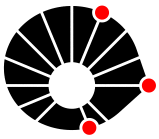

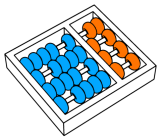

Armazenamento Persistente no Kubernetes Usando LVM com Discos Compartilhados

José Carlos Cieni Júnior *Islene Calciolari Garcia*

Relatório Técnico - IC-PFG-25-59
Projeto Final de Graduação
2025 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Armazenamento Persistente no Kubernetes Usando LVM com Discos Compartilhados

José Carlos Cieni Júnior* Islene Calciolari Garcia*

Resumo

A arquitetura extensível do Kubernetes, consolidada como uma das principais plataformas para orquestração de contêineres, viabiliza a integração de múltiplos sistemas de armazenamento persistente. Essa integração é padronizada pela especificação CSI (*Container Storage Interface*), que define um conjunto de chamadas RPC para o desenvolvimento de plugins (drivers) por terceiros. Nesse contexto, o LVM (*Logical Volume Manager*) destaca-se como uma ferramenta madura e onipresente para o gerenciamento de discos em ambientes Linux. Apesar de sua simplicidade para o gerenciamento de volumes locais, a aplicação do LVM em clusters com armazenamento compartilhado tradicionalmente depende da configuração e instalação de softwares adicionais para formação de cluster e locking distribuído, o que introduz uma complexidade elevada ao processo de implantação. Este trabalho apresenta o desenvolvimento do *csi-shared-lvm*, um driver CSI que utiliza a biblioteca de *leader election* do Kubernetes para coordenar a manipulação dos metadados do LVM e prevenir a corrupção de dados nos volumes provisionados, dispensando a necessidade de ferramentas externas. O resultado é uma solução de armazenamento robusta, de código aberto, implantação simples e que otimiza significativamente os processos de gerenciamento e instalação de armazenamento persistente em ambientes de infraestrutura compartilhada.

Palavras-chave: Kubernetes, Armazenamento Persistente, Armazenamento Compartilhado, CSI, LVM, Sistemas Distribuídos.

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

Sumário

1 Introdução

A revolução dos contêineres, impulsionada pelo surgimento do Docker, transformou fundamentalmente a maneira como softwares são desenvolvidos, empacotados e operados. O paradigma de Infraestrutura como Código (IaC) e a imutabilidade dos contêineres permitiram que aplicações fossem escaladas horizontalmente com facilidade sem precedentes. Inicialmente, esse ecossistema focou-se primariamente em aplicações *stateless* (sem estado), como servidores web e microsserviços de processamento, onde a perda de uma instância não acarretava perda de dados críticos.

Neste contexto, o Kubernetes se estabeleceu como um padrão *de facto* para a orquestração de contêineres, transformando a maneira como aplicações são implantadas em nuvens públicas, infraestruturas privadas (*on-premises*) e em ambientes de desenvolvimento e experimentação, como os populares *homelabs*. No entanto, à medida que a adoção de tecnologias *cloud-native* amadureceu, surgiu a demanda crescente de migrar cargas de trabalho *stateful* (com estado) — como bancos de dados, sistemas de mensageria e sessões em servidores web — para dentro de clusters. Diferentemente das aplicações *stateless*, essas cargas de trabalho exigem garantias estritas de persistência, durabilidade e consistência de dados.

O desafio do armazenamento em ambientes orquestrados reside na natureza dinâmica dos contêineres. Um Pod (a menor unidade de execução do Kubernetes) é efêmero, podendo ser encerrado em um nó e reiniciado em outro a qualquer momento. O armazenamento, portanto, deve ser desacoplado do ciclo de vida do contêiner e do nó físico, sendo capaz de "seguir" a aplicação onde quer que ela seja agendada.

Para endereçar essa demanda de forma padronizada e extensível, a comunidade de desenvolvimento do Kubernetes, em colaboração com outros projetos de orquestração, liderou a elaboração da especificação CSI (*Container Storage Interface*). O objetivo primário da CSI é desacoplar completamente a lógica de armazenamento do núcleo do orquestrador, definindo uma interface padrão baseada em chamadas RPC. Isso permite que qualquer provedor de armazenamento desenvolva um *plugin* (driver) compatível com o ecossistema.

Em infraestruturas Linux tradicionais, o *Logical Volume Manager* (LVM) é uma ferramenta consolidada e flexível para o gerenciamento de armazenamento em bloco. Contudo, a utilização de volumes LVM em um ambiente de armazenamento compartilhado, acessado por múltiplos nós de um cluster, apresenta desafios críticos de controle de acesso concorrente. Operações de escrita nos metadados dos volumes (como criação, exclusão ou expansão), se executadas de forma não coordenada, podem corromper informações vitais, como a localização física dos dados (*extents*), resultando em perda total de dados. As soluções existentes para mitigar esses riscos, como o *Clustered LVM* (CLVM), dependem de uma pilha de software complexa e anterior à era dos contêineres, envolvendo gerenciadores de *locking* distribuído (como dlm e sanlock) e camadas de comunicação de cluster (corosync, pacemaker). Implantar e manter essa pilha dentro ou paralelamente a um cluster Kubernetes adiciona uma sobrecarga operacional significativa e cria uma redundância funcional, visto que o próprio Kubernetes já é um gerenciador de cluster altamente sofisticado.

É relevante notar que, embora existam implementações de drivers CSI para LVM desenvolvidas pela comunidade, a maioria absoluta delas foca exclusivamente no provisionamento de volumes locais ou depende das já citadas soluções complexas como o CLVM. Essa abor-

dagem tem limitações significativas: ao criar o volume no disco de um nó específico, gera-se uma forte afinidade entre a carga de trabalho (*pod*) e aquela máquina, impedindo o seu reagendamento em outros nós do cluster. Essa restrição compromete uma das principais vantagens da orquestração: a resiliência e a distribuição flexível de aplicações. Dessa forma, permanece uma lacuna clara no que tange a soluções integradas para o uso de LVM em cenários de armazenamento compartilhado.

Este trabalho partiu da premissa de que o próprio Kubernetes já fornece os mecanismos necessários para gerenciar o acesso concorrente de forma integrada e elegante. O driver `csi-shared-lvm` utiliza a biblioteca de *leader election* do Kubernetes, que por sua vez faz uso da API de *Leases*, para garantir que apenas uma instância de seu componente de controle (*controller plugin*) esteja ativa por vez. Essa instância eleita se torna a única autoridade para executar operações de modificação no LVM, coordenando o acesso aos metadados e eliminando o risco de corrupção. A implementação resultante demonstra que é possível construir uma solução mais simples e robusta, que elimina a necessidade de softwares de cluster externos e alinha o gerenciamento do armazenamento ao ecossistema *cloud-native*. Este projeto foi desenvolvido como software de código aberto, visando facilitar sua adoção, manutenção e a contribuição da comunidade.

2 Revisão Teórica

Nesta seção, são apresentados os conceitos fundamentais que servem como base para o desenvolvimento deste projeto.

2.1 Contêineres

Contêineres são uma tecnologia de virtualização em nível de sistema operacional que permite empacotar e isolar aplicações com suas dependências completas — bibliotecas, binários e arquivos de configuração — em um ambiente de execução autocontido. Diferentemente das máquinas virtuais (VMs), que virtualizam o hardware e executam um sistema operacional completo para cada instância, os contêineres compartilham o mesmo kernel do sistema operacional do hospedeiro (*host*). Essa característica os torna extremamente leves, portáteis e eficientes, permitindo que sejam iniciados em segundos e consumam significativamente menos recursos de CPU, memória e armazenamento.

A viabilidade técnica dos contêineres no Linux baseia-se em uma combinação de recursos fundamentais do kernel que proporcionam isolamento e controle granular de recursos:

- **Namespaces (Espaços de Nomes):** Isolam os recursos globais do sistema, criando a ilusão de que cada contêiner possui sua própria instância exclusiva do sistema operacional. Os principais *namespaces* utilizados são:
  - **PID (*Process ID*):** Isola a árvore de processos, impedindo a visualização ou interação com processos externos.
  - **NET (*Network*):** Fornece uma pilha de rede própria, com interfaces, tabelas de roteamento e regras de firewall independentes.

  - **MNT (*Mount*):** Isola os pontos de montagem do sistema de arquivos, garantindo uma hierarquia de diretórios exclusiva.
  - **UTS (*UNIX Time-sharing System*):** Permite a configuração independente de *hostname* e nome de domínio.
  - **IPC (*Inter-Process Communication*):** Isola recursos de comunicação entre processos, como filas de mensagens e semáforos.
  - **USER:** Mapeia IDs de usuário e grupo, permitindo privilégios de *root* dentro do contêiner sem conceder acesso administrativo ao *host*.

- **Control Groups (*cgroups*):** Gerenciam e limitam o consumo de recursos (CPU, memória, I/O de disco e rede) por grupos de processos, garantindo que um único contêiner não possa monopolizar os recursos do sistema hospedeiro.
- **Union Filesystems (OverlayFS):** Permitem a sobreposição de sistemas de arquivos, viabilizando a construção eficiente de imagens em camadas imutáveis.

Historicamente, projetos como o LXC (*Linux Containers*) foram pioneiros ao fornecer ferramentas para interagir com esses recursos. No entanto, foi o Docker que democratizou a tecnologia ao introduzir um fluxo de trabalho simplificado e o conceito de imagens baseadas em camadas (*layered filesystem*). Isso facilitou a distribuição e o versionamento de *software*, resolvendo o clássico problema do *"na minha máquina funciona"* e garantindo que a aplicação se comporte da mesma forma independentemente do ambiente. Posteriormente, a *Open Container Initiative* (OCI) padronizou o formato das imagens e o tempo de execução (*runtime*), garantindo a interoperabilidade entre diferentes ferramentas. Essa combinação de isolamento, eficiência e padronização consolidou os contêineres como a base da arquitetura de microsserviços e das práticas modernas de DevOps.

2.2 Kubernetes

Com a popularização dos contêineres, surgiu a necessidade de orquestrar essas cargas de trabalho em larga escala. O Kubernetes (frequentemente abreviado como K8s) consolidou-se como a plataforma padrão para essa tarefa. Originalmente desenvolvido pelo Google e atualmente mantido pela *Cloud Native Computing Foundation* (CNCF), o Kubernetes abstrai a complexidade da infraestrutura subjacente, permitindo automatizar a implantação, o escalonamento e o gerenciamento de aplicações em contêineres.

O funcionamento da plataforma baseia-se em um **modelo declarativo**: o operador define o estado desejado do sistema (por exemplo, o número de réplicas de uma aplicação) através de manifestos YAML, e o Kubernetes trabalha continuamente para reconciliar o estado atual do cluster com o desejado, oferecendo recursos nativos de autorrecuperação (*self-healing*) e balanceamento de carga.

Arquiteturalmente, um cluster Kubernetes é dividido em dois planos principais:

1. **Control Plane (Plano de Controle):** Atua como o cérebro do cluster, tomando decisões globais e detectando eventos. Seus principais componentes são:

- **kube-apiserver:** O componente central que expõe a API do Kubernetes. É o único ponto de entrada para operações de gerenciamento.
- **etcd:** Um banco de dados chave-valor distribuído, consistente e de alta disponibilidade, utilizado como a fonte única de verdade para todos os dados e estados do cluster.
- **kube-scheduler:** Responsável por observar a criação de novos *Pods* e atribuí-los aos nós de trabalho mais adequados, baseando-se em disponibilidade de recursos e políticas de afinidade.
- **kube-controller-manager:** Executa os processos de controle que monitoram o estado do cluster (como detectar nós inativos ou garantir o número correto de réplicas de uma aplicação).

2. **Worker Nodes (Nós de Trabalho):** São as máquinas (físicas ou virtuais) onde as aplicações são efetivamente executadas. Cada nó contém:

   - **kubelet:** O agente principal que se comunica com o *Control Plane* e garante que os contêineres descritos nos *Pods* estejam ativos e saudáveis.
   - **kube-proxy:** Mantém as regras de rede no *host*, permitindo a comunicação entre os serviços dentro e fora do cluster.
   - **Container Runtime:** O software responsável pela execução dos contêineres (ex: *containerd*, CRI-O).

No Kubernetes, a menor unidade de implantação é o *Pod*, que pode conter um ou mais contêineres compartilhando armazenamento e rede. Contudo, devido à natureza efêmera dos *Pods* (que podem ser destruídos e recriados em nós diferentes a qualquer momento), o gerenciamento de dados persistentes exige abstrações específicas, como os *PersistentVolumes* (PVs) e *PersistentVolumeClaims* (PVCs).

2.3 gRPC

O gRPC (*gRPC Remote Procedure Calls*) é um *framework* de código aberto de alto desempenho, desenvolvido pelo Google, que permite a uma aplicação cliente executar métodos em uma aplicação servidora localizada em outra máquina com a mesma simplicidade de uma chamada de função local. Essa abstração facilita enormemente a construção de sistemas distribuídos complexos.

Diferentemente das APIs tradicionais baseadas no estilo arquitetural REST — que geralmente utilizam o protocolo HTTP/1.1 e formatos de texto verbosos como JSON — o gRPC foi projetado para baixa latência e alta eficiência, baseando-se em dois pilares fundamentais:

- **Protocol Buffers (Protobuf):** Em vez de JSON ou XML, o gRPC utiliza o Protobuf como sua Linguagem de Definição de Interface (IDL) e mecanismo de serialização. O Protobuf serializa dados estruturados em um formato binário compacto, o que reduz drasticamente o tamanho dos pacotes de rede (*payloads*) e o consumo de CPU

necessário para processá-los. Além disso, o Protobuf impõe uma abordagem *Contract-First*: a interface do serviço é definida estritamente em arquivos `.proto`, a partir dos quais o código do cliente (*stub*) e do servidor é gerado automaticamente em diversas linguagens, garantindo tipagem forte e consistência entre as partes.

- **HTTP/2:** O gRPC utiliza o HTTP/2 como camada de transporte, herdando benefícios como a multiplexação (múltiplas requisições paralelas sobre uma única conexão TCP), compressão de cabeçalhos e suporte nativo a streaming bidirecional.

2.4 Container Storage Interface (CSI)

A *Container Storage Interface* (CSI) é uma especificação padrão da indústria que visa expor sistemas de armazenamento de bloco e de arquivos (*Storage Providers* - *SPs*) a cargas de trabalho em orquestradores de contêineres (*Container Orchestrators* - COs), como o Kubernetes.

Historicamente, a lógica para interagir com volumes de armazenamento (como AWS EBS, GCP PD, servidores NFS, etc.) era integrada diretamente ao código-fonte do Kubernetes, um modelo conhecido como *in-tree*. Essa abordagem apresentava desafios significativos: a adição de novos *drivers* ou a correção de *bugs* dependia inteiramente do ciclo de lançamento do próprio Kubernetes, além de introduzir riscos de estabilidade e segurança ao núcleo do orquestrador. A CSI resolveu esse problema ao desacoplar a camada de armazenamento, permitindo que *plugins* (drivers) sejam desenvolvidos e atualizados por terceiros de forma independente (*out-of-tree*).

A arquitetura da CSI define um conjunto de serviços gRPC que um *driver* deve implementar para gerenciar o ciclo de vida completo de um volume. Esses serviços são divididos em três categorias principais:

1. **Identity Service:** Responsável por expor informações sobre o *plugin*, como nome, versão e capacidades suportadas, além de permitir verificações de saúde (*health checks*) por parte do orquestrador.

2. **Controller Service:** Gerencia as operações de alto nível e globais do cluster, que não estão atreladas a um nó específico. Suas principais funções incluem o provisionamento de volumes (`CreateVolume`, `DeleteVolume`) e anexação (`ControllerPublishVolume`, que conecta um volume a um nó). No contexto do Kubernetes, este componente geralmente é implantado como um *Deployment* ou *StatefulSet*.

3. **Node Service:** Executa operações locais no nó onde a carga de trabalho será executada. O orquestrador invoca este serviço para preparar o dispositivo (`NodeStageVolume`, que realiza formatação e montagem global) e para disponibilizá-lo ao Pod (`NodePublishVolume`, que realiza o *bind mount* no diretório do contêiner). Este componente é tipicamente implantado como um *DaemonSet*, garantindo uma instância em cada nó do cluster.

Essa separação de responsabilidades é crucial para a arquitetura proposta neste trabalho: enquanto o *Node Service* deve rodar em todos os nós para montar os volumes LVM, o

Controller Service — responsável por alterar os metadados do LVM — exige um controle rigoroso de concorrência, justificando o uso de mecanismos de eleição de líder.

2.5 Logical Volume Manager (LVM)

O *Logical Volume Manager* (LVM) é um *framework* de mapeamento de dispositivos para o kernel Linux que introduz uma camada de abstração entre o armazenamento físico e o sistema de arquivos. Diferentemente do particionamento tradicional, que é estático e rígido, o LVM permite um gerenciamento dinâmico, possibilitando o redimensionamento de volumes e a agregação de múltiplos discos em um único *pool* de armazenamento sem interrupção de serviço.

A arquitetura do LVM organiza-se em três níveis hierárquicos, fundamentados no conceito de *Extents*:

1. **Physical Volumes (PVs):** Representam a camada física bruta, que pode ser um disco rígido local, uma partição ou uma LUN (*Logical Unit Number*) oriunda de uma SAN. O LVM divide cada PV em pequenos blocos de tamanho fixo chamados **Physical Extents (PEs)**.

2. **Volume Groups (VGs):** Atuam como um repositório de armazenamento unificado, agrupando um ou mais PVs. O espaço total de um VG é a soma de todos os PEs disponíveis nos discos físicos subjacentes.

3. **Logical Volumes (LVs):** São os dispositivos de bloco virtuais consumidos pelo sistema operacional (equivalentes a partições). O LVM cria esses volumes mapeando **Logical Extents (LEs)** para os **Physical Extents (PEs)** do grupo.

É esse mapeamento entre LEs e PEs que confere ao LVM sua flexibilidade: os dados de um volume lógico não precisam estar contíguos no disco físico, podendo estar espalhados por vários discos (*spanning*) ou distribuídos para performance (*striping*). Além disso, essa abstração permite funcionalidades avançadas como *Thin Provisioning* (alocação sob demanda) e *Snapshots* (cópias de leitura/escrita em um ponto no tempo), essenciais para ambientes de contêineres.

A característica crítica para a compreensão deste trabalho reside na manipulação dos *metadados* do LVM. As informações que descrevem o mapeamento entre os extents (qual LE aponta para qual PE) são gravadas em uma área reservada no início de cada *Physical Volume*. Em uma instalação padrão, o LVM assume que o sistema operacional tem acesso exclusivo aos discos. Consequentemente, não existe um mecanismo nativo de bloqueio (*locking*) para ambientes onde múltiplos *hosts* acessam o mesmo disco compartilhado simultaneamente. Se dois nós tentarem modificar a estrutura de um VG ao mesmo tempo (por exemplo, criando volumes), ambos lerão os metadados, farão alterações na memória e tentarão gravar de volta. Isso resulta em uma condição de corrida (*race condition*) que corrompe os metadados, levando à perda irreversível do mapeamento dos dados. É este problema específico que as soluções de orquestração de armazenamento compartilhado precisam resolver.

3 Cenário e Problematização

O gerenciamento de armazenamento em ambientes distribuídos, como clusters Kubernetes, impõe desafios de coordenação que inexistem em servidores isolados (*standalone*). O cenário típico abordado neste trabalho envolve uma infraestrutura — seja *on-premises* ou em nuvem — onde é disponibilizado um único dispositivo de bloco compartilhado de grande capacidade (por exemplo, um LUN via iSCSI ou *Fibre Channel*), visível simultaneamente por todos os nós do cluster.

O problema central reside em como particionar e distribuir esse recurso monolítico entre múltiplas aplicações de forma segura, dinâmica e isolada.

3.1 Limitações dos Sistemas de Arquivos Tradicionais

Uma abordagem inicial e ingênua seria formatar esse dispositivo compartilhado com um sistema de arquivos padrão (como ext4 ou XFS) e montá-lo simultaneamente em todos os nós. Essa estratégia, contudo, é inviável. Sistemas de arquivos tradicionais não são *cluster-aware*: eles operam sob a premissa de que o sistema operacional local detém controle exclusivo sobre o dispositivo.

Se múltiplos nós montarem o mesmo volume em modo de leitura e escrita, cada *kernel* tentará gerenciar o *journal*, alocar *inodes* e atualizar blocos de forma independente, desconhecendo as operações dos vizinhos. Isso resulta em um estado de inconsistência conhecido como *Split-Brain*, levando inevitavelmente à corrupção severa do sistema de arquivos e perda de dados. Embora existam sistemas de arquivos projetados para esse fim (como GFS2 ou OCFS2), sua complexidade de gerenciamento e overhead de performance muitas vezes não justificam o uso para cargas de trabalho que requerem apenas isolamento de volume, e não compartilhamento de arquivos simultâneo.

3.2 O LVM e o Desafio da Concorrência de Metadados

O LVM apresenta-se como a ferramenta ideal para resolver o problema do particionamento. Ao tratar o grande LUN compartilhado como um *Physical Volume* (PV), é possível criar volumes lógicos (LVs) dinamicamente para atender às demandas de *Persistent Volume Claims* (PVCs) do Kubernetes.

No entanto, o uso de LVM em mídia compartilhada desloca o problema de concorrência da camada de dados para a camada de gerenciamento. As operações que alteram a estrutura do *Volume Group* — como `lvcreate`, `lvremove` ou `lvextend` — modificam os metadados gravados no início do disco.

Sem um mecanismo de coordenação, ocorre uma **condição de corrida**: se o Nó A e o Nó B tentarem criar volumes simultaneamente, ambos lerão o mesmo estado inicial dos metadados, calcularão novas alocações de *extents* que podem conflitar e tentarão gravar as alterações. O último a escrever sobrescreverá as modificações do primeiro, corrompendo o mapa de alocação do LVM e potencialmente inutilizando todo o *Volume Group*.

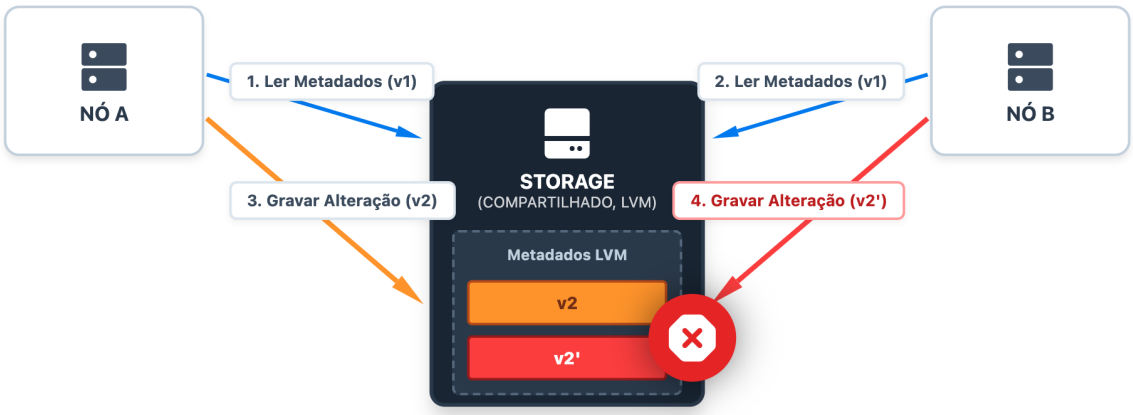

Figura 1: Ilustração da condição de corrida (Race Condition) ao manipular metadados LVM em armazenamento compartilhado sem coordenação. A escrita do Nó B sobrescreve a do Nó A, resultando em corrupção dos dados.

É crucial distinguir aqui os dois planos de operação:

1. **Plano de Controle (Crítico):** Operações que alteram metadados (criar/deletar volumes). Exigem exclusividade absoluta.
2. **Plano de Dados (Seguro):** Operações de leitura/escrita dentro do sistema de arquivos do volume lógico. Como o Kubernetes garante, através do modo de acesso `ReadWriteOnce` (RWO), que um volume só será montado em um nó por vez, não há risco de concorrência nesta camada.

3.3 A Complexidade do Clustered LVM (CLVM)

A solução tradicional da indústria para o problema dos metadados é o *Clustered LVM* (CLVM). Esta arquitetura utiliza um gerenciador de *locking* distribuído (como `dlm` ou `sanlock`) para serializar o acesso aos metadados. Para funcionar, essas ferramentas exigem uma infraestrutura de cluster subjacente, tipicamente composta por *daemons* como Corosync e Pacemaker, responsáveis por gerenciar a associação dos nós (*membership*) e o *quorum*.

No contexto do Kubernetes, adotar o CLVM implica em criar um "cluster dentro do cluster". Isso gera uma redundância conceitual e uma elevada sobrecarga operacional: o administrador precisa gerenciar e depurar uma pilha de software de alta disponibilidade complexa e legada apenas para provisionar discos.

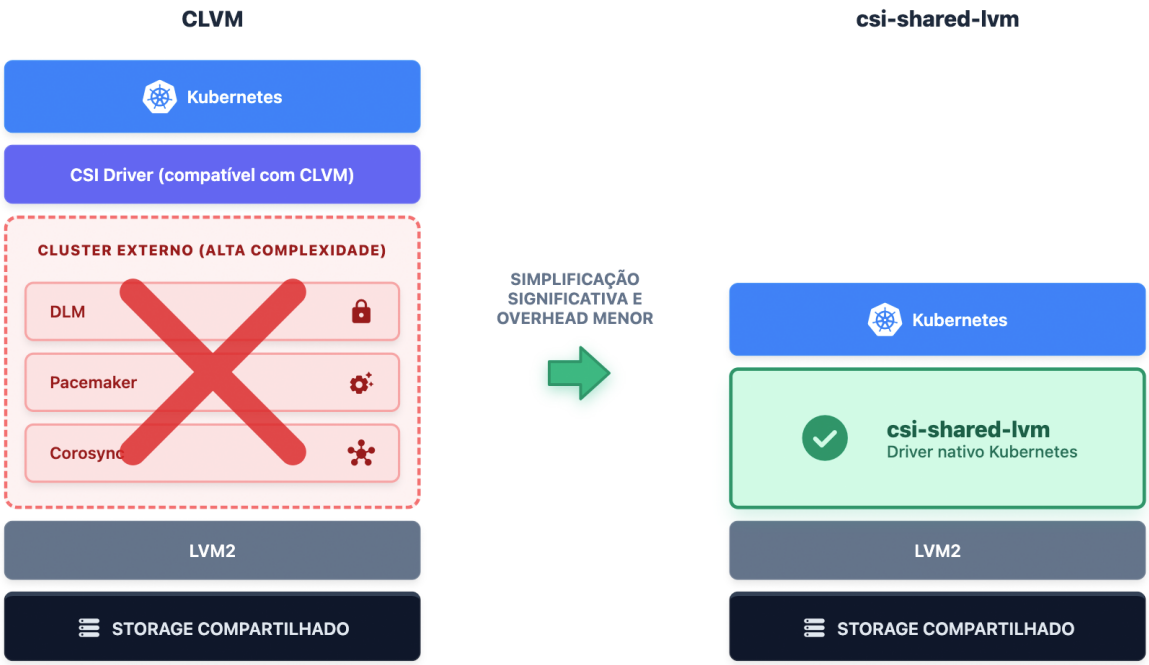

Figura 2: Comparativo da pilha de software. À esquerda, a complexidade da abordagem tradicional baseada em CLVM. À direita, a abordagem simplificada proposta neste trabalho, eliminando a dependência de gerenciadores de cluster externos.

3.4 Abordagem Proposta: Coordenação Nativa via Kubernetes

A proposta deste trabalho é eliminar a dependência de clusters externos, delegando a responsabilidade de coordenação ao próprio Kubernetes.

A arquitetura proposta utiliza o pacote `k8s.io/client-go/tools/leaderelection` para implementar um padrão de **Eleição de Líder**. Neste modelo, múltiplas réplicas do controlador de armazenamento podem existir, mas apenas a instância que detém o *lease* (o Líder) tem permissão para executar comandos que modificam os metadados do LVM. As demais instâncias permanecem em modo de espera (*standby*), respeitando esse bloqueio lógico. Essa abordagem alinha o gerenciamento do armazenamento às primitivas nativas do orquestrador, resultando em uma solução robusta, de implantação simplificada e livre de dependências de sistemas legados.

4 Arquitetura de CSI no Kubernetes

A arquitetura de integração de drivers CSI no Kubernetes é um exemplo de extensibilidade e design descentralizado. O Kubernetes não interage diretamente com o código do driver de armazenamento. Em vez disso, ele utiliza um conjunto de contêineres auxiliares, conhecidos como *sidecars*, que fazem a ponte entre os objetos da API do Kubernetes (como `PersistentVolume`, `PersistentVolumeClaim`, `StorageClass`) e as chamadas gRPC defini-

das pela especificação CSI. Essa arquitetura de sidecars permite que os drivers CSI sejam desenvolvidos e mantidos de forma independente do ciclo de lançamento do Kubernetes.

4.1 Componentes Auxiliares (sidecars)

A implementação do driver `csi-shared-lvm` utiliza os seguintes componentes oficiais mantidos pelo Kubernetes:

- **external-provisioner:** Monitora a criação de novos `PersistentVolumeClaims` (PVCs). Ao identificar uma solicitação que referencia a *StorageClass* do driver, ele invoca a chamada gRPC `CreateVolume`. Após o sucesso da operação, o sidecar é responsável por criar o objeto `PersistentVolume` (PV) correspondente na API do Kubernetes.

- **external-resizer:** Observa alterações no campo de capacidade dos PVCs existentes. Caso o usuário solicite mais espaço, este componente aciona a chamada `ControllerExpandVolume` no driver, permitindo o redimensionamento dinâmico dos volumes lógicos.

- **node-driver-registrar:** Um componente essencial que roda em cada nó do cluster. Ele registra o driver junto ao *kubelet* local, informando o caminho do *socket* UNIX onde o driver está escutando. Isso permite que o *kubelet* saiba como comunicar-se com o driver para realizar operações de montagem.

4.2 Topologia de Implantação

Para atender aos requisitos de consistência de metadados discutidos na seção anterior, o driver é dividido em dois módulos de implantação distintos:

1. **Controller Plugin (Deployment):** Responsável pelas operações globais que modificam os metadados do LVM (*Provisioning, Resizing*). Este componente é implantado como um *Deployment*. É aqui que reside a lógica de **Leader Election**: embora múltiplas réplicas possam estar em execução para alta disponibilidade, apenas o líder ativo processa as chamadas do `external-provisioner`, garantindo a serialização das escritas no *Volume Group*.

2. **Node Plugin (DaemonSet):** Responsável pelas operações locais de montagem e formatação. Este componente é implantado como um *DaemonSet*, garantindo que uma instância do driver esteja em execução em cada nó do cluster. Ele recebe comandos diretamente do *kubelet* para executar `NodeStageVolume` (ativação do LV) e `NodePublishVolume` (montagem no Pod).

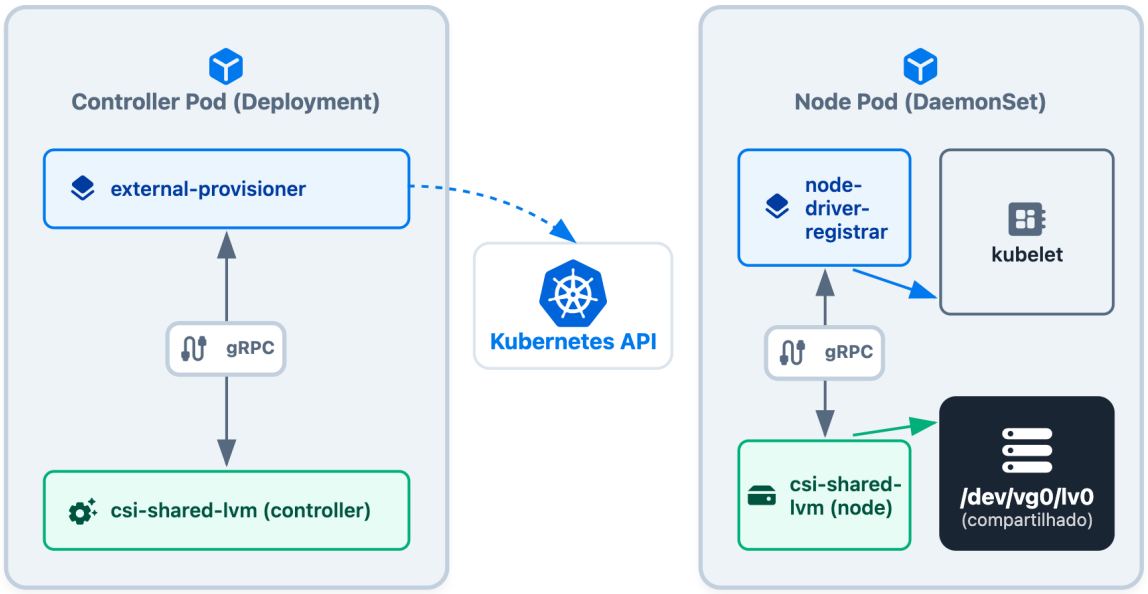

Figura 3: Arquitetura de microsserviços do driver. Os componentes auxiliares (sidecars) do Kubernetes comunicam-se com o driver desenvolvido via gRPC usando sockets UNIX.

5 Implementação do Driver

A implementação do driver materializa os conceitos teóricos discutidos anteriormente em uma solução de software desenvolvida na linguagem Go. O projeto segue a estrutura padrão de drivers CSI, segregando as responsabilidades em dois componentes distintos: o *Controller Plugin*, focado no gerenciamento e segurança dos metadados, e o *Node Plugin*, focado na disponibilização do armazenamento.

5.1 Controller Plugin

O *Controller Plugin* é o componente crítico do sistema. Implantado como um *Deployment*, ele centraliza todas as operações que modificam a estrutura do *Volume Group* (VG) compartilhado.

Diferentemente de aplicações *stateless* tradicionais, onde múltiplas réplicas processam requisições simultaneamente, o nosso controlador opera em um modelo de **Alta Disponibilidade Ativo-Passivo** (*HA-A/P*). O desafio técnico primordial é garantir que apenas um processo, em todo o cluster, tenha permissão para executar comandos de escrita no LVM (`lvcreate`, `lvremove`, `lvextend`) a qualquer momento.

Para solucionar isso de forma nativa, utilizamos o pacote `k8s.io/client-go/tools/leaderelection`. Ao iniciar, cada réplica do controlador compete para adquirir um *Lease* (uma trava lógica baseada na API de `Leases` do Kubernetes).

O ciclo de vida da aplicação é regido pelos *callbacks* deste mecanismo:

1. **OnStartedLeading:** Executado somente quando a instância adquire o *lease*. Neste

momento, o servidor gRPC é iniciado e o driver começa a aceitar chamadas `CreateVolume`.

2. **OnStoppedLeading:** Executado se a instância perder a liderança. A aplicação é encerrada imediatamente para evitar qualquer risco de *split-brain*.

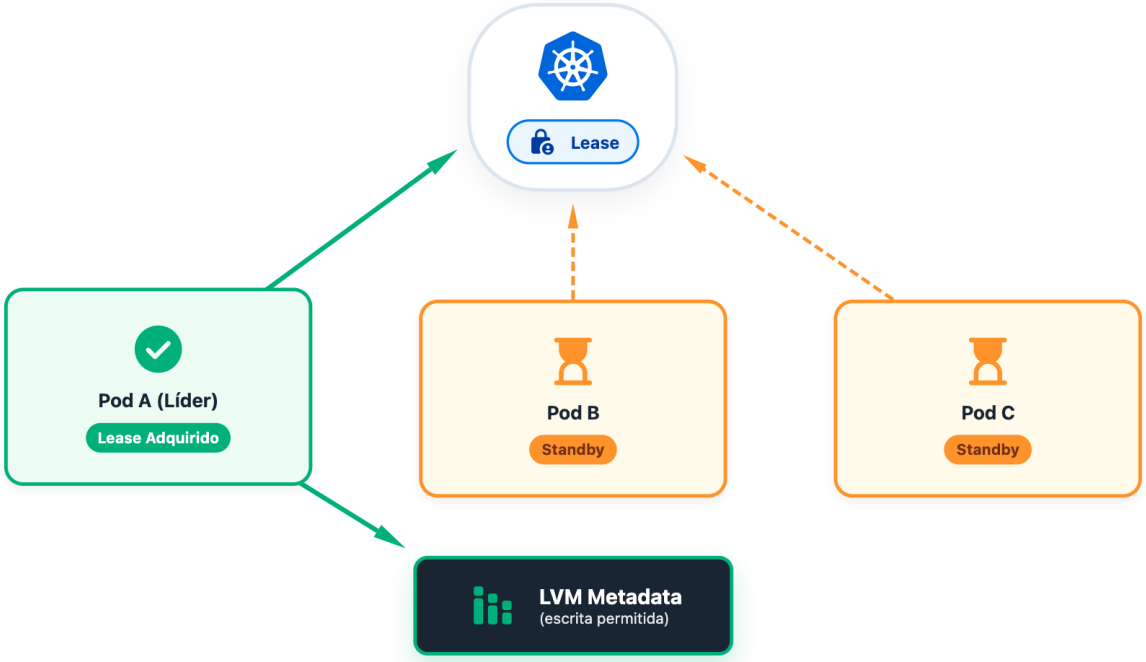

Figura 4: Mecanismo de Alta Disponibilidade com Eleição de Líder. Apenas a instância que detém o Lease na API do Kubernetes tem permissão para enviar comandos de escrita ao LVM.

Essa abordagem garante a atomicidade das transações no LVM sem a necessidade de ferramental externo.

5.2 Node Plugin

O *Node Plugin* é o agente distribuído, implantado como um *DaemonSet* em modo privilegiado para ter acesso aos dispositivos `/dev` do *host*. Suas operações são locais e seguras para execução paralela.

A implementação do serviço gRPC *Node* foca em três etapas críticas:

1. **Ativação (`NodeStageVolume`):** Em armazenamento compartilhado, a criação de um volume lógico em um nó não o torna automaticamente visível em outros. O driver executa `lvchange -ay <vg>/<lv>` para forçar o kernel a escanear e ativar o dispositivo de bloco localmente.

2. **Formatação Segura:** Utilizamos a biblioteca `k8s.io/mount-utils`, padrão do ecossistema Kubernetes. Ela oferece funções robustas como `SafeFormatAndMount`, que verifica se o dispositivo já possui um sistema de arquivos antes de tentar formatá-lo, garantindo a idempotência da operação.
3. **Montagem (`NodePublishVolume`):** Realiza o *bind mount* do volume preparado para o diretório alvo dentro do contêiner do usuário.

Para garantir a manutenibilidade e a testabilidade do código, evitamos chamadas de sistema dispersas (*exec.Command*) ao longo do projeto. Em vez disso, desenvolvemos uma camada de abstração dedicada no pacote pkg/lvm.

Esta biblioteca interna utiliza o padrão de projeto *builder/parser*:

- **Builder:** Constrói os argumentos de linha de comando a serem executados, possibilitando validação posterior.
- **Parser:** Analisa a saída dos comandos LVM para montar as estruturas de dados usadas pela aplicação.

Essa arquitetura permitiu a criação de uma suíte de testes unitários (`pkg/lvm/*_test.go`) que valida a lógica do driver sem a necessidade de um ambiente LVM real, acelerando o ciclo de desenvolvimento.

5.3 Exemplo de fluxo de execução: criar e usar um volume

A interação entre os componentes durante o provisionamento de um volume segue o seguinte fluxo lógico:

1. O usuário cria um `PersistentVolumeClaim` (PVC) e um `Pod` consumindo esse PVC.
2. O sidecar `external-provisioner` detecta o novo PVC e aciona o driver.
3. O *Controller Plugin* (Líder) recebe a chamada `CreateVolume`.
4. O driver executa `lvcreate` no armazenamento compartilhado.
5. O sidecar `external-provisioner` recebe a confirmação e cria um `PersistentVolume` (PV) correspondente.
6. O Pod é agendado em um Nó.
7. O *kubelet* invoca `NodeStageVolume` no *Node Plugin* desse nó.
8. O driver executa `lvchange -ay`, tornando o dispositivo `/dev/<vg>/<lv>` acessível.
9. O driver formata (se necessário) e monta o volume em *staging*.
10. O *kubelet* invoca `NodePublishVolume` no *Node Plugin* desse nó.
11. O driver monta o volume dentro do Pod.
12. O Pod está pronto para iniciar.

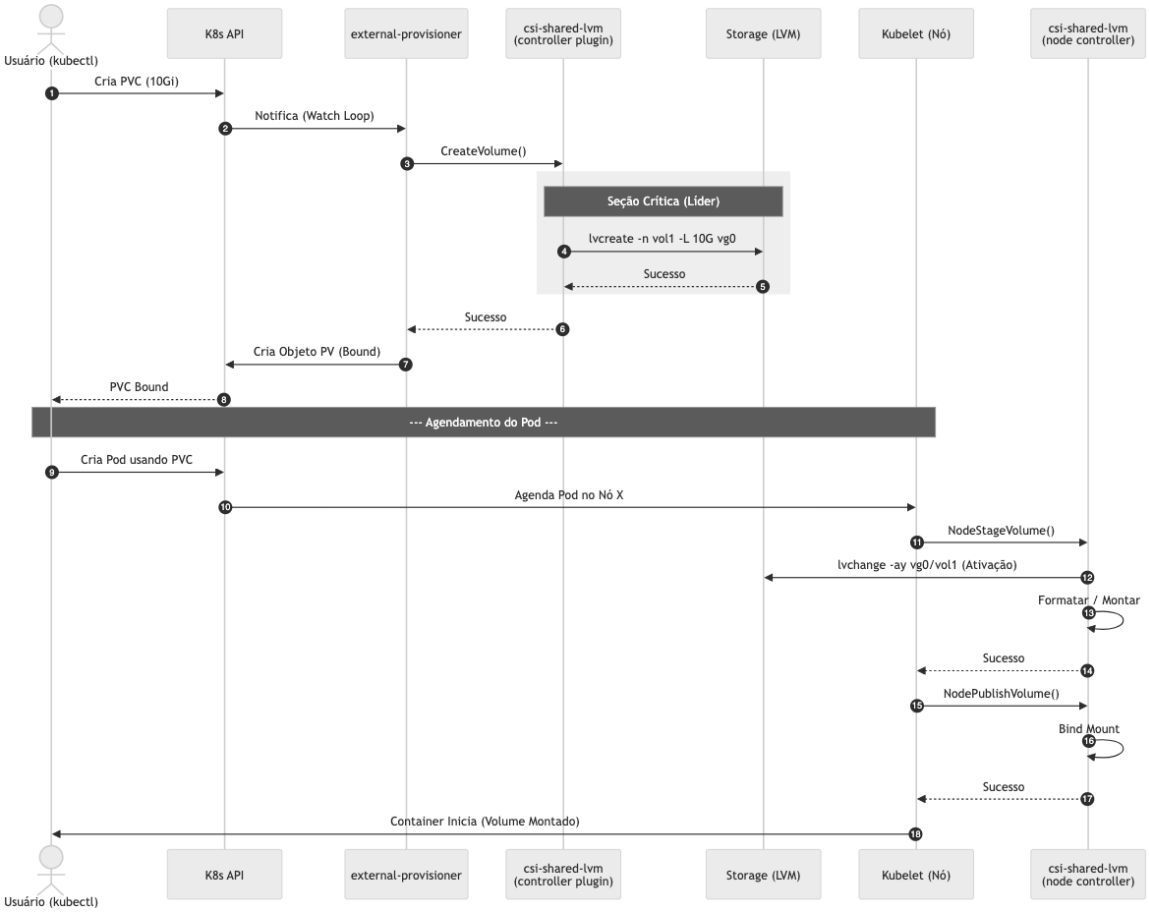

Figura 5: Diagrama de sequência detalhando o fluxo de provisionamento de um volume, desde a solicitação do PVC até a montagem no Pod.

6 Resultados e Aprendizados

O desenvolvimento do driver `csi-shared-lvm` cumpriu seu objetivo primário: validar a hipótese de que as primitivas de coordenação nativas do Kubernetes são suficientes para gerenciar o armazenamento compartilhado, eliminando a necessidade de complexas camadas de *cluster* externas.

O resultado final é um driver funcional, robusto e aderente à especificação CSI, capaz de operar em produção para cargas de trabalho que exigem persistência em mídias compartilhadas.

Um dos aprendizados mais significativos foi verificar na prática o nível de maturidade do ecossistema de desenvolvimento do Kubernetes. A complexidade de implementar o controle de concorrência foi praticamente zerada pelo uso da biblioteca `k8s.io/client-go/tools/leaderelection`, já amplamente testada e utilizada por diversos componentes *core* do próprio Kubernetes e de terceiros.

Da mesma forma, a utilização de `k8s.io/mount-utils` para as operações do *Node Plugin* garantiu que a formatação e montagem dos volumes tivessem acesso às mesmas ferramentas que o próprio orquestrador.

Outro desafio técnico superado foi a interação segura com o LVM. A decisão arquitetural de não espalhar chamadas de sistema (`exec.Command`) pelo código, centralizando-as no pacote `pkg/lvm`, provou-se acertada, promovendo a reutilização de código e facilitando na testagem dos módulos. A implementação do padrão *builder/parser* nesta camada de abstração trouxe benefícios diretos: a construção programática de comandos evita erros de sintaxe em tempo de execução, ao passo que a separação possibilitou criar uma suíte abrangente de testes unitários, tanto para formação dos comandos, quanto para validação e processamento das saídas, sem a necessidade de um ambiente LVM real para execução.

Durante os testes, observou-se que a arquitetura não introduz degradação de performance no caminho de dados (*data path*). Como o driver atua apenas no plano de controle (setup/teardown), uma vez que o volume está montado, o Pod acessa o dispositivo de bloco diretamente através do Kernel, com desempenho nativo do LVM.

No plano de controle, adotou-se conscientemente um design *Fail-Closed*. Se o controlador líder perder a conexão com a API do Kubernetes e não conseguir renovar seu *Lease*, ele encerra o processo imediatamente (*crash*), em vez de tentar operar com informações obsoletas ou redefinir o seu estado interno manualmente enquanto aguarda obter novamente a liderança. Em sistemas de armazenamento distribuído, a integridade dos dados deve sempre prevalecer sobre a disponibilidade do plano de gerenciamento.

Por fim, a condução do projeto sob a filosofia de código aberto influenciou positivamente a qualidade final da entrega. A necessidade de facilitar a adoção por terceiros impulsionou a criação de documentação detalhada, a padronização do código e o desenvolvimento de facilitadores de instalação, como os Helm Charts. O projeto não apenas resolve um problema técnico, mas serve como referência para desenvolvedores que desejam entender a criação de extensões complexas para o Kubernetes.

7 Trabalhos Futuros

Embora o `csi-shared-lvm` tenha atingido seus principais objetivos, o caminho para transformá-lo em uma solução de armazenamento de classe empresarial envolve aprimoramentos em três eixos principais: garantia de qualidade, expansão de funcionalidades e resiliência avançada.

7.1 Evolução da Estratégia de Testes

A atual suíte de testes unitários garante a correção da lógica de *parsing* e construção de comandos, mas opera sobre cenários simulados (mocks). Para elevar a confiabilidade do driver, propõe-se uma estratégia de testes em camadas:

1. **Testes de Integração LVM:** Implementação de um *pipeline* que provisione ambientes efêmeros (utilizando QEMU ou Vagrant) com dispositivos de bloco reais. Isso

permitiria que o wrapper `pkg/lvm` executasse comandos de fato no kernel, validando o comportamento do driver contra diferentes versões do LVM2 e do Linux.

2. **Conformidade CSI:** Integração com o pacote *csi-sanity*. Esta suíte executa uma bateria exaustiva de testes contra o endpoint gRPC do driver, verificando o nível de aderência à especificação CSI.

3. **Validação End-to-End (E2E):** Utilização do *framework* de testes do Kubernetes para validar o ciclo de vida completo: provisionamento, montagem, escrita de dados, redimensionamento e exclusão, garantindo a estabilidade do driver em um cluster real sob carga.

7.2 *Topology-Awareness*

Em *clusters* heterogêneos ou multi-zona, nem todos os nós possuem acesso ao mesmo armazenamento compartilhado. A implementação do suporte à topologia permitiria que o Scheduler do Kubernetes tomasse decisões inteligentes, agendando Pods apenas nos nós que efetivamente tem acesso ao *Volume Group* solicitado.

8 Agradecimentos

Agradeço primeiramente a Deus, por me conceder força, saúde e sabedoria para concluir esta etapa.

À minha família e, em especial, à minha esposa Júlia e à minha filha Maria Clara, pelo amor, pela paciência e pelo apoio incondicional durante os momentos mais desafiadores desta jornada. A compreensão e o carinho de vocês durante as horas dedicadas ao desenvolvimento deste projeto foi fundamental.

Aos meus amigos, que não me deixaram desistir nos momentos mais difíceis da minha graduação.

À Prof.ª Dr.ª Islene Calciolari Garcia, pela orientação e confiança depositada em aceitar esta proposta de projeto.

Ao Instituto de Computação (IC) e à Universidade Estadual de Campinas (UNICAMP), pela excelência na formação acadêmica e por proporcionarem o ambiente e os recursos necessários para o meu desenvolvimento acadêmico e profissional.

9 Conclusões

Este trabalho demonstrou a viabilidade técnica e prática de desenvolver um driver CSI para armazenamento compartilhado LVM utilizando exclusivamente as primitivas nativas do Kubernetes para o controle de concorrência. Com a implementação do csi-shared-lvm, o objetivo central foi plenamente alcançado, entregando uma solução robusta que suporta o ciclo de vida completo dos volumes — incluindo provisionamento dinâmico, expansão e operação em modos `Filesystem` e Block. O resultado é uma redução drástica na barreira de

entrada e na sobrecarga operacional, dispensando o uso de softwares de cluster complexos em favor de uma arquitetura limpa e integrada.

Além da eficácia funcional, o desenvolvimento do projeto evidenciou a importância de boas práticas de engenharia de software. A estruturação do código com ênfase na testabilidade, aliada ao reaproveitamento de bibliotecas comunitárias maduras, garantiu que o foco permanecesse na lógica de negócio. Essa abordagem resultou em um software estável e de fácil manutenção, pronto para evoluir com a colaboração da comunidade open source.

Conclui-se, também, que o Kubernetes não é apenas um orquestrador de contêineres, mas também uma plataforma poderosa para a construção de sistemas distribuídos complexos. Ao alavancar suas APIs e padrões de design nativos, foi possível resolver um problema clássico de infraestrutura de armazenamento de maneira mais simples, elegante e eficiente do que as abordagens tradicionais.

O código-fonte completo, a documentação e os artefatos de instalação estão disponíveis publicamente sob a licença MIT no repositório **github.com/cienijr/csi-shared-lvm**.

10 Referências Bibliográficas

AMAZON WEB SERVICES. **What is Containerization?** Disponível em: `https://aws.amazon.com/what-is/containerization/`. Acesso em: 12 dez. 2025.

BOTH, David. Logical Volume Management (LVM). *In:* USING and Administering Linux: Volume 2: Zero to SysAdmin: Advanced Topics. Berkeley, CA: Apress, 2023. p. 1–16. ISBN 978-1-4842-9615-8. DOI: `10.1007/978-1-4842-9615-8_20`. Disponível em: `https://doi.org/10.1007/978-1-4842-9615-8_20`.

CSI PROJECT MAINTAINERS. **Container Storage Interface (CSI) Specification**. Disponível em: `https://github.com/container-storage-interface/spec`. Acesso em: 12 dez. 2025.

ELLINGWOOD, Justin. **An Introduction to LVM Concepts, Terminology, and Operations**. DigitalOcean. Disponível em: `https://www.digitalocean.com/community/tutorials/an-introduction-to-lvm-concepts-terminology-and-operations`. Acesso em: 12 dez. 2025.

GOOGLE LLC. **Overview**. Disponível em: `https://protobuf.dev/overview/`. Acesso em: 12 dez. 2025.

GRPC AUTHORS. **gRPC on HTTP/2 Engineering a Robust, High-performance Protocol**. Disponível em: `https://grpc.io/blog/grpc-on-http2/`. Acesso em: 12 dez. 2025.

GRPC AUTHORS. **Introduction to gRPC**. Disponível em: `https://grpc.io/docs/what-is-grpc/introduction/`. Acesso em: 12 dez. 2025.

KOUTOUPIS, Petros. High-availability storage with HA-LVM. **Linux Journal**, Belltown Media, Houston, TX, v. 2014, n. 247, nov. 2014. ISSN 1075-3583.

MOREAU, Julien; FONTAINE, Chloé. Architecting Scalable Persistent Storage for Kubernetes with CSI And Container-Native Storage Solutions. **International Journal of Informatics and Data Science Research**, v. 1, n. 2, p. 8–21, mar. 2024. Disponível em: `http://eprints.umsida.ac.id/16180/1/8-21%2BArchitecting%2BScalable%2BPersistent%2BStorage%2Bfor%2BKubernetes.pdf`.

RED HAT. **Configuring and managing logical volumes**. Disponível em: `https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/configuring_and_managing_logical_volumes/index`. Acesso em: 12 dez. 2025.

RED HAT. **Introduction to Kubernetes Architecture**. Disponível em: `https://www.redhat.com/en/topics/containers/kubernetes-architecture`. Acesso em: 12 dez. 2025.

RED HAT. **What is Kubernetes?** Disponível em: `https://www.redhat.com/en/topics/containers/what-is-kubernetes`. Acesso em: 12 dez. 2025.

SEIDMAN, Gerry. **Understanding Kubernetes Storage: Getting in Deep by Writing a CSI Driver**. Santa Clara, CA: USENIX Association, fev. 2020. Disponível em: `https://www.usenix.org/conference/vault20/presentation/seidman`. Acesso em: 12 dez. 2025.

SHEMYAKINSKAYA, Anastasia; NIKIFOROV, Igor. Disk Space Management Automation with CSI and Kubernetes. *In:* YANG, Xin-She *et al.* (ed.). **Proceedings of Seventh International Congress on Information and Communication Technology**. Singapore: Springer Nature Singapore, 2023. p. 171–179. ISBN 978-981-19-1607-6.

THE KUBERNETES AUTHORS. **Cluster Architecture**. Disponível em: `https://kubernetes.io/docs/concepts/architecture/`. Acesso em: 12 dez. 2025.

THE KUBERNETES AUTHORS. **Container Storage Interface (CSI) for Kubernetes GA**. Disponível em: `https://kubernetes.io/blog/2019/01/15/container-storage-interface-ga/`. Acesso em: 12 dez. 2025.

THE KUBERNETES AUTHORS. **Kubernetes CSI Developer Documentation**. Disponível em: `https://kubernetes-csi.github.io/docs/`. Acesso em: 12 dez. 2025.

THE KUBERNETES AUTHORS. **Overview**. Disponível em: `https://kubernetes.io/docs/concepts/overview/`. Acesso em: 12 dez. 2025.

WIKIPEDIA CONTRIBUTORS. **Logical Volume Manager (Linux)**. Disponível em: `https://en.wikipedia.org/wiki/Logical_Volume_Manager_(Linux)`. Acesso em: 12 dez. 2025.