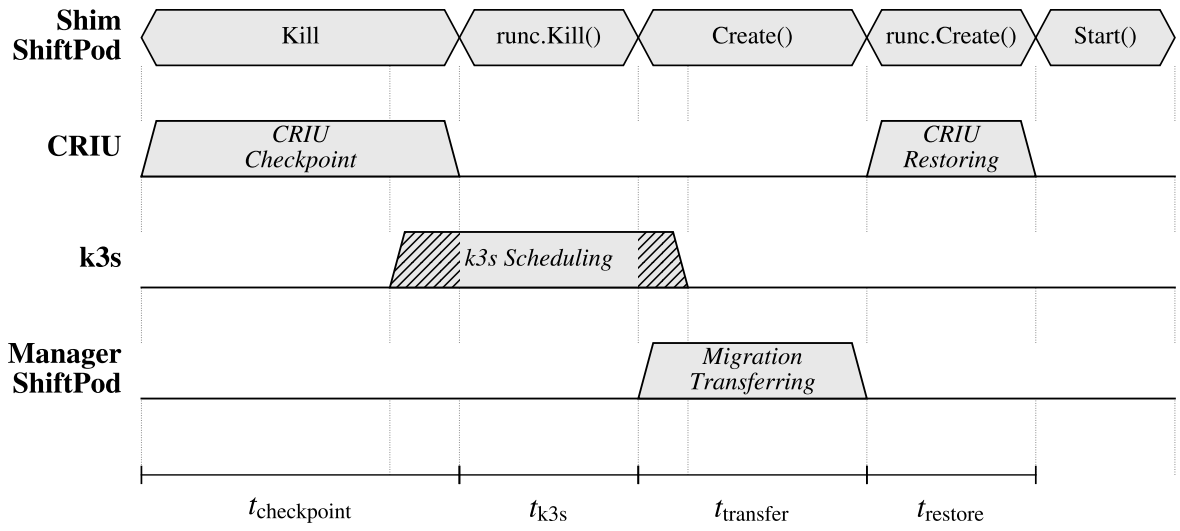
O escalonamento e a preparação do pod substituto são conduzidos pelo kubelet, o agente do k3s responsável por gerir o ciclo de vida dos pods em cada nó. Essa atividade é concorrente às fases medidas pelo shim, sobrepondo-se ao final do *checkpoint* e ao início da transferência, representação adotada na Figura 13 pela faixa hachurada que cruza os limites desses blocos.

Outra consideração importante sobre a medição de t_{da} diz respeito à forma como o Go trata valores de tempo. A função `time.Now()` retorna um valor que pode conter tanto o horário do relógio do sistema quanto uma componente monotônica, e a subtração entre dois desses valores dá preferência à componente monotônica.

Isso é relevante porque o CRIU salva o deslocamento do relógio monotônico no momento do *checkpoint* e o reaplica no processo restaurado por meio de um time *namespace* (The Linux Kernel Community, 2024b). Como consequência, do ponto de vista do processo restaurado, o relógio monotônico não avança durante a transferência pela rede, o que faria com que esse intervalo fosse descontado de t_{da} caso a medição utilizasse a componente monotônica.

Para evitar esse efeito, a aplicação de benchmark remove a componente monotônica antes

Figura 13 – Linha do tempo das métricas de temporização durante uma migração



Fonte: Elaborado pelo autor (2026).

da subtração, de modo que o cálculo passe a utilizar o relógio do sistema. Com essa correção, t_{da} inclui integralmente o intervalo em que o processo esteve suspenso, contemplando também a fase de transferência entre nós.

Além das métricas temporais, o mesmo procedimento coleta indicadores de recursos em paralelo à execução dos experimentos. A utilização de CPU e o *I/O wait* de cada nó são obtidos por leitura de `/proc/stat` via *Secure Shell* (SSH), com os percentuais calculados a partir da variação dos contadores entre leituras consecutivas, em intervalo nominal de 1 s. O consumo de CPU e de memória dos contêineres provém da API `metrics.k8s.io`, consultada a cada 2 s. As estatísticas de recursos reportadas nas Seções 6.7 e 6.8 agregam as observações contidas na janela da migração, delimitada pelo instante de congelamento detectado pela aplicação e pela duração t_{da} por ela registrada, o que resulta em 1427 a 1590 observações por cenário, das quais se reportam mediana, P_{95} e máximo.

6.4 DEFINIÇÃO DA MATRIZ DE EXPERIMENTOS

Os experimentos foram organizados em quatro cenários de migração, denotados M_{ij}, em que i indica o nó de origem e j o nó de destino: migração de N_1 para N_2 (M_{12}); migração de N_2 para N_1 (M_{21}); migração dentro de N_1, com origem e destino no mesmo nó (M_{11}); e migração dentro de N_2 (M_{22}).

Para cada cenário, foram avaliados três fatores experimentais: o tamanho da memória alocada pela aplicação, o uso de compressão zstd L1, nas condições habilitada e desabilitada, e o padrão de conteúdo da memória, classificado como linear ou aleatório. Os cenários de mesmo nó seguiram a mesma matriz de fatores dos cenários entre nós, ainda que o efeito da compressão

seja secundário na ausência da transferência de rede.

Cada combinação de fatores define um cenário experimental independente, para o qual foram coletadas 30 amostras. Esse tamanho amostral foi adotado como regra prática para obter uma estimativa estável da média, mantendo viável o custo de execução dos experimentos. Para distribuições sem assimetria ou caudas excessivamente pronunciadas, amostras dessa ordem geralmente permitem uma aproximação adequada da distribuição amostral da média. A Tabela 6 resume as variáveis da matriz experimental e a quantidade de condições avaliadas em cada uma delas, o que resulta em 96 combinações e 2880 amostras coletadas.

Tabela 6 – Resumo da matriz de experimentos

Variável	**Condições**	**Quantidade**
Direção de migração	M_{11}, M_{12}, M_{21} e M_{22}	4
Alocação de memória	8, 16, 32, 64, 128 e 256 MB	6
Padrão de memória	Linear e Aleatório	2
Compressão zstd	Habilitada e Desabilitada	2
Total de combinações	$4 \times 6 \times 2 \times 2$	96
Total de amostras	96×30	2880

Fonte: Elaborado pelo autor (2026).

Um critério de exclusão foi aplicado às medidas de t_{da}. O temporizador interno da aplicação de benchmark retém a última pausa registrada acima do limiar, de modo que, havendo mais de uma detecção em uma mesma execução, o valor retido não é atribuível à migração, e sim a alguma suspensão posterior do processo. As amostras nessa condição foram descartadas na etapa de análise, com base no número de detecções e não no valor obtido, de modo que a exclusão independe do resultado da medição. Foram excluídas 103 dessas 2880 amostras (3,58%), distribuídas de forma desigual entre os cenários (M_{11}, 7,1%; M_{12}, 4,0%; M_{21}, 3,1%; M_{22}, 0,1%), restando entre 25 e 30 amostras por cenário. O descarte afeta apenas t_{da}, uma vez que as durações de fase e t_{di} provêm do shim e do manager, não do temporizador interno, e preservam as 30 amostras originais.

Essa desigualdade entre cenários poderia enviesar a comparação entre eles. Para verificar, as médias foram recalculadas incluindo as amostras descartadas. As médias reportadas neste capítulo ficam entre 0,9% abaixo e 6,8% acima das obtidas sem o descarte, e em onze das doze comparações entre cenários de mesma origem o sentido da diferença se mantém. A única exceção afeta o ponto de cruzamento entre a migração no mesmo nó e a migração entre nós, e é retomada na Seção 6.7.2, onde esse limiar é estabelecido.

Como o desvio padrão real da população não é conhecido a priori, utiliza-se a distribuição t de Student no lugar da distribuição normal. A distribuição t tem forma semelhante à normal, porém com caudas mais largas, o que compensa a incerteza adicional introduzida por estimar o desvio padrão a partir da própria amostra. Com isso, o intervalo de confiança de 95% para a

média é calculado por:

$$\bar{x} \pm t_{n-1;0,975} \cdot \frac{s}{\sqrt{n}} \tag{2}$$

onde s é o desvio padrão amostral e $t_{n-1;0,975}$ é o quantil da distribuição t de Student com $n-1$ graus de liberdade para nível de confiança de 95%. Para as durações de fase e para t_{di}, com $n = 30$, o fator vale 2,045. Para t_{da}, cujo n varia entre 25 e 30 em razão do descarte descrito acima, o fator situa-se entre 2,045 e 2,064. Os resultados são apresentados como média com intervalo de confiança e complementados por boxplots que expõem a distribuição completa de cada cenário. A coleta foi automatizada por um script, responsável por acionar a migração por meio da deleção do pod no Kubernetes, aguardar a estabilização do sistema e registrar as métricas de cada amostra.

A validade de cada amostra foi verificada por meio da duração de congelamento registrada pela aplicação. Como o CRIU necessariamente suspende o processo durante o *checkpoint*, toda migração bem-sucedida produz um intervalo de congelamento mensurável. Caso a aplicação retorne sem registrar qualquer congelamento após um ciclo de migração, isso indica que o contêiner foi reiniciado em branco, sem restauração a partir do *checkpoint*, e a amostra é descartada como falha.

6.5 CALIBRAÇÃO DO TAMANHO DE BLOCO DE TRANSMISSÃO

O parâmetro `CHUNK_SIZE_KB` do manager define o tamanho, em kilobytes, dos blocos enviados na transmissão gRPC durante a transferência do artefato de *checkpoint* entre os nós. Antes da coleta principal, foi executado um experimento exploratório para determinar o valor que minimiza a latência de transferência.

Foram avaliados quatro valores candidatos (512, 1024, 2048 e 4096 KB) com 10 amostras por configuração no cenário sem compressão com 256 MB de alocação, que representa o pior caso para a rede e, portanto, o maior impacto do parâmetro. Os resultados estão apresentados na Tabela 7.

Tabela 7 – Resultados da calibração de `CHUNK_SIZE_KB`, sem compressão, 256 MB, 10 amostras

Tamanho do bloco (KB)	$\bar{t}_{\text{transfer}}$ (ms)	$P_{95}(t_{\text{transfer}})$ (ms)
512	3163,0	3257,7
1024	3162,7	3394,8
2048	3201,3	3342,5
4096	3231,1	3578,2

Fonte: Elaborado pelo autor (2026).

Todos os tamanhos testados apresentaram latência média dentro de uma faixa de 2,2%, indicando que a sobrecarga de protocolo das mensagens gRPC não é o fator dominante no cenário avaliado. Embora 512 KB e 1024 KB tenham apresentado médias praticamente idênticas,

optou-se por manter 512 KB por apresentar percentil de 95% inferior e, portanto, comportamento ligeiramente mais estável. A hipótese mais plausível para explicar a ausência de ganho com blocos maiores é que a largura de banda da rede já está saturada. A vazão medida nos próprios experimentos ficou entre 70 e 85 MB/s para transferências grandes, com picos de até 99 MB/s, enquanto a CPU processa dados a uma taxa muito superior, de modo que reduzir o número de mensagens não alivia o gargalo real. O leve crescimento da latência para blocos maiores (de $P_{95} = 3{,}26$ s em 512 KB até 3,58 s em 4096 KB) é consistente com a hipótese de pressão de cache, pois o BCM2711 dispõe de 1 MB de cache L2 compartilhado entre os quatro núcleos Cortex-A72 (Raspberry Pi Ltd, 2024), de modo que um bloco de 512 KB cabe integralmente nesse cache. Parcelas maiores ultrapassam essa capacidade, fazendo com que parte dos dados permaneça na memória principal durante o processamento e precise ser recarregada sucessivamente na cache, o que eleva a latência média por byte dado que o tempo de acesso à memória principal é significativamente maior que o da cache L2. Essa explicação, porém, não foi verificada experimentalmente e permanece como hipótese explicativa.

Com base nesses resultados, adotou-se `CHUNK_SIZE_KB = 512` em todos os experimentos da coleta principal. Essa configuração apresentou a menor latência de cauda ($P_{95} = 3{,}26$ s) entre todas as configurações testadas.

6.6 MIGRAÇÃO ENTRE NÓS DISTINTOS

Esta seção apresenta os resultados de indisponibilidade para os cenários M_{12} e M_{21}, que envolvem transferência de rede do arquivo de *checkpoint*. Para evitar a combinação de fatores que torne a discussão difícil de acompanhar, as figuras de t_{da} e de decomposição por fase adotam a configuração de compressão habilitada como regime de referência, por ser a recomendada para migração entre nós segundo a análise da Seção 6.6.3. O efeito da compressão é isolado nessa subseção dedicada, onde as quatro combinações de compressão (habilitada e desabilitada) e padrão de conteúdo (linear e aleatório) são comparadas diretamente. A assimetria observada entre as direções de migração é tratada na Seção 6.8.

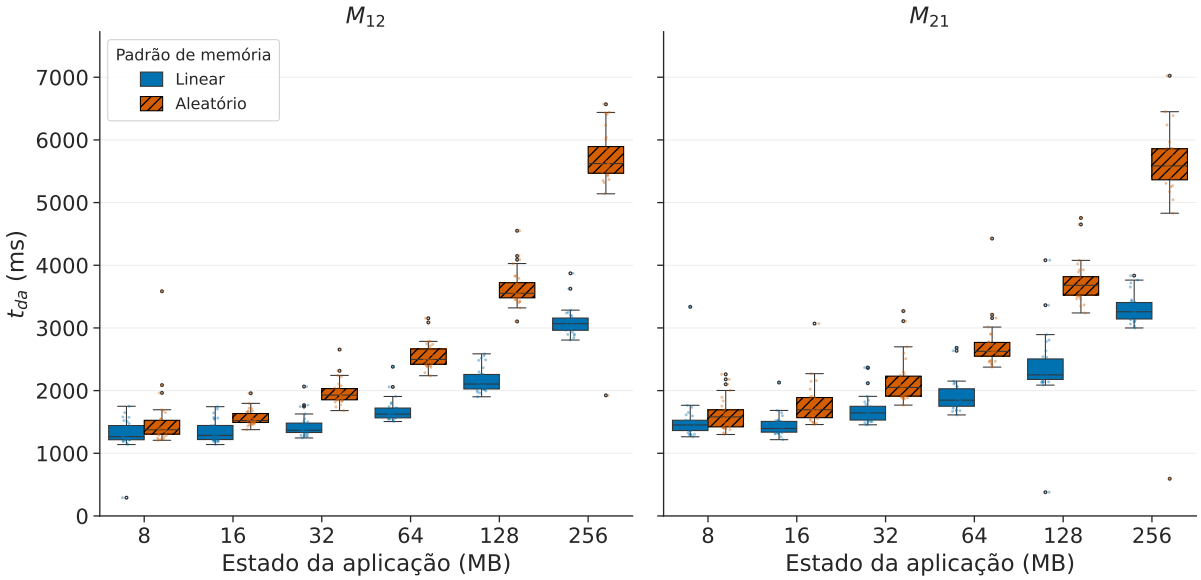
6.6.1 Indisponibilidade total observada da aplicação

A Figura 14 apresenta a distribuição de t_{da} para os dois cenários entre nós, com compressão habilitada, por tamanho de memória e por padrão de conteúdo.

Para o padrão linear, a indisponibilidade média cresce de (1307 ± 100) ms (8 MB) para (3102 ± 87) ms (256 MB) no cenário M_{12} e de (1526 ± 147) ms para (3302 ± 82) ms no cenário M_{21}. O crescimento é aproximadamente linear com o tamanho do artefato transferido, e a dispersão, expressa pela altura das caixas, mantém-se controlada ao longo da faixa avaliada.

Para o padrão aleatório, o crescimento de t_{da} é mais elevado, com médias alcançando (5609 ± 316) ms em M_{12} e (5498 ± 413) ms em M_{21} para 256 MB. O crescimento também é aproximadamente linear com o tamanho do estado, porém com inclinação cerca de duas vezes

Figura 14 – Indisponibilidade t_{da} na migração entre nós distintos. Cada caixa agrega entre 25 e 30 amostras (Seção 6.4), com compressão habilitada e padrões linear e aleatório.



Fonte: Elaborado pelo autor (2026).

maior que a do padrão linear. A diferença entre os padrões surge da etapa de transferência, pois o conteúdo aleatório praticamente não é reduzido pela compressão zstd L1. O contraste é analisado em detalhe na Seção 6.6.3. Em síntese, o crescimento de t_{da} acompanha o volume de dados efetivamente transferido, sendo aproximadamente linear nos dois padrões e com dispersão controlada, porém substancialmente mais acentuado no padrão aleatório, cuja baixa compressibilidade penaliza a etapa de transferência.

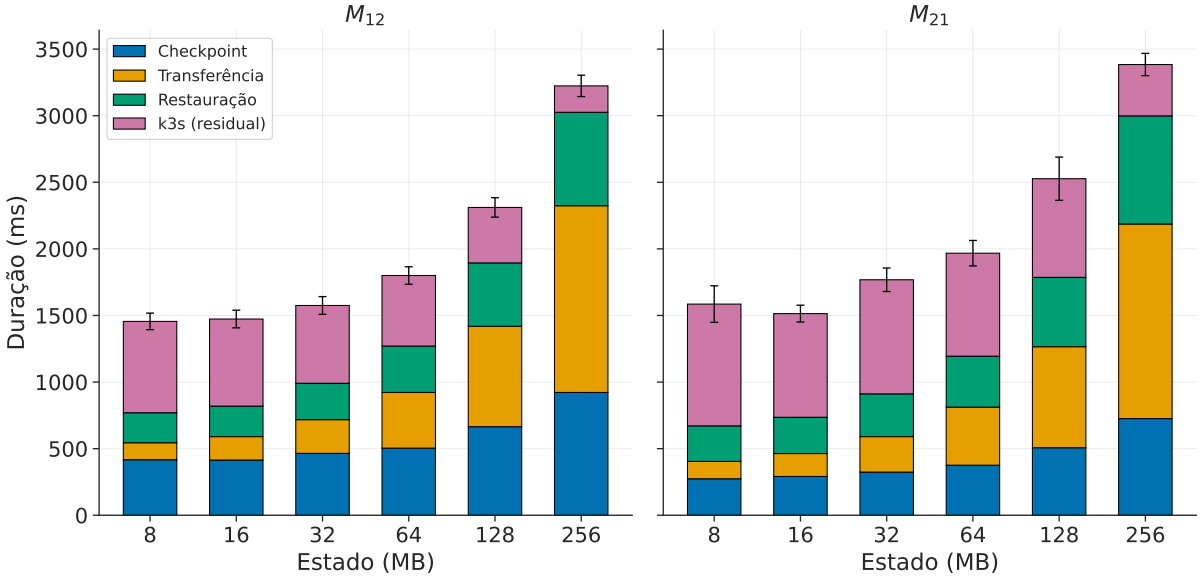
6.6.2 Decomposição por fase

A Figura 15 decompõe t_{di} nas quatro fases definidas na Seção 6.3: $t_{\text{checkpoint}}$, t_{transfer}, t_{restore} e t_{k3s}. O padrão é fixado em linear para evitar a variação adicional introduzida pelo conteúdo aleatório, cuja contribuição é analisada na subseção seguinte.

Dois regimes distintos surgem da decomposição. Para estados grandes (128 e 256 MB), a transferência de rede torna-se o componente dominante da indisponibilidade, alcançando (1402 ± 19) ms (43% do total) em M_{12} com 256 MB e (1461 ± 16) ms (43%) em M_{21}. Para estados pequenos (8 a 16 MB), a transferência representa menos de 15% do total e o t_{k3s} passa a ser a maior componente, respondendo por 44% a 58% da janela decomposta. A Tabela 8 consolida os valores médios por fase para as duas direções, no regime de compressão habilitada e padrão linear.

As fases de *checkpoint* e de restauração crescem aproximadamente com o tamanho do estado, como esperado, pois a geração do *checkpoint* e a restauração das páginas de memória pelo CRIU são proporcionais à quantidade de memória do processo. A fase de transferência cresce quase linearmente com o tamanho do artefato efetivamente transmitido, o que é compatível com

Figura 15 – Decomposição da indisponibilidade por fase nos cenários entre nós, compressão habilitada e padrão linear. Barras representam a média das 30 amostras e somam t_{di}, cujo intervalo de confiança de 95% da média é indicado pela barra de erro no topo.



Fonte: Elaborado pelo autor (2026).

Tabela 8 – Decomposição da indisponibilidade por fase nos cenários entre nós, compressão habilitada, padrão linear. As quatro fases somam t_{di} e agregam 30 amostras por linha. O t_{da} é a indisponibilidade percebida pela aplicação e agrega n amostras.

Mem.	M_{12} (ms)							M_{21} (ms)						
(MB)	t_{ckpt}	t_{tr}	t_{rst}	t_{k3s}	t_{di}	n	$t_{da} \pm \text{IC}_{95\%}$	t_{ckpt}	t_{tr}	t_{rst}	t_{k3s}	t_{di}	n	$t_{da} \pm \text{IC}_{95\%}$
8	417	128	225	686	1455	28	1307 ± 100	274	130	267	914	1585	28	1526 ± 147
16	414	176	229	654	1473	29	1353 ± 66	291	172	272	778	1514	30	1443 ± 64
32	465	253	273	584	1575	29	1433 ± 68	324	266	321	857	1768	28	1699 ± 93
64	504	418	348	530	1800	27	1682 ± 74	376	436	382	773	1968	28	1913 ± 100
128	665	754	476	416	2311	29	2173 ± 75	507	759	521	741	2527	30	2364 ± 211
256	922	1402	702	198	3224	28	3102 ± 87	726	1461	812	386	3384	29	3302 ± 82

Fonte: Elaborado pelo autor (2026).

a largura de banda disponível na rede. As duas medidas de indisponibilidade aparecem lado a lado na Tabela 8. Em todas as linhas, t_{di} supera t_{da}, conforme previsto na Seção 6.3.

A fase t_{k3s}, em contraste com as demais, decresce com o aumento do tamanho de memória. Esse comportamento é compatível com o paralelismo entre as fases medidas e o agendamento do pod substituto pelo K3s. Assim que o pod de origem entra em estado de terminação, o controlador de réplicas já cria o pod substituto, e o escalonador decide em qual nó alocá-lo, antes mesmo do shim iniciar o *checkpoint* CRIU. Esse agendamento, portanto, ocorre em paralelo às fases de *checkpoint*, transferência e restauração. Para estados pequenos, o *checkpoint*, a transferência e a restauração concluem antes do kubelet de destino finalizar a preparação do pod substituto, e t_{k3s} captura a cauda do agendamento não coberta por essas fases. Para estados grandes, as fases medidas se estendem o suficiente para absorver a maior parte do tempo de agendamento, reduzindo o intervalo remanescente.

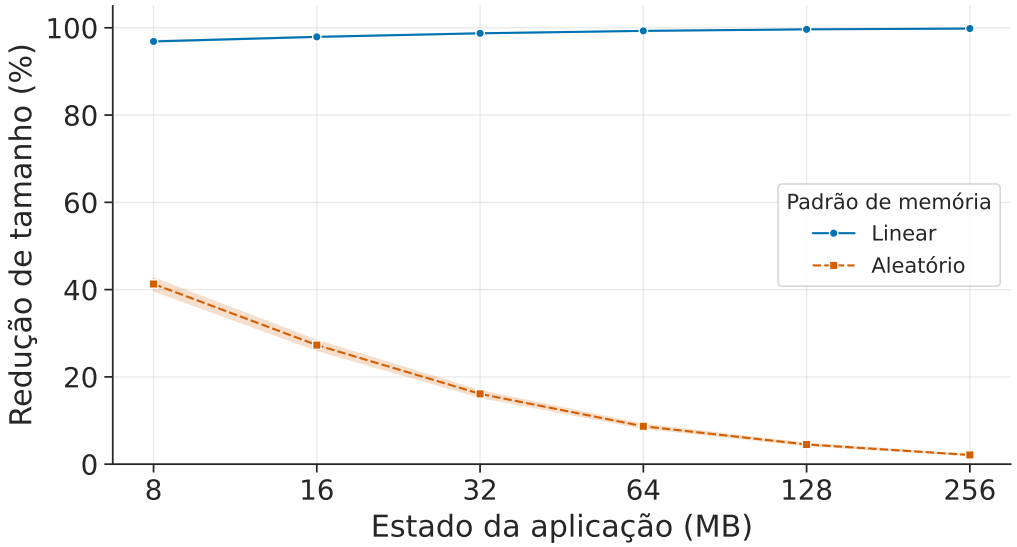
Os resultados mostram que a fase de *checkpoint* permaneceu inferior no N_2 em relação ao N_1 para todos os tamanhos avaliados, o que sugere a presença de assimetria de desempenho entre os dois nós. As causas prováveis dessa assimetria são discutidas na Seção 6.8.

Em síntese, a contribuição relativa de cada fase se inverte com o tamanho do estado, onde a parcela restante associada à coordenação do orquestrador é a maior componente para estados pequenos, enquanto a transferência de rede predomina para estados grandes.

6.6.3 Efeito da compressão e tempo de transferência

Esta subseção isola o efeito da compressão zstd L1 sobre a transferência de rede e sobre a indisponibilidade total, comparando as quatro combinações de compressão (habilitada, desabilitada) e padrão de conteúdo (linear, aleatório). A Figura 16 exibe a taxa de compressão do artefato em função do tamanho de memória.

Figura 16 – Redução do tamanho do arquivo de *checkpoint* com zstd L1 em função do tamanho de memória, por padrão de conteúdo



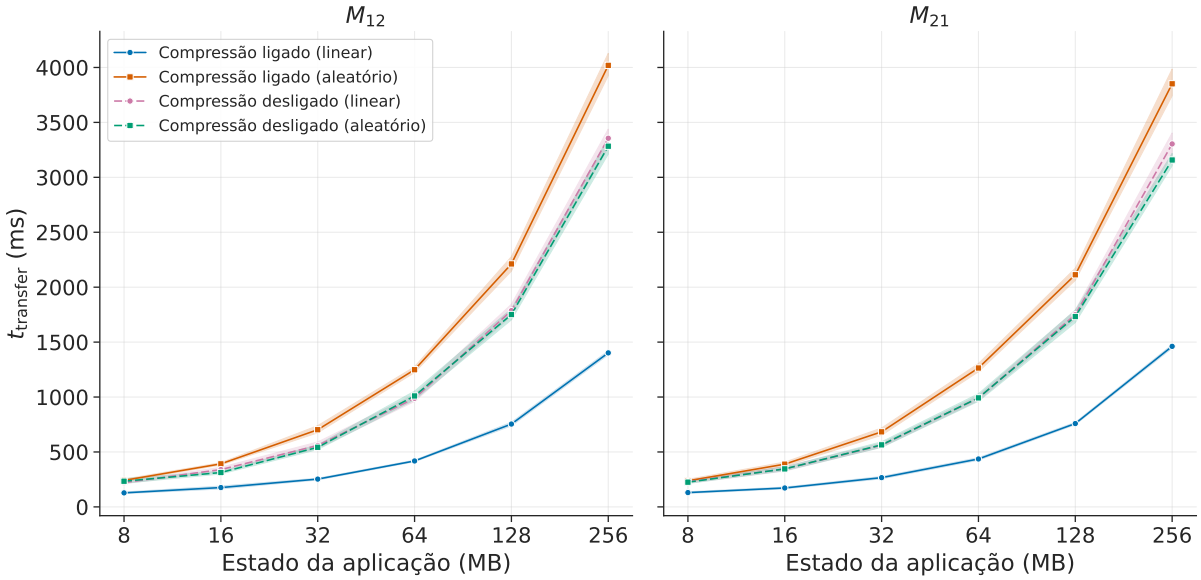
Fonte: Elaborado pelo autor (2026).

Para o padrão linear, a compressão reduz o artefato em mais de 96% já em 8 MB e a redução cresce com o tamanho, atingindo 99,8% em 256 MB ($s_{\text{comp}} \approx 0{,}5$ MB contra $s_{\text{raw}} \approx 275$ MB). Para o padrão aleatório, a redução decresce de 41% em 8 MB para 2% em 256 MB. Esse comportamento decorre da parcela do artefato que não corresponde ao bloco alocado, da ordem de 7 MB e composta por metadados e páginas do próprio *runtime*. Essa parcela é altamente compressível e responde por quase metade do artefato em 8 MB, o que explica a redução agregada de 41% mesmo com um bloco incompressível. À medida que o bloco aleatório passa a responder pela maior parte do artefato, a redução converge para a incompressibilidade desse bloco.

A Figura 17 mostra a duração da transferência em função do tamanho de memória, com um painel para cada direção de migração entre nós (M_{12} e M_{21}). Cada painel traz quatro curvas,

uma para cada combinação de compressão (habilitada e desabilitada) e padrão de conteúdo (linear e aleatório).

Figura 17 – Duração da transferência de rede em função do tamanho de memória, para os cenários entre nós. As bandas indicam intervalos de confiança de 95%.



Fonte: Elaborado pelo autor (2026).

Três observações se destacam. A primeira é que as duas curvas sem compressão, para os padrões linear e aleatório, se sobrepõem ao longo de toda a faixa de memória avaliada, uma vez que, sem compressão, o volume transmitido é aproximadamente igual ao tamanho bruto do artefato em ambos os padrões. A segunda é que, no padrão linear com compressão habilitada, t_{transfer} permanece abaixo de 1,5 s mesmo em 256 MB, consequência da redução de aproximadamente 580 vezes no tamanho do artefato transmitido. A redução na duração da transferência, porém, não é proporcional à redução no tamanho do artefato, pois a compressão introduz sobrecarga de CPU tanto na origem, durante a compressão do artefato, quanto no destino, durante sua descompressão antes da restauração CRIU. A terceira é que, no padrão aleatório, a curva com compressão habilitada permanece acima das duas curvas sem compressão, com penalidade que parte de cerca de 3% em 8 MB e se estabiliza entre 21% e 30% a partir de 32 MB na duração da transferência, indicando que o custo de CPU da tentativa de compressão não é compensado por redução útil no volume transmitido, uma vez que o artefato comprimido mantém tamanho quase idêntico ao bruto.

A Tabela 9 quantifica o efeito da compressão sobre a indisponibilidade total para o tamanho de memória de 256 MB, o pior caso para a rede.

Esse comportamento na etapa de transferência se reflete diretamente em t_{da}. Para dados lineares, a compressão reduz a indisponibilidade em cerca de 40% no cenário M_{12} e 38% em M_{21}. Cabe ressaltar que a fase de transferência engloba a codificação de todos os 275 MB brutos por tar e zstd, além da transmissão dos 0,5 MB resultantes pela rede, e não uma simples transferência

Tabela 9 – Efeito da compressão para 256 MB. Médias de 28 a 30 amostras por cenário.

Direção	**Padrão**	s_{raw} (MB)	s_{comp} (MB)	**Redução** (%)	t_{da} sem comp. (ms)	t_{da} com comp. (ms)	Δt_{da} (%)
M_{12}	Linear	275,1	0,5	99,8	5169 ± 157	3102 ± 87	-40
M_{21}	Linear	275,1	0,5	99,8	5344 ± 244	3302 ± 82	-38
M_{12}	Aleatório	274,7	268,9	2,1	4999 ± 130	5609 ± 316	$+12$
M_{21}	Aleatório	274,6	268,9	2,1	4902 ± 310	5498 ± 413	$+12$

Fonte: Elaborado pelo autor (2026).

de 0,5 MB. Para dados aleatórios, em contraste, a compressão eleva t_{da} em cerca de 12%, sem redução relevante no volume transmitido. No geral, a adequação de habilitar a compressão depende da compressibilidade do estado da aplicação. Considerando que o caso de uso alvo envolve sistemas embarcados industriais com estado predominantemente estruturado, adotou-se a compressão como configuração de referência nas demais análises deste capítulo.

6.7 MIGRAÇÃO NO MESMO NÓ

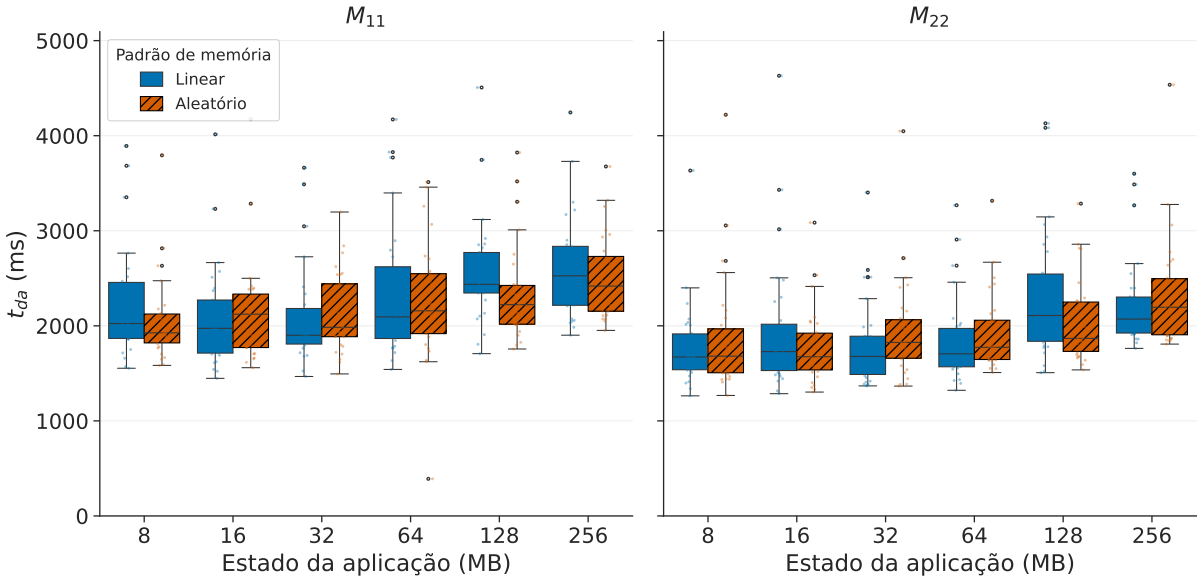
Os cenários de migração no mesmo nó (M_{11} e M_{22}) eliminam a fase de transferência pela rede, o que permite isolar, em maior medida, o t_{k3s} em comparação com os cenários entre nós. Esses cenários também têm significado operacional, pois correspondem à recriação do Pod no mesmo nó com preservação de estado, situação que ocorre em atualizações de configuração do contêiner e em operações de manutenção que exigem reinício sem perda de contexto. Para manter a consistência da apresentação ao longo do capítulo, adota-se aqui a mesma configuração de referência da Seção 6.6, com compressão habilitada, embora, nesse caso, a ausência de transferência em rede faça com que essa configuração não tenha efeito sobre o comportamento observado.

6.7.1 Indisponibilidade e decomposição por fase

Sem a etapa de transferência, t_{di} reduz-se no mesmo nó a três parcelas: $t_{\text{checkpoint}}$, t_{restore} e t_{k3s}. Esse regime isola melhor o t_{k3s}, pois elimina a única fase cuja duração depende do volume trafegado pela rede, e permite observar quanto da indisponibilidade permanece mesmo quando não há dados a transmitir. As Figuras 18 e 19 apresentam, respectivamente, a distribuição de t_{da} nos dois cenários e a decomposição de t_{di} por fase.

Três aspectos chamam atenção nessas figuras. Primeiro, na Figura 18 as caixas para os padrões linear e aleatório em cada tamanho de memória se sobrepõem em larga medida, indicando que t_{da} varia pouco com o padrão de conteúdo no regime de mesmo nó. Esse comportamento é explicado pela ausência da fase de transferência e pela estabilidade das fases de *checkpoint* e restauração entre os padrões. Segundo, t_{da} permanece na mesma ordem de grandeza ao longo de toda a faixa de memória, com médias variando entre (1771 ± 165) ms (M_{22}, 8 MB) e (2607 ± 212) ms (M_{11}, 256 MB) com padrão linear. Terceiro, a decomposição da Figura 19

Figura 18 – Indisponibilidade t_{da} na migração no mesmo nó. Cada caixa agrega entre 25 e 30 amostras (Seção 6.4), com compressão habilitada.

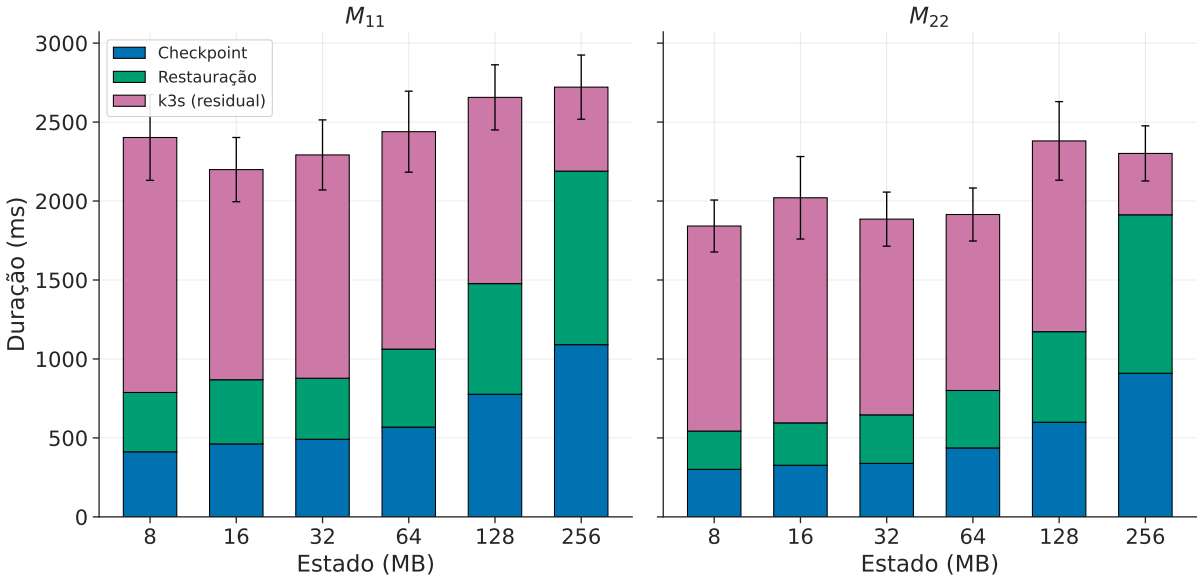


Fonte: Elaborado pelo autor (2026).

evidencia que t_{k3s} é a maior parcela isolada de t_{di} em toda a faixa de até 128 MB, com valores entre 1114 ms e 1615 ms, superiores aos observados em qualquer cenário entre nós.

A Tabela 10 consolida os valores médios por fase para os dois cenários de mesmo nó, padrão linear.

Figura 19 – Decomposição da indisponibilidade por fase nos cenários de mesmo nó, compressão habilitada e padrão linear. Barras representam a média das 30 amostras e somam t_{di}, cujo intervalo de confiança de 95% da média, calculado com distribuição t de Student, é indicado pela barra de erro no topo.



Fonte: Elaborado pelo autor (2026).

Tabela 10 – Decomposição da indisponibilidade por fase nos cenários de mesmo nó, compressão habilitada, padrão linear. As três fases somam t_{di} e agregam 30 amostras por linha. O t_{da} é a indisponibilidade percebida pela aplicação e agrega n amostras.

Mem.	M_{11} (ms)						M_{22} (ms)					
(MB)	t_{ckpt}	t_{rst}	t_{k3s}	t_{di}	n	$t_{da} \pm \text{IC}_{95\%}$	t_{ckpt}	t_{rst}	t_{k3s}	t_{di}	n	$t_{da} \pm \text{IC}_{95\%}$
8	412	376	1615	2402	26	2227 ± 247	301	242	1298	1842	30	1771 ± 165
16	462	406	1331	2199	29	2078 ± 208	327	268	1425	2021	30	1949 ± 262
32	492	386	1414	2292	27	2119 ± 214	339	307	1240	1885	30	1812 ± 171
64	569	494	1377	2439	28	2341 ± 272	437	364	1114	1915	30	1843 ± 168
128	776	701	1179	2656	29	2548 ± 211	599	573	1208	2381	30	2293 ± 249
256	1091	1099	532	2721	28	2607 ± 212	909	1003	389	2301	30	2227 ± 175

Fonte: Elaborado pelo autor (2026).

A tendência decrescente de t_{k3s} com o aumento do tamanho de memória observada no mesmo nó, pronunciada sobretudo a partir de 128 MB, é explicada pelo mesmo mecanismo discutido na Seção 6.6.2: para estados maiores, a fase de *checkpoint* excede o tempo de agendamento do kubelet, de modo que t_{k3s} passa a representar apenas o tempo remanescente entre o término das fases medidas e a conclusão da preparação do pod substituto pelo kubelet.

A Tabela 11 consolida a utilização de CPU do conjunto de nós na janela temporal da migração, definida a partir do instante de início do congelamento detectado pela aplicação de benchmark e da duração total de indisponibilidade t_{da} por ela registrada. Nesse intervalo, as medianas permanecem entre 14,7% e 17,5%, com P_{95} entre 44,6% e 49,0% e máximos próximos de 93%. Complementarmente, o *I/O wait* apresentou mediana entre 0,74% e 0,84% e P_{95} entre 2,27% e 4,89%, com picos pontuais de até 40,47%, indicando que o armazenamento não constitui gargalo sustentado nesse intervalo.

Tabela 11 – Utilização de CPU do conjunto de nós durante as janelas de migração, por cenário

Cenário	**Mediana (%)**	P_{95} **(%)**	**Máx. (%)**
M_{12}	16,4	44,8	92,7
M_{21}	15,1	49,0	93,4
M_{11}	14,7	47,8	92,7
M_{22}	17,5	44,6	92,9

Fonte: Elaborado pelo autor (2026).

As diferenças de CPU entre M_{12} e M_{21} permanecem modestas nesse intervalo, o que sugere que a assimetria observada na indisponibilidade não decorre de um desequilíbrio significativo de utilização do conjunto de nós durante a janela da migração.

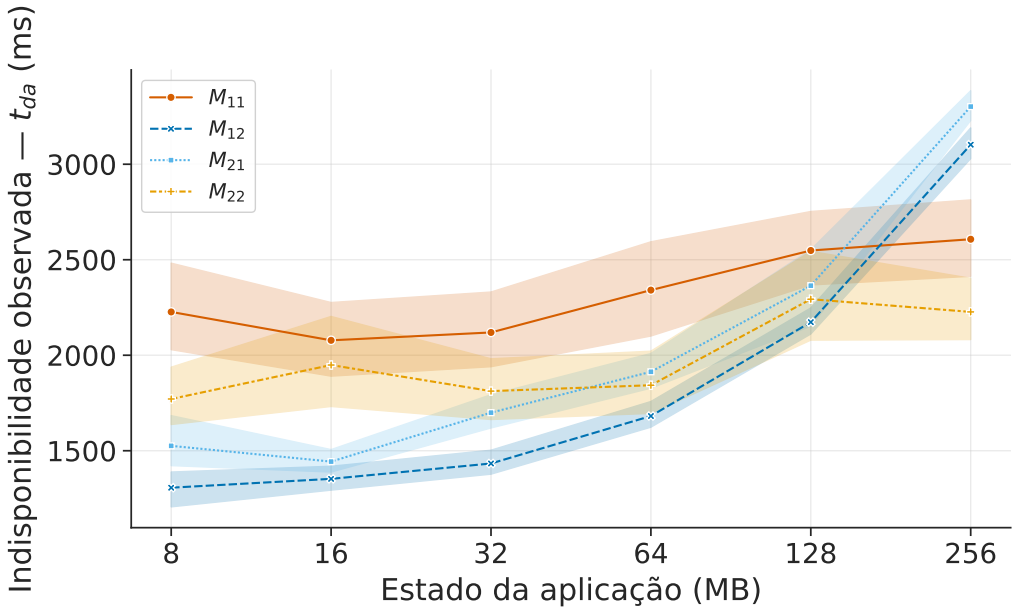
A explicação mais coerente com os dados é a serialização da fila de trabalho do kubelet. No mesmo nó, uma única instância do kubelet processa as etapas de encerramento, limpeza e inicialização em sequência, adicionando o intervalo de limpeza ao caminho crítico do agendamento. Nesse regime, portanto, a ausência de transferência de rede é compensada por um t_{k3s} maior que o observado nos cenários entre nós. A confirmação definitiva dessa hipótese requer

instrumentação interna do kubelet, fora do escopo deste trabalho.

6.7.2 Comparação com migração entre nós

A Figura 20 apresenta a comparação direta de t_{da} entre os quatro cenários de migração, com compressão habilitada e padrão linear.

Figura 20 – Indisponibilidade t_{da} por tipo de migração em função do tamanho de memória, com compressão habilitada e padrão linear



Fonte: Elaborado pelo autor (2026).

O resultado observado contraria a expectativa inicial de que a ausência de transferência de rede tornaria a migração no mesmo nó mais rápida. Para estados pequenos (8 a 32 MB), o cenário de mesmo nó apresenta t_{da} de 7% a 70% superior aos cenários entre nós, refletindo o t_{k3s} mais elevado, possivelmente associado à serialização do kubelet. Para 8 MB, a indisponibilidade média no cenário M_{11} (2227 ± 247) ms é 70% maior que em M_{12} (1307 ± 100) ms.

Já em 64 MB a inversão se manifesta no N_2, onde o cenário M_{22} passa a apresentar indisponibilidade inferior à de M_{21}. Em 256 MB, a migração no mesmo nó é mais rápida em ambos os nós de origem. No N_2, a indisponibilidade média em M_{22} (2227 ± 175) ms é 33% inferior à de M_{21} (3302 ± 82) ms. No N_1, a de M_{11} (2607 ± 212) ms é 16% inferior à de M_{12} (3102 ± 87) ms.

A comparação relevante é entre os cenários que partem do mesmo nó de origem, já que representam a escolha disponível ao escalonador. O ponto de cruzamento depende do nó de origem, situando-se entre 32 e 64 MB para um pod originado no N_2 (M_{22} contra M_{21}) e entre 128 e 256 MB para um pod originado no N_1 (M_{11} contra M_{12}), nas condições do ambiente experimental.

O limiar do N_2 é sensível ao critério de descarte definido na Seção 6.4, e é essa a exceção mencionada naquela seção. Em 64 MB, a média de t_{da} em M_{21} é de 1913 ms com o descarte e de 1798 ms sem ele, contra 1843 ms em M_{22} nos dois casos. A inversão observada em 64 MB depende, portanto, do descarte, e sem ele o cruzamento passaria para a faixa entre 64 e 128 MB. O limiar de N_1 não é afetado.

6.8 ASSIMETRIA ENTRE DIREÇÕES

Os experimentos exploratórios revelaram um problema de saturação de escrita no cartão microSD originalmente instalado no N_1, um modelo Sandisk de 8 GB, que elevava a indisponibilidade das migrações destinadas a esse nó. O cartão foi substituído por um WD Purple SD101 de 32 GB, e o *I/O wait* do nó caiu de P_{95} de 31,8% para 6,2%. Todos os dados apresentados neste capítulo foram coletados após essa substituição.

A substituição não eliminou por completo a diferença entre as direções. A Figura 21 e a Tabela 12 mostram medianas de t_{da} próximas entre si, com M_{12} abaixo de M_{21} em todos os regimes avaliados e diferenças entre 1,8% e 12,5% nas medianas agregadas por configuração.

Tabela 12 – Medianas de t_{da} por direção de migração entre nós, agregadas por configuração

Compressão	**Padrão**	M_{12} (ms)	M_{21} (ms)	Δ (%)
Ligada	Linear	1582	1779	+12,5
Ligada	Aleatório	2264	2464	+8,8
Desligada	Linear	2100	2137	+1,8
Desligada	Aleatório	2179	2326	+6,8

Fonte: Elaborado pelo autor (2026).

Essas medianas agregam todos os tamanhos de estado, o que mostra que a assimetria não depende da configuração de compressão nem do padrão de conteúdo. Observando cada tamanho, com compressão habilitada e padrão linear, a diferença absoluta entre as duas direções mantém-se praticamente constante, com média de 200 ms e desvio de 60 ms de 8 a 256 MB.

Se a assimetria viesse da vazão do disco ou da rede, ela cresceria com o tamanho do estado. Como não cresce, a causa mais provável é um custo fixo por migração. As hipóteses examinadas a seguir apontam para o plano de controle hospedado no N_1.

O consumo de CPU é reportado pelo Kubernetes em milicores, unidade em que 1000 milicores equivalem a um núcleo, e é apresentado a seguir como fração de um núcleo. No cenário de mesmo nó, o consumo do manager permanece próximo de zero, com mediana nula e P_{95} entre 0,3% e 1% de um núcleo, pois a compressão do artefato não ocorre nesse modo e o manager apenas registra o CRD de migração. Nos cenários entre nós, contudo, o manager executa compressão sobre o arquivo de *checkpoint*, e seu consumo de CPU cresce com o tamanho do estado. Em 256 MB, as medianas atingem 17,3% de um núcleo em M_{12} e 15,6% em M_{21}, com P_{95} de 34,9% e 32,6%, respectivamente. Como cada nó dispõe de quatro núcleos, mesmo o pior caso corresponde a menos de 9% da capacidade de processamento disponível. A diferença

entre as duas direções é pequena e não reproduz a assimetria observada em t_{da}, de modo que os dados de recursos são compatíveis com uma vantagem estrutural do N_2 por não hospedar o plano de controle do k3s, mas não indicam um gargalo dominante de CPU ou de I/O.

Parte da atividade do plano de controle do k3s ocorre fora da janela medida por t_{da}. O kube-apiserver, o kube-scheduler e o kube-controller-manager, todos hospedados no N_1, processam o encerramento do pod antigo antes do instante de congelamento e reconciliam o estado do conjunto de nós após a retomada do pod restaurado, intervalos que não entram na janela de indisponibilidade. A influência desses componentes sobre t_{da} é indireta e atua em dois sentidos opostos.

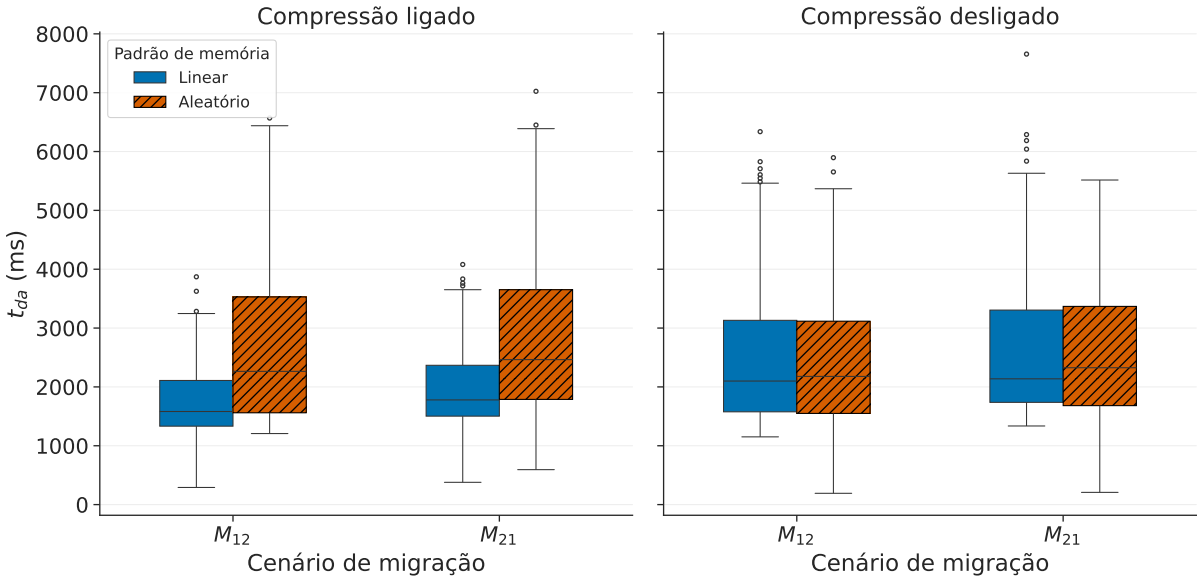
O primeiro efeito favorece justamente a direção mais lenta. Quando a restauração ocorre no N_2, caso de M_{12}, o kubelet local precisa atravessar a rede para atualizar o status do pod junto ao kube-apiserver hospedado no N_1; quando ocorre no N_1, caso de M_{21}, essa comunicação é local e sem custo de rede. Isoladamente, portanto, esse mecanismo penalizaria M_{12}, e não a direção em que a maior indisponibilidade foi de fato observada.

O segundo efeito atua no sentido contrário. Em M_{21}, o N_1 recebe e grava o artefato de *checkpoint* ao mesmo tempo em que atende as escritas do kube-apiserver no armazenamento SQLite local. As duas atividades disputam o mesmo cartão microSD e passam a esperar uma pela outra na fila de escrita do dispositivo, situação que não ocorre na direção oposta, já que o N_2 não hospeda o plano de controle.

Como a assimetria medida penaliza M_{21}, o segundo efeito prevalece sobre o primeiro no ambiente avaliado. O M_{21} é mais lento mesmo tendo um mecanismo a seu favor, o que reforça a hipótese de que o plano de controle disputa o disco do N_1 com a gravação do artefato. Ainda assim, é uma hipótese e não uma demonstração. A Figura 21 apresenta a comparação direta das distribuições de t_{da} nas duas direções, para compressão habilitada e desabilitada.

Em sistemas destinados a ambientes reais, o uso de armazenamento com taxa de escrita mais elevada, como memórias *embedded MultiMediaCard* (eMMC) ou módulos *Non-Volatile Memory Express* (NVMe) acoplados via *Peripheral Component Interconnect Express* (PCIe), reduziria tanto a variabilidade associada ao dispositivo quanto o acréscimo de 200 ms discutido acima, cuja atribuição definitiva exigiria instrumentação interna dos componentes do k3s.

Figura 21 – Distribuição de t_{da} por direção de migração entre nós, compressão habilitada e desabilitada, após a substituição do cartão SD do N_1



Fonte: Elaborado pelo autor (2026).

6.9 DISCUSSÃO DOS RESULTADOS

Os resultados permitem responder às quatro questões experimentais enunciadas no início deste capítulo.

A influência do tamanho do estado distribui-se de forma desigual entre as fases. A transferência é a que mais cresce com o artefato processado, enquanto o *checkpoint* e a restauração crescem mais lentamente, por conterem uma componente fixa relevante. O t_{k3s}, por sua vez, decresce, por ser progressivamente absorvido pelas fases medidas. Nos cenários entre nós, o sistema opera em dois regimes. Até 64 MB, a parcela residual t_{k3s} predomina na decomposição observada. A partir de 128 MB, predomina o volume de dados transferido.

O ganho obtido com a compressão zstd L1 depende da entropia do estado (Seção 6.6.3). Em conteúdo de baixa entropia o ganho é expressivo, enquanto que em conteúdo muito aleatório a compressão só acrescenta custo de CPU e deve ser desabilitada. Como a entropia do estado é uma propriedade da aplicação, e não do sistema de migração, esse parâmetro precisa ser configurável por carga de trabalho.

Os limiares de indisponibilidade produzidos pelo sistema situam-se na faixa de segundos. Os valores médios de t_{da} para o padrão linear com compressão habilitada ficam entre 1,3 s e 3,3 s nos cenários entre nós e entre 1,8 s e 2,6 s nos cenários de mesmo nó. Esses valores são compatíveis com requisitos de controle tolerantes a indisponibilidades de alguns segundos, típicos de tarefas de coordenação e supervisão em vPLCs industriais.

A comparação entre a migração no mesmo nó e a migração entre nós distintos contraria a expectativa inicial. A Seção 6.7.2 mostra que a ausência de transferência de rede não é suficiente para tornar a migração no mesmo nó mais rápida para estados pequenos, pois o t_{k3s} mais elevado,

possivelmente associado à serialização do kubelet, supera o tempo poupado com a ausência de transferência. A troca se inverte a partir de algumas dezenas de megabytes, em ponto que depende do nó de origem. Esse resultado tem implicação prática para políticas de balanceamento de carga: para estados pequenos, migrar para um nó diferente é preferível mesmo quando existe alternativa no mesmo nó.

7 CONCLUSÃO

Este trabalho investigou a migração de aplicações com estado em ambientes embarcados homogêneos, com foco em plataformas ARM64 orquestradas por k3s. O problema central reside no fato de que as soluções consolidadas de migração em contêineres foram projetadas para ambientes de nuvem, e pouco se sabe sobre seu comportamento quando aplicadas a dispositivos embarcados, onde o custo, o consumo energético e a capacidade de processamento são limitados. Para endereçar essa lacuna, foi desenvolvido o ShiftPod, um sistema de migração de contêineres integrado ao ciclo de vida nativo do Kubernetes por meio de um shim compatível com a interface do containerd, complementado por um manager distribuído, implantado em cada nó do *cluster* e responsável por coordenar a transferência de estado entre nós por meio de troca de mensagens. O sistema foi avaliado em um *cluster* de dois Raspberry Pi 4 ligados por Gigabit Ethernet, com uma aplicação de teste dedicada e instrumentação que coleta a indisponibilidade percebida pela aplicação e a contribuição relativa de cada fase.

7.1 SÍNTESE DAS CONTRIBUIÇÕES

A principal contribuição é a demonstração de que a migração com estado baseada em *checkpoint* e *restore* CRIU é viável em plataformas embarcadas ARM64, desde que a arquitetura do sistema de migração respeite as restrições do ambiente. Uma característica central do ShiftPod é a transparência para a aplicação migrada, em que a captura e a restauração do estado operam no nível do contêiner e do sistema operacional, sem exigir refatoração do código da aplicação, nem instrumentação específica, nem alterações na forma como ela é empacotada. Essa propriedade é relevante para o domínio industrial, em que as aplicações candidatas à migração incluem códigos de PLC virtualizados e outros componentes legados cuja modificação é inviável na prática.

Adicionalmente, a instrumentação desenvolvida para separar a indisponibilidade percebida pela aplicação (t_{da}) da janela medida pela infraestrutura (t_{di}), e para decompor esta última nas contribuições das fases de *checkpoint*, transferência, restauração e do resíduo t_{k3s}, fornece uma metodologia de medição que pode ser reaproveitada em avaliações futuras de sistemas análogos, oferecendo base quantitativa para decisões de projeto que antes dependiam apenas de observações qualitativas. A instrumentação, o código-fonte dos componentes e os scripts de automação dos experimentos estão disponíveis no repositório do projeto (Andrade, 2026), o que permite reproduzir as medições apresentadas.

7.2 PRINCIPAIS RESULTADOS

A avaliação experimental caracterizou o desempenho do sistema em diferentes cenários de migração, contemplando variações no tamanho do estado, na direção de migração e no uso de compressão. No regime de referência, com padrão de conteúdo linear e compressão habilitada, os tempos médios de indisponibilidade situaram-se entre 1,3 s e 3,3 s na migração

entre nós, alcançando 5,6 s no pior caso avaliado, que combina 256 MB de estado aleatório com compressão habilitada. A faixa de referência é compatível com tarefas de coordenação e supervisão em vPLCs industriais, embora exceda as janelas de latência exigidas por sistemas de controle determinístico (*soft* ou *hard real-time*). A análise detalhada dos regimes de operação, incluindo a eficácia da compressão zstd L1 sobre estados de baixa entropia e a inversão de desempenho observada entre a migração local e remota, é apresentada na Seção 6.9.

7.3 LIMITAÇÕES

O escopo de validade dos resultados é delimitado pelas escolhas metodológicas do ambiente experimental. A homogeneidade da plataforma ARM64 elimina variáveis associadas à conversão de arquitetura que seriam inevitáveis em cenários heterogêneos, os quais não foram abordados. O *cluster* composto por apenas dois nós restringe a análise de escalonabilidade, não permitindo inferir o comportamento do sistema em implantações com dezenas de nós. Por fim, a aplicação de benchmark foi projetada para maximizar a reprodutibilidade e isolar variáveis, de modo que os valores absolutos de indisponibilidade podem diferir quando observados em aplicações reais com maior diversidade de recursos em uso, como descritores de arquivo abertos, conexões TCP ativas e segmentos de memória compartilhada.

7.4 TRABALHOS FUTUROS

A partir dos resultados obtidos, foram identificados os seguintes caminhos de continuidade:

- Migração ao vivo com *lazy-pages* do CRIU (CRIU Project, 2026c): o processo é restaurado no nó de destino antes da transferência integral de sua memória, e as páginas que faltam são buscadas sob demanda no nó de origem, onde o processo original permanece congelado. Isso tende a reduzir o t_{da} médio em ordens de grandeza, mas cada falta de página passa a depender da rede, o que pode piorar o pior caso;
- Exposição das métricas de migração para algoritmos de balanceamento externos: as medições coletadas pelo próprio ShiftPod podem ser consumidas por algoritmos que determinem estratégias dinâmicas de balanceamento entre nós, substituindo as políticas estáticas atualmente empregadas;
- Análise aprofundada do escalonador do k3s: investigar os algoritmos internos de escalonamento e admissão de pods para identificar oportunidades de otimização que reduzam a parcela de t_{di} atribuída a t_{k3s}, especialmente no regime de estados pequenos, em que essa contribuição predomina;
- Extensão para ambientes heterogêneos: investigar a viabilidade de migração entre nós de arquiteturas distintas, cenário não contemplado pela validação homogênea ARM64

adotada neste trabalho e que envolve desafios adicionais associados à conversão de representações de código e de estado entre arquiteturas;

- Seleção automática da compressão: estimar a entropia do estado capturado antes da transmissão, por amostragem das páginas de memória, e habilitar ou desabilitar a codificação conforme o ganho esperado, substituindo o parâmetro estático adotado neste trabalho;
- Validação sobre outras distribuições do Kubernetes: o sistema apoia-se apenas em pontos de extensão padronizados, a especificação *Runtime v2*, a RuntimeClass e os recursos customizados, de modo que a operação sobre uma distribuição completa do Kubernetes é esperada, porém não foi verificada
- Proximidade do cliente como critério de migração: as métricas expostas pelo sistema poderiam ser combinadas com medições de latência de rede para aproximar a aplicação do cliente que a consome, caso de uso pertinente em ambientes de borda e distinto do balanceamento por carga computacional;
- Avaliação com aplicações industriais reais: substituir a aplicação de benchmark sintética por códigos de PLC virtualizados permitiria observar o impacto de recursos adicionais (descritores de arquivo, conexões TCP ativas, memória compartilhada) sobre t_{da}.

7.5 DECLARAÇÃO DE USO DE INTELIGÊNCIA ARTIFICIAL GENERATIVA

Durante a preparação desta dissertação, o autor utilizou a ferramenta de inteligência artificial generativa Gemini, principalmente como apoio à revisão gramatical e ao refinamento da redação em língua portuguesa. O autor revisou, verificou e editou todo o texto assim produzido, e assume integral responsabilidade pelo conteúdo apresentado, assegurando sua validade acadêmica.

REFERÊNCIAS

ANDRADE, Kelvin César de. **ShiftPod: migração com estado de contêineres entre nós k3s**. 2026. Repositório de código-fonte. Disponível em: https://github.com/kelvincesar/shiftpod. Acesso em: 19/09/2026. Citado 2 vezes nas páginas 47 e 82.

BURNS, Brendan *et al.* **Kubernetes: up and running: dive into the future of infrastructure**. [S.l.]: O'Reilly Media, 2022. Citado 4 vezes nas páginas 17, 23, 24 e 25.

CACCIABUE, Daniele *et al.* Stateless job offloading for mobile robots in kubernetes. In: **Proceedings of 1st International Workshop on MetaOS for the Cloud-Edge-IoT Continuum (MECC 2024)**. Torino, Italy: ACM, 2024. Citado 3 vezes nas páginas 28, 36 e 38.

CALAGNA, Antonio *et al.* Design, modeling, and implementation of robust migration of stateful edge microservices. **IEEE Transactions on Network and Service Management**, IEEE, v. 21, n. 2, p. 1877–1893, 2024. Citado 4 vezes nas páginas 28, 29, 35 e 38.

CALAGNA, Antonio *et al.* MOSE: A novel orchestration framework for stateful microservice migration at the edge. **IEEE Transactions on Network and Service Management**, IEEE, v. 22, n. 5, p. 4827–4841, 2025. Citado 3 vezes nas páginas 19, 35 e 38.

COLLET, Yann; KUCHERAWY, Murray. **Zstandard Compression and the 'application/zstd' Media Type**. [S.l.], 2021. Disponível em: https://www.rfc-editor.org/rfc/rfc8878. Acesso em: 01/07/2026. Citado na página 61.

Containerd Authors. **Runtime v2 - Shim API**. 2024. Disponível em: https://github.com/containerd/containerd/blob/main/core/runtime/v2/README.md. Acesso em: 10/03/2025. Citado 4 vezes nas páginas 27, 32, 48 e 91.

CRIU Project. **CRIU**. 2025. Disponível em: https://criu.org/Main_Page. Acesso em: 18/09/2025. Citado 2 vezes nas páginas 18 e 29.

CRIU Project. **Freezing the tree**. 2025. Documentação oficial do CRIU. Disponível em: https://criu.org/Freezing_the_tree. Acesso em: 26/09/2025. Citado na página 29.

CRIU Project. **Memory dumping and restoring**. 2025. Documentação oficial do CRIU. Disponível em: https://criu.org/Memory_dumping_and_restoring. Acesso em: 26/09/2025. Citado na página 30.

CRIU Project. **Parasite code**. 2025. Documentação oficial do CRIU. Disponível em: https://criu.org/Parasite_code. Acesso em: 26/09/2025. Citado na página 29.

CRIU Project. **TCP connection**. 2025. Documentação oficial do CRIU. Disponível em: https://criu.org/TCP_connection. Acesso em: 26/09/2025. Citado na página 30.

CRIU Project. **What cannot be checkpointed**. 2025. Documentação oficial do CRIU. Disponível em: https://criu.org/What_cannot_be_checkpointed. Acesso em: 26/09/2025. Citado na página 30.

CRIU Project. **CLI option --skip-in-flight**. 2026. Documentação oficial do CRIU. Disponível em: https://criu.org/CLI/opt/--skip-in-flight. Acesso em: 15/09/2026. Citado na página 30.

CRIU Project. **CLI option --tcp-close**. 2026. Documentação oficial do CRIU. Disponível em: https://criu.org/CLI/opt/--tcp-close. Acesso em: 15/09/2026. Citado na página 30.

CRIU Project. **Lazy migration**. 2026. Documentação oficial do CRIU. Disponível em: https://criu.org/Lazy_migration. Acesso em: 23/04/2026. Citado 2 vezes nas páginas 31 e 83.

CRIU Project. **Live migration**. 2026. Documentação oficial do CRIU. Disponível em: https://criu.org/Live_migration. Acesso em: 07/05/2026. Citado na página 30.

CRIU Project. **Page server**. 2026. Documentação oficial do CRIU. Disponível em: https://criu.org/Page_server. Acesso em: 07/05/2026. Citado na página 31.

DAS, Rohit; SIDHANTA, Subhajit. LIMOCE: Live migration of containers in the edge. In: **2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)**. [S.l.]: IEEE, 2021. p. 606–609. Citado 2 vezes nas páginas 35 e 38.

DOBIES, Jason; WOOD, Joshua. **Kubernetes operators: Automating the container orchestration platform**. [S.l.]: O'Reilly Media, 2020. Citado 3 vezes nas páginas 24, 26 e 29.

DONOVAN, Alan A. A.; KERNIGHAN, Brian W. **The Go programming language**. [S.l.]: Addison-Wesley Professional, 2015. Citado na página 31.

gRPC Contributors. **Introduction to gRPC**. 2023. Disponível em: https://grpc.io/docs/what-is-grpc/introduction/. Acesso em: 18/09/2025. Citado na página 32.

IORIO, Marco *et al.* Computing without borders: The way towards liquid computing. **IEEE Transactions on Cloud Computing**, IEEE, v. 11, n. 3, p. 2820–2838, 2023. Citado na página 37.

K3s Project Authors. **Basic Network Options**. 2026. Documentação oficial do K3s. Disponível em: https://docs.k3s.io/networking/basic-network-options. Acesso em: 15/09/2026. Citado na página 28.

K3s Project Authors. **Cluster Datastore**. 2026. Documentação oficial do K3s. Disponível em: https://docs.k3s.io/datastore. Acesso em: 15/09/2026. Citado na página 28.

KLABNIK, Steve; NICHOLS, Carol; KRYCHO, Chris. **The Rust Programming Language**. No Starch Press, 2025. Disponível em: https://doc.rust-lang.org/book/. Citado na página 32.

KOZIOLEK, Heiko; BURGER, Andreas; PEEDIKAYIL, Abdulla Puthan. Fast state transfer for updates and live migration of industrial controller runtimes in container orchestration systems. **Journal of Systems and Software**, v. 211, p. 112004, 2024. ISSN 01641212. Disponível em: https://linkinghub.elsevier.com/retrieve/pii/S0164121224000475. Acesso em: 12/03/2024. Citado 4 vezes nas páginas 18, 19, 37 e 38.

kube-rs Authors. **kube: Kubernetes client and controller runtime for Rust**. 2024. Disponível em: https://kube.rs/. Acesso em: 18/03/2025. Citado na página 56.

Kubernetes Authors. **Operator pattern**. 2025. Documentação oficial do projeto Kubernetes. Disponível em: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/. Acesso em: 15/09/2026. Citado na página 26.

Kubernetes Authors. **Pod Lifecycle**. 2025. Documentação oficial do projeto Kubernetes. Disponível em: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/. Acesso em: 02/07/2026. Citado na página 51.

Kubernetes Authors. **ReplicaSet**. 2025. Documentação oficial do projeto Kubernetes. Disponível em: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/. Acesso em: 02/07/2026. Citado na página 51.

Kubernetes Authors. **RuntimeClass**. 2025. Documentação oficial do projeto Kubernetes. Disponível em: https://kubernetes.io/docs/concepts/containers/runtime-class/. Acesso em: 02/07/2026. Citado na página 26.

Kubernetes Authors. **kubernetes/pkg/controller/controller_utils.go**. 2026. Código-fonte do projeto Kubernetes. Disponível em: https://github.com/kubernetes/kubernetes/blob/master/pkg/controller/controller_utils.go. Acesso em: 02/07/2026. Citado na página 51.

Kubernetes SIG Node. **KEP-2008: Forensic Container Checkpointing**. 2024. *Kubernetes Enhancement Proposal*. Disponível em: https://github.com/kubernetes/enhancements/tree/master/keps/sig-node/2008-forensic-container-checkpointing. Acesso em: 02/07/2026. Citado na página 36.

LASI, Heiner *et al.* Industry 4.0. **Business & Information Systems Engineering**, Springer, v. 6, n. 4, p. 239–242, 2014. Citado na página 17.

LOVE, Robert. **Linux kernel development**. [S.l.]: Pearson Education, 2010. Citado 2 vezes nas páginas 21 e 22.

MA, Lele; YI, Shanhe; LI, Qun. Efficient service handoff across edge servers via Docker container migration. In: **Proceedings of the Second ACM/IEEE Symposium on Edge Computing (SEC '17)**. [S.l.]: ACM, 2017. Citado 2 vezes nas páginas 35 e 38.

Open Container Initiative. **OCI Runtime Specification**. [S.l.], 2024. Disponível em: https://specs.opencontainers.org/runtime-spec/. Acesso em: 02/07/2026. Citado na página 27.

PEREZ, Diogenes Javier *et al.* How real (time) are virtual PLCs? In: **2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)**. [S.l.]: IEEE, 2022. p. 1–8. Citado 2 vezes nas páginas 17 e 40.

PICKARTZ, Simon *et al.* Migrating linux containers using criu. In: **ISC High Performance Workshops 2016**. [S.l.]: Springer International Publishing AG, 2016. (LNCS 9945), p. 674–684. Citado 2 vezes nas páginas 34 e 38.

POGGIANI, Leonardo *et al.* Live migration of multi-container Kubernetes pods in multi-cluster serverless edge systems. In: **Proceedings of the 1st Workshop on Serverless at the Edge (SEATED '24)**. Pisa, Italy: ACM, 2024. Citado 3 vezes nas páginas 36, 37 e 38.

QIU, Tie *et al.* Edge computing in industrial internet of things: Architecture, advances and challenges. **IEEE Communications Surveys & Tutorials**, IEEE, v. 22, n. 4, p. 2462–2488, 2020. Citado na página 17.

RANCHER. **K3s: Lightweight Kubernetes**. 2019. Disponível em: https://k3s.io/. Acesso em: 26/09/2025. Citado 2 vezes nas páginas 28 e 41.

Raspberry Pi Ltd. **BCM2711 Microprocessor Documentation**. [S.l.], 2024. Commit ba54f8d5e74e651f3bec5e6112e5a0e999af91c8. Disponível em: https://github.com/raspberrypi/documentation/blob/ba54f8d5e74e651f3bec5e6112e5a0e999af91c8/documentation/asciidoc/computers/processors/bcm2711.adoc. Acesso em: 10/04/2026. Citado na página 69.

REBER, Adrian. **Forensic container checkpointing in Kubernetes**. 2022. Blog oficial do projeto Kubernetes. Disponível em: https://kubernetes.io/blog/2022/12/05/forensic-container-checkpointing-alpha/. Acesso em: 02/07/2026. Citado na página 36.

ROSSO, Josh *et al.* **Production Kubernetes**. [S.l.]: O'Reilly Media, 2021. Citado na página 26.

Rust Foundation. **The Rust Programming Language Official Website**. 2025. Disponível em: https://www.rust-lang.org. Acesso em: 12 out. 2025. Citado na página 32.

SCHMIDT, Henri. **State-Preserving Container Orchestration in Failover Scenarios**. Dissertação (Mestrado) — Technische Universität Dortmund, 2023. Citado 2 vezes nas páginas 36 e 38.

SHANNON, Claude E. A mathematical theory of communication. **The Bell System Technical Journal**, Nokia Bell Labs, v. 27, n. 3, p. 379–423, 1948. Citado na página 61.

SILVERTHORNE, Valerie; HENDRICK, Stephen; CNCF. **CNCF Annual Survey 2024: Approaching a Decade of Code, Cloud, and Change**. [S.l.], 2025. Published March 2025. Disponível em: https://www.linuxfoundation.org/hubfs/Research%20Reports/cncf_annual_survey24_031225a.pdf?hsLang=en. Citado na página 23.

SOLLFRANK, Michael *et al.* Evaluating docker for lightweight virtualization of distributed and time-sensitive applications in industrial automation. **IEEE Transactions on Industrial Informatics**, IEEE, v. 17, n. 5, p. 3566–3576, 2021. Citado 5 vezes nas páginas 17, 23, 37, 38 e 40.

SOUSSI, Wissem; GÜR, Gürkan; STILLER, Burkhard. Democratizing container live migration for enhanced future networks: A survey. **ACM Computing Surveys**, ACM, v. 57, n. 4, p. 1–37, 2024. Citado 2 vezes nas páginas 19 e 34.

STOYANOV, Radostin *et al.* **Announcing the Checkpoint/Restore Working Group**. 2026. Kubernetes Contributors Blog. Disponível em: https://www.kubernetes.dev/blog/2026/01/21/introducing-checkpoint-restore-wg/. Acesso em: 02/07/2026. Citado na página 36.

The Linux Kernel Community. **namespaces(7) — Linux manual page**. 2024. Disponível em: https://man7.org/linux/man-pages/man7/namespaces.7.html. Acesso em: 15/09/2026. Citado na página 22.

The Linux Kernel Community. **time_namespaces(7) — Linux manual page**. 2024. Disponível em: https://man7.org/linux/man-pages/man7/time_namespaces.7.html. Acesso em: 19/04/2026. Citado 2 vezes nas páginas 23 e 65.

TOŠIĆ, Aleksandar. Run-time application migration using checkpoint/restore in userspace. **Journal of Web Engineering**, River Publishers, v. 23, n. 5, p. 735–748, 2024. Citado 3 vezes nas páginas 18, 34 e 38.

VENKATESH, Ranjan Sarpangala *et al.* Fast in-memory criu for docker containers. In: **Proceedings of the International Symposium on Memory Systems (MEMSYS '19)**. [S.l.]: ACM, 2019. Citado na página 30.

ZHANG, Hansheng *et al.* KubeSPT: Stateful pod teleportation for service resilience with live migration. **IEEE Transactions on Services Computing**, 2025. Citado 3 vezes nas páginas 19, 37 e 38.

GLOSSÁRIO

Checkpoint: cópia persistida do estado de execução de uma aplicação, utilizada para possibilitar sua restauração posterior.

Contêiner stateful: contêiner cuja execução depende da preservação de estado interno, como memória, descritores de arquivo, conexões ou dados temporários associados ao processo.

DaemonSet: objeto do Kubernetes que garante a execução de uma instância de um determinado Pod em cada nó elegível do cluster.

Namespace: mecanismo de isolamento do Linux que restringe a visibilidade de recursos do sistema operacional, como processos, rede, pontos de montagem e identificadores.

Pod: unidade básica de execução do Kubernetes, composta por um ou mais contêineres que compartilham rede, armazenamento e contexto de execução em um mesmo nó.

Restore: etapa de recriação de um processo ou contêiner a partir de um checkpoint previamente gerado, reconstituindo seu estado de execução.

Shim: processo intermediário entre o *runtime* de contêiner e a aplicação, responsável por adaptar interfaces, encaminhar eventos e gerenciar operações de ciclo de vida.

Write throttling: condição em que o subsistema de armazenamento passa a limitar ou degradar a taxa efetiva de escrita sob carga sustentada, elevando a latência das operações de I/O.

APÊNDICE A – EXEMPLOS DE RECURSOS DO SHIFTPOD

O Código A.1 apresenta um exemplo do recurso ContainerMigration em estado final, após a conclusão de uma migração entre dois nós.

Código A.1 – Exemplo de recurso ContainerMigration após conclusão da migração

```
apiVersion: shiftpod.io/v1
kind: ContainerMigration
metadata:
  name: migration-abc123-<container-id>
spec:
  pod_template_hash: "abc123"
  source_node: pi1
  source_pod: vpod-abc123-xyz
  target_node: pi2
  target_pod: vpod-abc123-qrs
  containers:
    - name: vpod
      id: <container-id>
      image_server: 10.42.0.5
      checkpoint_path: /var/lib/shiftpod/checkpoints/<pod-uid>
status:
  phase: Completed
  checkpoint_duration_ms: 895
  transfer_duration_ms: 1460
  restore_duration_ms: 706
  total_duration_ms: 3061
  total_downtime_ms: 3192
  checkpoint_size_bytes: 275627483
  compressed_size_bytes: 524670
  memory_usage_bytes: 279146496
  process_count: 6
  open_files_count: 6
  tcp_connections_count: 69
  checkpoint_completed_at: "2026-01-01T11:52:08.382Z"
  restore_completed_at: "2026-01-01T11:52:10.691Z"
```

APÊNDICE B – INTERFACE DO SHIM NA ESPECIFICAÇÃO *RUNTIME V2*

A Tabela 13 apresenta as chamadas definidas pela especificação *Runtime v2* do containerd (Containerd Authors, 2024) e indica quais são interceptadas pelo shim do ShiftPod e quais são repassadas ao runc.

Tabela 13 – Tratamento das chamadas do serviço de tarefas pelo shim do ShiftPod

Chamada	**Tratamento**	**Comportamento acrescentado**
`Create`	Interceptada	Lê a especificação OCI, identifica o tipo do contêiner, consulta o manager em busca de *checkpoint* disponível e injeta o caminho local na requisição repassada ao runc
`Kill`	Interceptada	Decide pela captura de estado a partir da anotação, do sinal e do estado do contêiner, aciona o CRIU e notifica o manager
`Start`	Repassada	Registro do instante de descongelamento e notificação `FinishRestore` ao manager
`Delete`	Repassada	Remoção do contêiner do estado em memória do shim
`Checkpoint`	Repassada	Nenhum
Demais chamadas	Repassadas	Nenhum. Abrange `State`, `Pids`, `Pause`, `Resume`, `Exec`, `ResizePty`, `CloseIO`, `Update`, `Wait`, `Stats`, `Connect` e `Shutdown`

Fonte: Elaborado pelo autor (2026).

UDESC UNIVERSIDADE DO ESTADO DE SANTA CATARINA | JOINVILLE CENTRO DE CIÊNCIAS TECNOLÓGICAS

UNIVERSIDADE DO ESTADO DE SANTA CATARINA – UDESC
BIBLIOTECA UNIVERSITÁRIA
REPOSITÓRIO INSTITUCIONAL

CENTRO DE CIÊNCIAS TECNOLÓGICAS – CCT

ATESTADO DE VERSÃO FINAL

Eu, Dr. Douglas Wildgrube Bertol, professor do curso de Programa de Pós- Graduação em Engenharia Elétrica, declaro que esta é a versão final aprovada pela comissão julgadora da dissertação/tese intitulada: **"Arquitetura para migração de aplicações com preservação de estado em ambientes embarcados"** de autoria do acadêmico Kelvin César de Andrade.

Joinville, 25 de setembro de 2026.

Assinatura digital do(a) orientador(a):

gov.br Documento assinado digitalmente
DOUGLAS WILDGRUBE BERTOL
Data: 25/09/2026 16:30:14-0300
Verifique em https://validar.iti.gov.br

_____

Douglas Wildgrube Bertol