Redbooks
ibm.com/redbooks

IBM Storage Scale Container Native: Implementation Guide for Kubernetes and Red Hat OpenShift

Phillip Gerrard
Marcelo Capile
Abhishek Jain
Maarten Kreuger
Alok Kushvaha
Jorge Padilla
Tan Long Siau
Robert Simon
Ankita Thange

IBM

Introduction

This document provides practical guidance for implementing IBM Storage Scale Container Native in Kubernetes and Red Hat OpenShift environments. It is written for architects, administrators, and operations teams who need to design, deploy, and manage enterprise-grade shared file system storage for containerized applications and Red Hat OpenShift Virtualization virtual machines.

The focus is on implementation rather than theory. Although this guide references IBM product documentation where appropriate, it emphasizes the decisions you must make, the configurations that matter in production, and the operational patterns that help you achieve reliable, high-performance storage integration with container platforms.

Who should read this document

This document is intended for the following users:

- Platform engineers responsible for Kubernetes or Red Hat OpenShift infrastructure
- Storage administrators managing IBM Storage Scale clusters
- Site reliability engineers who need to understand storage integration and operational patterns
- Technical architects designing hybrid or containerized data platforms
- Operations teams who will manage storage lifecycle and perform recovery operations

You should have working knowledge of Kubernetes or Red Hat OpenShift, container concepts, and basic storage principles. Familiarity with IBM Storage Scale is helpful but not required.

Overview

IBM Storage Scale Container Native: Implementation Guide for Kubernetes and Red Hat OpenShift establishes the foundation for understanding IBM Storage Scale Container Native. This section includes the technical definition of IBM Storage Scale Container Native, how it differs from traditional CSI drivers, the factors driving adoption, and the key principles that govern the separation of responsibilities between Kubernetes/Red Hat OpenShift and IBM Storage Scale.

IBM Storage Scale Container Native Architecture and Platform Considerations covers the architectural model and layering of IBM Storage Scale Container Native deployments, including the separation of management and data planes. The section describes how operators bridge Kubernetes intent with Storage Scale execution, platform-specific considerations for both Red Hat OpenShift and upstream Kubernetes, and storage layer options (remote, local, and direct-attach).

Deploying IBM Storage Scale Container Native on Kubernetes provides detailed guidance on deploying IBM Storage Scale Container Native in a hybrid architecture where Kubernetes serves as the control plane while an external Storage Scale cluster delivers storage services. This section covers infrastructure preparation, platform validation, operator installation, and establishing both remote and local filesystem access.

Deploying IBM Storage Scale Container Native on Red Hat OpenShift describes the end-to-end deployment on Red Hat OpenShift Container Platform, highlighting Red Hat OpenShift-specific architectural requirements such as MachineConfig, MachineConfigPools, CRI-O, and RHCOS. This section documents platform differences while preserving a consistent deployment model.

Troubleshooting and Debugging IBM Storage Scale Container Native Deployment describes practical troubleshooting techniques for installation and configuration issues. This section provides a layered debugging approach that validates Kubernetes platform health, operator status, Storage Scale connectivity, and filesystem state while correlating errors across components.

Application Use Cases explores practical application use cases and provides guidance on selecting appropriate storage configurations for containerized workloads. This section examines persistent volume options, filesystem access types, and detailed use cases for enterprise applications including IBM FileNet® Content Manager and PostgreSQL databases.

Accelerating GenAI with IBM Storage Scale Container Native demonstrates how IBM Storage Scale Container Native accelerates generative AI workloads by providing high-performance shared storage for large language model inference systems. This section covers integration with vLLM Production Stack and LMCache to enable KV cache sharing across multiple GPU nodes.

High Availability, Data Sharing, Disaster Recovery explores high availability, data sharing, and disaster recovery strategies for IBM Storage Scale Container Native deployments. This section examines three dual-site scenarios: stretched Storage Scale clusters, local clusters using cached S3 storage, and full disaster recovery with replicated block storage.

Authors

Phillip Gerrard is a Project Leader for the International Technical Support Organization working out of Beaverton, Oregon. As part of IBM® for over 20 years, he has authored and contributed to hundreds of technical documents published to IBM.com and worked directly with IBM's largest customers to implement storage solutions and resolve critical situations. As a team lead and Subject Matter Expert for the IBM Spectrum Protect support team, he is experienced in leading and growing international teams of talented IBMers, developing and implementing team processes, creating and delivering education. Phillip holds a degree in computer science and business administration from Oregon State University.

Marcelo Capile is a hands-on technology professional with over 15 years of experience in IT and a career built within IBM. His background spans critical environments, complex operations, and the kind of challenges that require not only technical knowledge, but also composure, consistency, and sound judgment. He has developed strong expertise in infrastructure, automation, cloud, observability, and platforms such as Linux, AIX, Kubernetes, and Red Hat OpenShift. Throughout his career, he has worked with Unix administration, backup strategies, storage, networking, Ansible automation, and Python and Shell scripting, as well as IBM Cloud, AWS, and GCP. Known for his practical mindset and collaborative approach, he combines technical depth with clear communication and a genuine willingness to help others grow.

Abhishek Jain is an IBM Master Inventor and Quality Architect for IBM Storage Scale, driving quality in container-native storage for Red Hat OpenShift and Kubernetes. With more than 75 US patents and multiple publications, he advocates hybrid cloud solutions.

Maarten Kreuger is a technical pre-sales at IBM Systems Storage in The Netherlands. After almost 30 years of designing, implementing, and upgrading IBM's POWER systems and storage solutions his focus has shifted to advising, supporting, and educating IBM's customers. His main expertise is on IBM Storage Scale and Red Hat OpenShift, which he shares in the File and Object community blog.

Alok Kushvaha is a Remote Technical Support Professional for IBM Storage Scale, based in India. He is an IBM Certified Storage Scale Subject Matter Expert with over 10 years of experience in enterprise storage technologies. Alok specializes in troubleshooting complex IBM Storage Scale environments and works closely with clients, development, and field teams to resolve critical issues and improve product quality. Prior to joining IBM, Alok spent several years as a NetApp Storage Quality Assurance Engineer, where he focused on storage interoperability across various platforms, multiple host operating systems, and SAN/NVMe fabrics. His technical interests include distributed file systems, storage performance, automation, and containerized environments.

Jorge Padilla is a DevOps Developer and CI/CD Team Lead for IBM Storage Scale Container Native (CNSA), with over 7 years of experience at IBM. He began his career as a Quality Assurance Engineer, building a strong foundation in software quality and test automation before transitioning into DevOps and CI/CD engineering. Over the past 5 years, Jorge has specialized in container-native technologies, automation, and infrastructure orchestration, with a strong focus on Kubernetes and Red Hat OpenShift. As part of the CNSA CI/CD team, he designs and maintains automated Jenkins pipelines that support the deployment, testing, and validation of complex storage and

container environments. His work involves developing Groovy-based Jenkins pipelines, Ansible playbooks, and Python automation tools to deploy and manage Red Hat OpenShift and Kubernetes clusters efficiently. For the last 2 years, Jorge has served as the leader of the CNSA CI/CD team, where he drives collaboration between development, testing, and automation teams to ensure alignment with product delivery goals. He plays a key role in improving CI/CD processes, integrating new requirements, and enhancing the overall quality and reliability of IBM Storage Scale Container Native solutions.

Tan Long Siau is a Storage Software Technical Specialist at IBM ASEAN, with more than a decade of experience in the storage industry. His expertise encompasses file and object storage, as well as containerized storage solutions. A seasoned technologist, he brings extensive knowledge in big data, artificial intelligence (AI), the Internet of Things (IoT), and high-performance computing architectures.

Robert Simon is a Senior Technical Staff Member (STSM) and Senior Subject Matter Expert specializing in IBM Storage Scale and IBM Storage Scale Systems, with more than 35 years of experience in enterprise computing. He brings deep expertise in distributed and clustered file systems, Linux-based enterprise platforms, high-availability and scale-out storage architectures, and advanced performance analysis and troubleshooting. In his role, Bob serves as an escalation point for the most complex customer issues, leading root-cause analysis for critical production problems and providing clear, actionable guidance to customers, support teams, and development organizations. He is widely recognized for his systems-level understanding, calm and methodical problem-solving approach, and ability to translate complex technical behavior into practical solutions, while also mentoring engineers and contributing to improved product quality and customer success across demanding environments such as HPC, AI, analytics, and large-scale enterprise deployments.

Ankita Thange is a Remote Technical Support Professional for IBM Storage Scale, based in India. She has over 7 years of experience in IT infrastructure and enterprise storage technologies. Ankita specializes in troubleshooting complex IBM Storage Scale environments and works closely with clients, development, and field teams to resolve critical issues and improve product quality.

Feedback

This document reflects practical experience implementing IBM Storage Scale Container Native in production environments. If you encounter issues not covered here, or if you have suggestions for improving the guidance, your feedback helps make this resource more valuable for the community.

Contact IBM Redbooks

IBM Storage Scale Container Native: Implementation guide for Kubernetes and Red Hat OpenShift

Purpose

This document explores the practical adoption of IBM Storage Scale Container Native in Kubernetes® or Red Hat® OpenShift® environments. Its goal is to connect the theoretical design with real-world operations, with particular emphasis on how IBM Storage Scale Container Native behaves under change, failure, and scale.

Rather than serving as exhaustive product documentation, this IBM Redbooks® publication concentrates on the aspects of IBM Storage Scale Container Native that most directly influence deployment success and long-term operability. These include architectural boundaries, how different components interact over their lifecycle, separation of roles between the container platform and the storage layer, and typical failure scenarios observed in production environments.

The objective is not to enumerate every possible configuration option, but to provide guidance that enables informed decision-making and helps practitioners avoid common pitfalls when integrating IBM Storage Scale with Kubernetes or Red Hat OpenShift.

Technical audience and assumed knowledge

This document targets technical professionals who work with IBM Storage Scale Container Native environments, whether they are designing, deploying, or supporting these systems. The primary audience includes:

- Platform engineers responsible for Kubernetes or Red Hat OpenShift infrastructure
- Storage administrators managing IBM Storage Scale clusters
- Site reliability engineers (SREs) supporting applications that rely on stateful data
- Technical architects designing hybrid or containerized data platforms

You should already be familiar with the following key concepts:

- Kubernetes or Red Hat OpenShift components. This includes understanding pods, namespaces, and persistent storage.
- The role of etcd as the Kubernetes system of record.
- Core IBM Storage Scale concepts, including clusters, filesystems, and quorum.

Prior working experience with IBM Storage Scale Container Native is not assumed. However, hands-on experience with both the container orchestration platform and the underlying storage technology helps to apply the concepts effectively.

IBM Storage Scale Container Native overview

IBM Storage Scale Container Native allows containerized applications to access IBM Storage Scale filesystems directly. Instead of relying on protocol gateways or abstract storage layers, IBM Storage Scale Container Native provides a native client interface.

From a technical perspective, IBM Storage Scale Container Native includes the following capabilities:

- Exposes Storage Scale filesystems to container workloads as mounted POSIX filesystems
- Uses Kubernetes or Red Hat OpenShift Custom Resources to declaratively define the desired storage state
- Employs an operator-based reconciliation model to align container orchestration platform intent with Storage Scale configuration
- Preserves a single global namespace across containerized and non-containerized clients

IBM Storage Scale Container Native does not replace IBM Storage Scale management, recovery logic, or data services, nor does it add storage functionality into Kubernetes or Red Hat OpenShift. Instead, IBM Storage Scale Container Native integrates Kubernetes or Red Hat OpenShift's declarative state with Storage Scale's existing way of operating. It keeps Storage Scale as the main authority for how the filesystem behaves.

Comparing IBM Storage Scale Container Native and CSI: Architectural and operational context

A common source of confusion is the assumption that IBM Storage Scale Container Native behaves similarly to a Container Storage Interface (CSI) driver. While both enable storage consumption from Kubernetes, they differ fundamentally in architecture and operational behavior.

CSI-based solutions typically include the following characteristics:

- Provision block or file volumes on a per-workload basis
- Abstract underlying storage behind Kubernetes-managed lifecycle semantics
- Rely heavily on Kubernetes primitives for attachment, detachment, and recovery

In contrast, IBM Storage Scale Container Native includes the following characteristics:

- Provides shared filesystem backed by IBM Storage Scale rather than per-volume isolation
- Delegates all data-path behavior to IBM Storage Scale
- Uses Kubernetes primarily as a control and orchestration plane while Storage Scale remains responsible for data management, consistency, performance, and recovery

These differences have direct implications for scalability, failure handling, performance characteristics, and operational responsibilities. Understanding this difference at the beginning helps avoid incorrect assumptions and provides the right context before diving deeper into the architecture and deployment models.

Technical factors driving IBM Storage Scale Container Native adoption

IBM Storage Scale Container Native is typically adopted in environments driven by one or more of the following technical requirements:

- Shared data access: In many enterprise environments, multiple workloads—both containerized and non-containerized—need concurrent access to the same datasets. A shared filesystem model simplifies this requirement and avoids unnecessary data duplication.
- High-performance I/O: Some applications require consistently low latency and high throughput. Introducing additional abstraction layers can sometimes limit performance tuning or introduce operational overhead. IBM Storage Scale Container Native allows workloads to leverage the native capabilities of IBM Storage Scale directly.
- Operational continuity: Organizations with existing IBM Storage Scale deployments often want to extend those environments into Kubernetes without redesigning their storage architecture. IBM Storage Scale Container Native enables that extension while preserving established policies, data placement strategies, and operational workflows.
- Lifecycle coordination: While data handling remains within Storage Scale, configuration and orchestration still need to align with Kubernetes-managed application lifecycles. IBM Storage Scale Container Native enables this alignment without shifting full storage lifecycle control into Kubernetes.

Scope and non-scope: Technical view

This section defines what is covered in this Redbooks publication and what is intentionally excluded. The objective is to concentrate on IBM Storage Scale Container Native behavior and integration characteristics rather than providing detailed product documentation.

In scope

- The architectural principles behind IBM Storage Scale Container Native and the roles of its core components
- Control plane and data plane responsibilities
- Key considerations influencing deployment models
- Operational lifecycle implications at a conceptual level
- Expected failure behaviors and high-level recovery characteristics

It emphasizes understanding how IBM Storage Scale Container Native behaves within a Kubernetes or Red Hat OpenShift integrated Storage Scale environment and how responsibilities are logically structured.

Out of scope

- Internal GPFS algorithms, metadata structures, or filesystem implementation details
- IBM Storage Scale System hardware configuration specifics or performance tuning guidance
- Detailed command syntax, configuration parameters, or step-by-step operational runbooks
- Disaster recovery architectures not specific to IBM Storage Scale Container Native integration

This scoped approach keeps the discussion focused on IBM Storage Scale Container Native behavior and integration rather than duplicating reference or implementation documentation.

Hardware and software requirements

This Redbooks publication is based on the latest available release of IBM Storage Scale Container Native. At the time of writing, the latest version is 6.0.0, which was used as the reference test environment throughout this publication.

For detailed hardware and software prerequisites, including supported platforms and compatibility considerations, refer to the official IBM Storage Scale Container Native documentation below:

- Hardware requirements
- Software requirements

Key technical takeaways

Before proceeding to the in-depth architectural analysis in section 2, understand the following principles:

- IBM Storage Scale Container Native integrates Kubernetes or Red Hat OpenShift intent with IBM Storage Scale management. It does not move storage logic into KKubernetes or Red Hat OpenShift.
- Kubernetes or Red Hat OpenShift control plane availability is critical for management operations but is not part of the data I/O path.
- IBM Storage Scale remains the system of record for filesystem integrity, performance, and recovery.
- Achieving consistent operational outcomes requires a well-defined separation of responsibilities across the container platform (Kubernetes/Red Hat OpenShift) and the underlying storage layer.

These principles establish the foundation for the detailed architectural and operational discussions that follow.

IBM Storage Scale Container Native architecture and platform considerations

This section establishes the architectural foundation of IBM Storage Scale Container Native and examines how it integrates with Kubernetes and Red Hat OpenShift platforms. It explains the critical separation between the management plane (Kubernetes or Red Hat OpenShift) and the data plane (IBM Storage Scale), which ensures storage operations continue even during control plane outages. It also explores platform-specific differences in lifecycle management, security enforcement, networking, and operational practices that influence IBM Storage Scale Container Native deployment and troubleshooting. Additionally, this section covers storage layer options—remote, local, and direct-attach—and their respective use cases and configuration requirements.

Architecture of Red Hat OpenShift and Kubernetes in an IBM Storage Scale Container Native deployment

IBM Storage Scale Container Native combines IBM Storage Scale with Kubernetes or Red Hat OpenShift's declarative state. While Kubernetes or Red Hat OpenShift serves as the main management and orchestration control plane, IBM Storage Scale functions as the data plane. It handles filesystem metadata, locking, I/O operations, and so on.

This clear division between management and data planes is a key principle of IBM Storage Scale Container Native. Kubernetes shows the desired state through declarative APIs and Custom Resources. Meanwhile, IBM Storage Scale carries out all data-path operations independently. This means that Storage Scale can keep processing application I/O even if Kubernetes control plane components are degraded or unavailable.

Architectural Model and Layering outlines the reference architecture for IBM Storage Scale Container Native deployments. The architectural model described here applies to both Red Hat OpenShift and upstream Kubernetes unless stated otherwise. Red Hat OpenShift and Upstream Kubernetes Considerations for IBM Storage Scale Container Native discusses platform-specific differences in behavior.

Architectural model and layering

An IBM Storage Scale Container Native deployment is composed of multiple architectural layers and components, each with a clearly defined scope of responsibility. These layers span from the underlying physical infrastructure to the Kubernetes storage abstractions consumed by applications.

A high-level overview

- Red Hat OpenShift forms the management plane, responsible for expressing intent, orchestrating lifecycle operations, and tracking system state.
- IBM Storage Scale forms the data plane, responsible for filesystem semantics, metadata consistency, and all I/O execution.
- IBM Storage Scale Container Native operators act as the translation layer, continuously reconciling Kubernetes intent with Storage Scale execution.

This layered separation enables strong fault isolation, predictable behavior during control plane failures, and clear troubleshooting boundaries.

The scope of this IBM Redbooks® publication is limited to IBM Storage Scale Container Native only. Management plane (Red Hat OpenShift cluster) is not discussed in this publication.

The following figure illustrates the IBM Storage Scale Container Native Architecture and the separation of management and data planes.

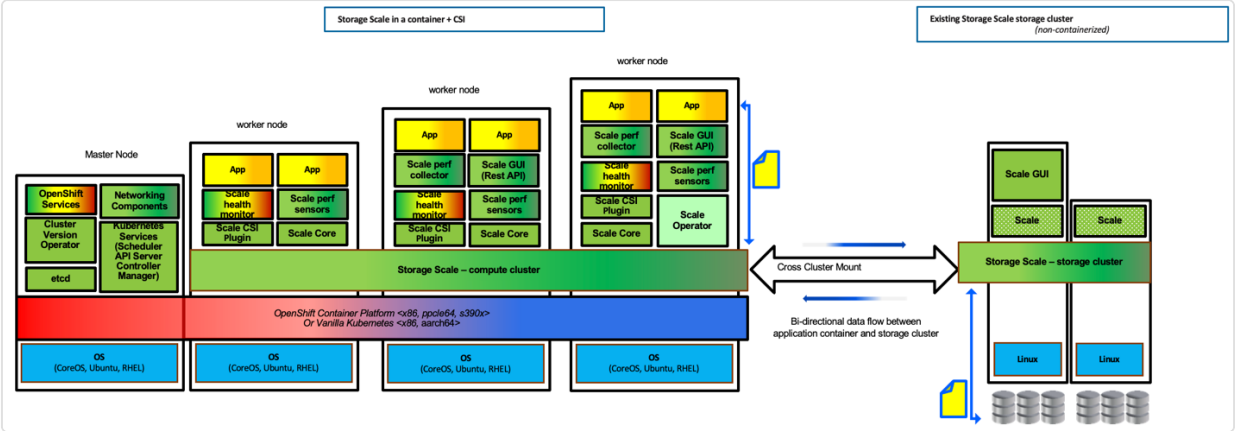

Figure: IBM Storage Scale Container Native Architecture Diagram showing separation of management plane (Red Hat OpenShift/Kubernetes) and data plane (IBM Storage Scale)

The figure shows a high-level overview of IBM Storage Scale Container Native and IBM Storage Scale Container Storage Interface (CSI) driver, which are implemented in this document, and their interactions with the non-containerized IBM Storage Scale storage server.

> **Note:** Red Hat OpenShift cluster configuration such as number of master-nodes, worker-nodes shown in the image are not standard and for illustrative purposes only. Single master is not allowed for standard Red Hat OpenShift Container Platform (OCP) clusters. Check Red Hat OpenShift product documentation for more details on master node count.

In recent years, enterprise IT environments have increasingly adopted container platforms such as Kubernetes and Red Hat OpenShift. Storage solutions have evolved to integrate more easily with containerized environments. IBM Storage Scale addresses this need through CNSA, enabling flexible deployment models for modern applications.

As shown in the following figure, the initial CNSA architecture follows a remote mount model. In this design, the container platform hosts a containerized instance of Storage Scale that operates purely as a remote mount cluster. The actual file systems and storage resources remain hosted on an existing, external Storage Scale storage cluster which is not containerized.

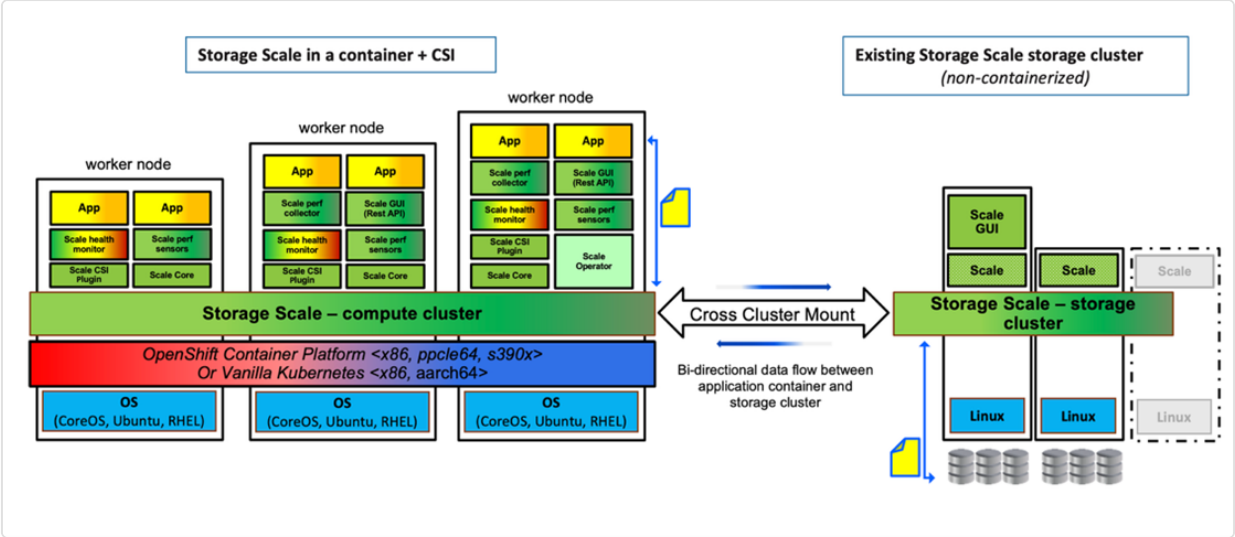

Figure: IBM Storage Scale Container Native Architecture Diagram showing separation of management and data planes

Inside the compute cluster, each worker node runs containerized components such as the Storage Scale core, CSI plugin, performance monitoring services, and GUI alongside application workloads.

However, these components do not manage storage locally but instead, they access data from the remote storage cluster using cross cluster mount capabilities, enabling a bidirectional data flow between applications and storage. Overall, this initial remote mount cluster approach enabled enterprises to bridge traditional storage environments with containerized workloads, providing a foundational step toward fully CNSA architectures.

As CNSA requirements increased, IBM Storage Scale extended the capabilities of CNSA beyond the initial remote mount only model to support more advanced deployment layout. One such enhancement is the direct storage attachment model, as shown in the following figure.

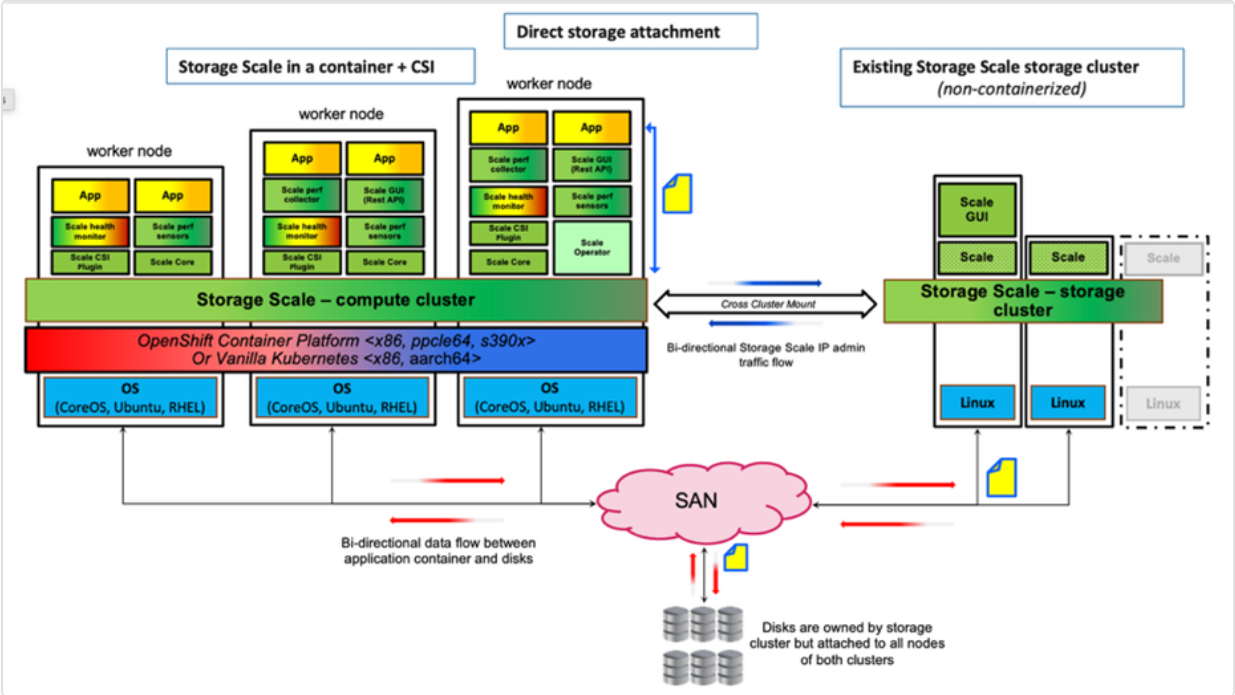

Figure: IBM Storage Scale Container Native Architecture Diagram showing separation of management and data planes

The containerized compute cluster running on platforms such as Kubernetes or Red Hat OpenShift not only acts as a client to the external Storage Scale cluster but also has direct access to the underlying storage devices through a SAN. The disks remain logically owned and managed by the storage cluster, but are physically accessible to nodes in both the compute and storage clusters.

The following figure shows how the architecture spans multiple LPARs, a common deployment model in IBM Z® environments. The containerized compute cluster runs within one LPAR A, leveraging Red Hat OpenShift on Red Hat OS, and the traditional Storage Scale storage cluster is deployed in a separate LPAR B.

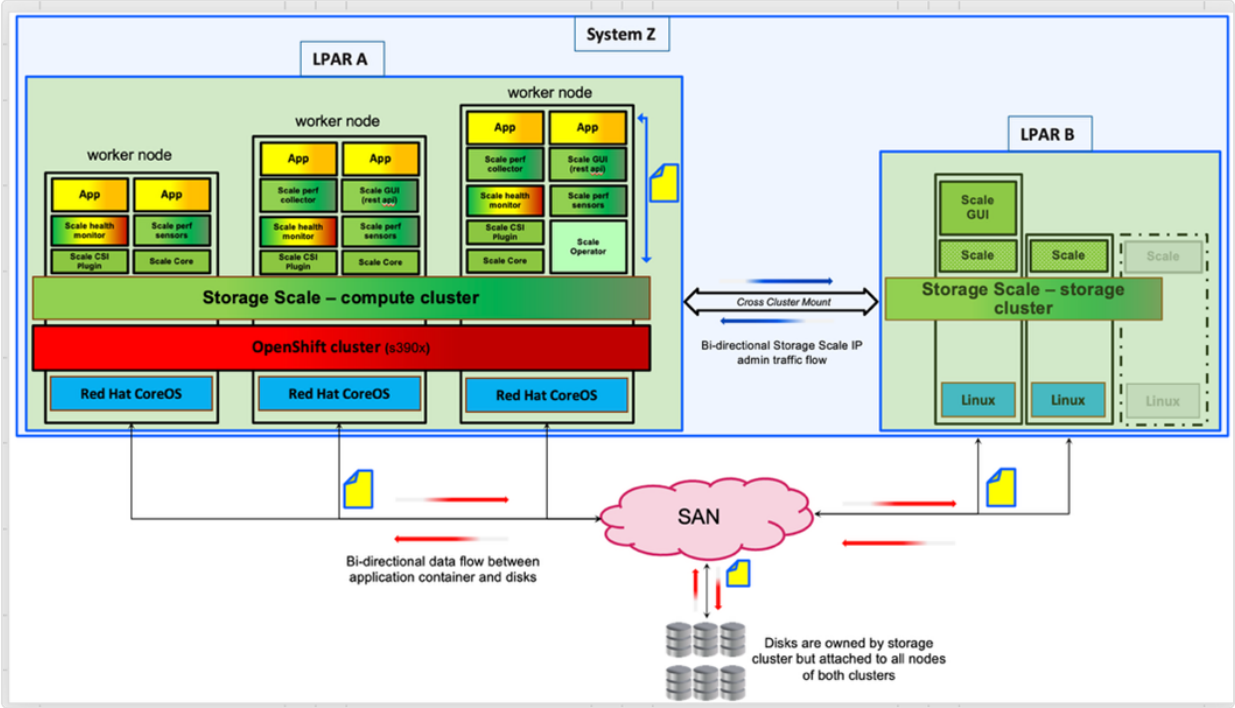

Figure: IBM Storage Scale Container Native Architecture Diagram showing direct storage attachment

Both partitions are connected through a SAN, allowing direct disk access across LPARs, while still maintaining logical ownership of disks within the storage cluster while the cross cluster mount capabilities enable seamless data access between the containerized applications and the storage cluster, with bidirectional administrative and data flows.

As it evolved more, it enables a fully integrated deployment model where storage services are no longer dependent on an external cluster. The following figure shows the local disk configuration with shared access, representing a significant step toward a fully CNSA architecture, where the container platform running on Kubernetes or Red Hat OpenShift hosts a complete Storage Scale cluster within the compute environment itself. All Storage Scale components, including core services, CSI plugin, monitoring tools, and management interfaces, are deployed as containers across worker nodes. There is no separate external storage cluster.

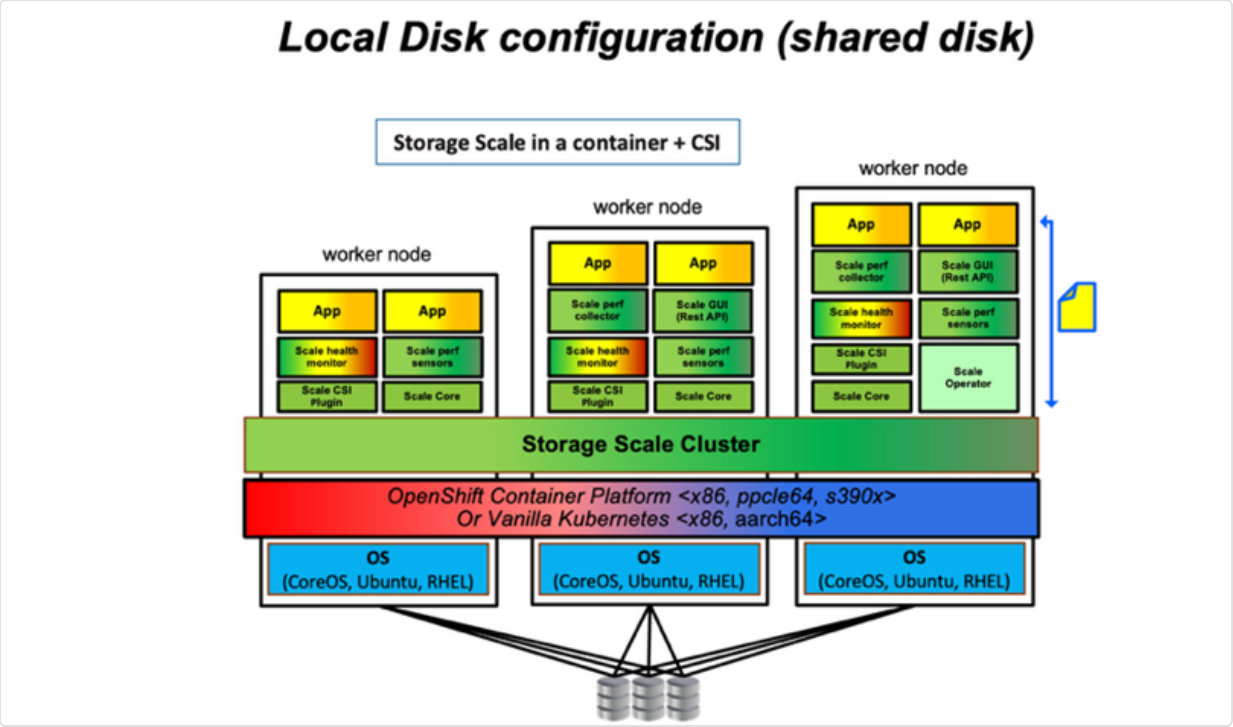

Figure: IBM Storage Scale Container Native Architecture Diagram showing direct storage attachment

Now it introduces a fully optimized shared disk direct attachment model, not in the need for cross cluster mounts entirely, which represents a significant advancement in simplifying architecture while improving performance. As shown in the following figure, storage devices are directly attached to all worker nodes through a shared Fibre Channel SAN, enabling the creation of Storage Scale NSDs with the cluster running entirely within the container environment to manage both data and metadata paths without requiring any external storage cluster or cross-cluster mounts.

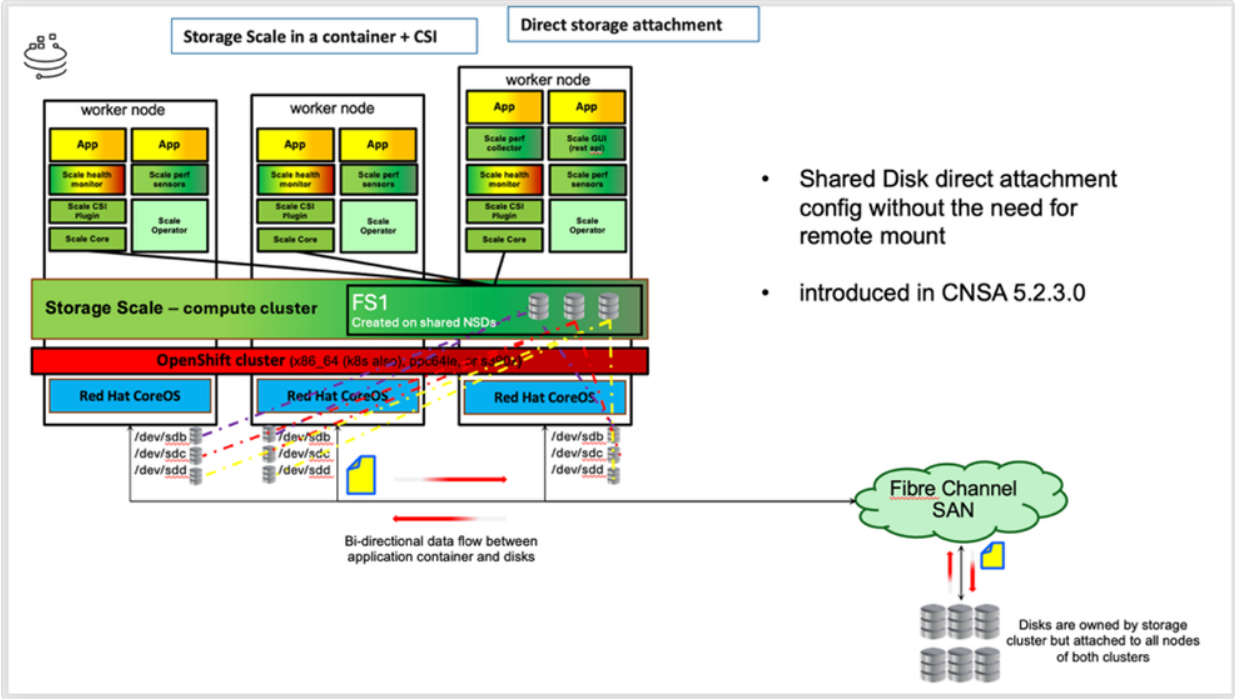

Figure: IBM Storage Scale Container Native Architecture Diagram showing local disk configuration with shared access

Deployment Type	How It Works	What Changes / Improves
Remote Mount	CNSA works only as a client and pulls data from an external IBM Storage Scale cluster.	Good starting point since it uses existing storage, but everything depends on the remote cluster.
Direct Attach + Remote	The compute cluster can access both remote storage and directly attached disks through SAN.	Adds better performance with direct disk access while still keeping the existing setup.
Direct Attach on LPAR	Same hybrid approach, but deployed across LPARs on IBM Z with shared disk access.	Brings separation and flexibility benefits of LPARs along with the hybrid model.
Local Filesystem	Storage Scale runs completely inside the container cluster using local or shared disks.	No need for an external storage cluster, making it a more self-contained setup.
Direct Attach (Optimized shared disk direct attachment model)	Shared disks are directly attached across all worker nodes, without any remote mounts.	Simplifies setup with great performance as everything is directly accessible.

IBM Storage Scale Container Native operators

IBM Storage Scale Container Native operators act as the control logic bridging Kubernetes intent with IBM Storage Scale execution.

IBM Storage Scale Container Native uses the following operators:

- IBM Storage Scale Operator
- IBM Storage Scale CSI Operator

Operators continuously watch IBM Storage Scale Container Native Custom Resources stored in etcd, compare desired state with observed state, and invoke Storage Scale commands on cluster nodes. Operators do not maintain independent state. All states are `Persisted` in the Kubernetes API.

Failures at this layer typically appear as stalled reconciliations, incomplete status updates, or resources blocked by finalizers or insufficient permissions.

IBM Storage Scale runtime: Data plane

The IBM Storage Scale runtime represents the actual data plane and includes:

- `mmfsd` daemons
- NSD servers
- CES services, if deployed
- Optional features such as AFM, encryption, and ILM

Although Storage Scale is managed through Kubernetes in an IBM Storage Scale Container Native deployment, it remains a node-based distributed filesystem. Kubernetes does not participate in metadata locking, data placement, or I/O execution.

This separation ensures that Storage Scale can continue servicing application I/O even when Kubernetes control plane components are unavailable.

Kubernetes Container Storage Interface abstractions

At the top of the stack, Kubernetes storage abstractions expose IBM Storage Scale to applications using native Kubernetes constructs:

- `PersistentVolumeClaims` (PVCs)
- `PersistentVolumes` (PVs)
- `StorageClasses`

The IBM Storage Scale CSI driver maps PVC requests to Storage Scale filesets and manages lifecycle operations such as creation, deletion, and expansion. From the application perspective, Storage Scale appears as standard Kubernetes storage while preserving native filesystem semantics.

Operational impact of control plane failures

A defining characteristic of IBM Storage Scale Container Native is the separation of management and data planes. When the Kubernetes control plane is degraded or unavailable, Storage Scale data-path operations typically continue uninterrupted.

The following table provides an overview of the operational behavior during Kubernetes control plane outages:

Function	Behavior
Existing GPFS mounts	Continue to function
Ongoing application I/O	Continues
New filesystems or filesets	Not possible
PVC provisioning	Not possible
Configuration changes	Not possible
Operator-driven recovery	Blocked

Architectural summary

In an IBM Storage Scale Container Native deployment, Kubernetes and Red Hat OpenShift serve as the authoritative management control plane, with etcd acting as the single source of 'Desired State'. IBM Storage Scale executes all data and metadata operations independently of Kubernetes, ensuring data availability even during control plane outages.

This layered architecture provides strong fault isolation, predictable behavior, and clear operational boundaries.

Red Hat OpenShift and upstream Kubernetes considerations for IBM Storage Scale Container Native

The IBM Storage Scale Container Native architectural model described in the previous section applies equally to both Red Hat OpenShift and upstream Kubernetes deployments. The differences discussed in this section affect platform behavior and operational characteristics but do not alter the underlying architecture.

Although Red Hat OpenShift is built on upstream Kubernetes, it adds its own platform services, default configurations, and security controls. These differences impact how IBM Storage Scale Container Native is deployed, operated, and supported on upstream Kubernetes and Red Hat OpenShift.

This section describes the platform-specific considerations that affect IBM Storage Scale Container Native deployment practices, lifecycle management, security enforcement, networking behavior, observability, and support workflows. Understanding these differences is essential for designing resilient IBM Storage Scale Container Native environments and for operating them effectively over time.

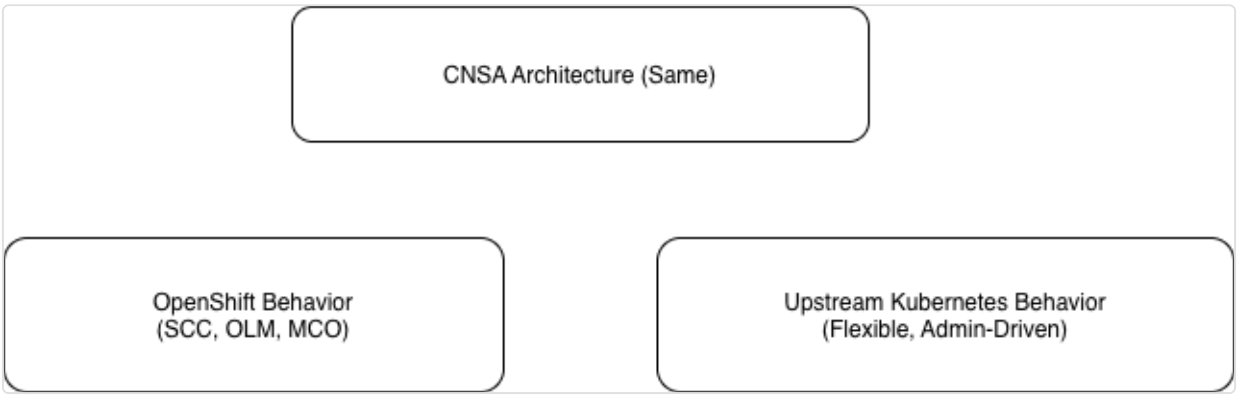

Figure: Diagram highlighting the architectural invariance of IBM Storage Scale Container Native across platforms, while emphasizing differences in platform-specific behavior and enforcement

Installation and lifecycle management

Upstream Kubernetes installations are typically user-driven and vary depending on tooling such as kubeadm, custom automation, or cloud-provider–managed services. As a result, control plane configuration, upgrade sequencing, and node lifecycle behavior are highly dependent on administrator practices.

Red Hat OpenShift provides an integrated installation and lifecycle management framework that treats the cluster as a cohesive platform. Control plane components, worker nodes, and platform services are installed, upgraded, and validated together.

For IBM Storage Scale Container Native deployments, this distinction has several implications:

- In Red Hat OpenShift, control plane upgrades and node configuration changes are tightly managed by built-in operators to ensure cluster stability and consistency.
- Rolling upgrades may temporarily pause pod scheduling or operator reconciliation.
- Host-level modifications must follow Red Hat OpenShift-supported mechanisms.
- IBM Storage Scale Container Native deployments on Red Hat OpenShift should be designed to tolerate transient reconciliation pauses and to align with platform-driven lifecycle events.

Operating system and node configuration

Upstream Kubernetes commonly runs on administrator-managed Linux distributions such as Red Hat Enterprise Linux. Host-level tuning for IBM Storage Scale is typically applied using automation tools or direct configuration changes.

Red Hat OpenShift uses Red Hat Enterprise Linux CoreOS (RHCOS), which is immutable and managed declaratively through the Machine Config Operator (MCO). Direct host modifications are not persistent and are reverted during reconciliation or upgrades.

For IBM Storage Scale Container Native deployments, this results in the following considerations:

- All persistent host configuration on Red Hat OpenShift must be expressed through MachineConfig objects.
- Node reboots are expected during configuration changes.
- Storage Scale host prerequisites must align with Red Hat OpenShift-supported kernel and OS versions.

Security model and privilege enforcement

The security model is one of the most significant differences between Red Hat OpenShift and upstream Kubernetes.

Upstream Kubernetes relies primarily on RBAC and Pod Security Standards, with privileged workloads enabled through pod specifications and role bindings. Red Hat OpenShift introduces SecurityContextConstraints (SCCs), which explicitly define what privileges a pod or service account may request.

IBM Storage Scale Container Native deployments have the following security requirements:

- IBM Storage Scale pods require privileged access, hostPath mounts, and specific Linux capabilities.
- On Red Hat OpenShift, service accounts must be explicitly bound to appropriate SCCs.
- Misconfigured SCCs commonly result in pod scheduling or startup failures, even when RBAC permissions are correct.
- Security configuration should be validated early in the deployment process to prevent privilege-related failures.

Networking and DNS

Upstream Kubernetes supports multiple Container Network Interface (CNI) implementations, selected and managed by the cluster administrator. Network behavior and routing semantics may therefore vary between environments.

Red Hat OpenShift standardizes on OVN-Kubernetes and tightly integrates networking with DNS, ingress, and routing services.

IBM Storage Scale Container Native environments have the following networking considerations:

- Storage Scale daemon discovery and operator communication depend on stable DNS resolution.
- Networking failures may affect both Kubernetes services and Storage Scale communication paths.
- Red Hat OpenShift networking behavior is more prescriptive but generally more predictable.

Image registry and authentication

In upstream Kubernetes, container image registry configuration and authentication are administrator-defined. Image pull secrets are typically managed at the namespace or service account level.

Red Hat OpenShift integrates entitlement management and image pull secrets at the platform level. CNSA can benefit from the cluster-wide pull secret for registry.redhat.io, but CNSA's own IBM images (from icr.io) still require a manually configured IBM Entitlement Key.

IBM Storage Scale Container Native deployments have the following image registry requirements:

- Red Hat OpenShift clusters must be configured with entitlement pull secrets to access IBM container images.
- Image pull failures are often cluster-wide and affect all Scale components simultaneously.

> **Note:** This is the same for both OCP and Kubernetes.

Operator lifecycle and platform integration

Both Red Hat OpenShift and upstream Kubernetes support operators, but Red Hat OpenShift provides deeper integration through the Operator Lifecycle Manager (OLM).

This integration affects IBM Storage Scale Container Native deployments in several ways:

- Scale operators are typically installed and managed through OLM on Red Hat OpenShift.
- Operator upgrades follow defined channels and approval workflows.
- Operator health and status are visible through Red Hat OpenShift-specific APIs and consoles.
- In upstream Kubernetes environments, operators are commonly deployed manually or through Helm, requiring greater administrative discipline to maintain lifecycle consistency.

Monitoring and observability

Monitoring and observability capabilities differ significantly between platforms. Upstream Kubernetes monitoring solutions vary by deployment and may be optional. Metrics collection and alerting must be explicitly designed and maintained. Red Hat OpenShift includes an integrated Prometheus-based monitoring stack, providing immediate visibility into control plane and node health.

IBM Storage Scale Container Native deployments have the following observability considerations:

- Red Hat OpenShift offers built-in metrics relevant to operator health and CSI behavior.
- Equivalent observability on upstream Kubernetes requires additional tooling and configuration.

Platform comparison summary

The following table summarizes the most relevant differences between Red Hat OpenShift and upstream Kubernetes for IBM Storage Scale Container Native deployments.

Area	Red Hat OpenShift	Upstream Kubernetes	IBM Storage Scale Container Native Impa
Host OS	RHCOS, immutable	Admin-managed	Declarative tuning on OCP
Security	SCC + RBAC	RBAC + Pod Security	SCC planning required
Networking	OVN-Kubernetes	Multiple CNIs	Predictable behavior on OCP
Operators	OLM-managed	Manual	Structured lifecycle on OCP
Monitoring	Built-in	Optional	Better visibility on OCP

Support and troubleshooting implications

From a support and operational perspective, Red Hat OpenShift and upstream Kubernetes present different troubleshooting models for IBM Storage Scale Container Native deployments.

Red Hat OpenShift enforces strong platform guardrails through integrated lifecycle management, security enforcement, and configuration reconciliation. While these controls reduce configuration drift and improve long-term stability, they also limit ad hoc configuration changes during active troubleshooting. As a result, IBM Storage Scale Container Native troubleshooting on Red Hat OpenShift typically begins with platform-level validation, including control plane health, node configuration compliance, SecurityContextConstraints, and operator lifecycle status, before proceeding to Storage Scale–specific analysis.

In contrast, upstream Kubernetes offers greater flexibility in host configuration, networking, and operational tooling. This flexibility can simplify certain investigative workflows but places greater responsibility on administrators to maintain consistency and compatibility across the cluster. Troubleshooting in these environments often involves verifying that platform-level assumptions have not been violated by manual changes or divergent configurations.

These differences lead to distinct support workflows:

- Red Hat OpenShift: Platform health and compliance checks precede Scale-specific investigation.
- Upstream Kubernetes: Host configuration and administrator-applied changes are often examined earlier.

Understanding these operational differences is essential when designing IBM Storage Scale Container Native environments, planning upgrades, and diagnosing issues across both platforms.

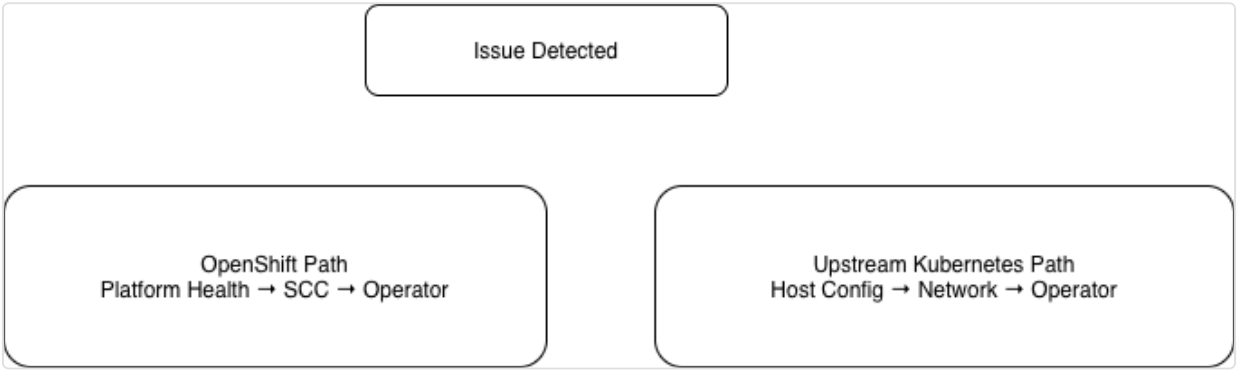

Figure: Diagram illustrating the typical troubleshooting flow for IBM Storage Scale Container Native deployments on Red Hat OpenShift and upstream Kubernetes platforms

Considerations summary

Red Hat OpenShift and upstream Kubernetes implement the same IBM Storage Scale Container Native architecture but impose different operational and support models.

Red Hat OpenShift is generally preferred for production deployments where predictability, integrated lifecycle management, and standardized behavior are priorities. Upstream Kubernetes remains suitable for development, testing, and specialized environments where flexibility outweighs platform standardization.

Designing and operating IBM Storage Scale Container Native deployments with these platform-specific considerations in mind improves reliability, supportability, and long-term operational success.

Storage layers and storage systems in IBM Storage Scale Container Native

IBM Storage Scale Container Native supports multiple storage layers that define how storage is provisioned, accessed, and managed for container workloads. These storage layers determine whether the file system is sourced from an external IBM Storage Scale cluster or created using local or directly attached storage within the Kubernetes environment.

Remote storage layer

In the Remote storage layer, IBM Storage Scale Container Native consumes storage from an existing IBM Storage Scale storage cluster. The file system is created and managed entirely on the remote cluster and is mounted into the IBM Storage Scale Container Native environment for containerized access.

The remote storage layer has the following key characteristics:

- Storage is hosted on a non-containerized remote IBM Storage Scale cluster.
- IBM Storage Scale Container Native does not manage disks or create the file system.
- Remote cluster must expose the filesystem through the IBM Storage Scale GUI (REST API).
- Applications access storage through the IBM Storage Scale CSI driver.

Local storage layer

The Local storage layer uses shared local disks or volumes attached to Kubernetes worker nodes. IBM Storage Scale Container Native creates and manages the IBM Storage Scale cluster and file system using these locally available shared disks, eliminating dependency on an external storage cluster.

The local storage layer has the following key characteristics:

- Storage is provided by locally available shared disks.
- IBM Storage Scale Container Native manages disk discovery, file system creation, and lifecycle.
- `LocalDisk` and `Filesystem` custom resources are required.
- Suitable for standalone or edge Kubernetes environments.

Direct-attach storage layer

The Direct-Attach storage layer is a storage model where Kubernetes worker nodes have direct access to SAN block devices. These disks are directly attached to the nodes of both Storage Scale remote cluster and IBM Storage Scale Container Native cluster.

The direct-attach storage layer has the following key characteristics:

- Worker nodes use SAN, NVMe, SAS, or mixed direct-attached storage.
- Provides high throughput and low-latency access.

- Storage Scale runs inside Kubernetes but closely aligns with traditional deployments.
- Requires careful network and node configuration.

Deployment considerations for storage layers

When deploying any storage layer in IBM Storage Scale Container Native, several architectural and networking considerations must be taken into account to ensure proper functionality and performance:

- All core Storage Scale pods require static IP communication.
- Either host networking or CNI networking must be selected.
- Host networking provides simplicity but limited isolation.
- CNI networking offers isolation but requires additional configuration.

Storage layer comparison

The following table provides a comparison of the storage layer components.

Storage Layer	Storage Source	File System Owner	External Dependency	Typical Use Case
Remote	External Storage Scale cluster	Remote cluster	Yes	Enterprise shared storage
Local	Shared local disks	IBM Storage Scale Container Native	No	Standalone or edge clusters
Direct-Attach	SAN / NVMe / SAS disks	IBM Storage Scale Container Native	No	High-performance on-prem clusters

Comparing local and remote file systems

IBM Storage Scale Container Native supports two types of file systems: remote file system and local file system. Each type differs in terms of storage source, deployment model, and configuration requirements.

Remote file system

A remote file system is a file system custom resource in IBM Storage Scale Container Native that uses an existing file system exported from a remote IBM Storage Scale storage cluster. In this model, IBM Storage Scale Container Native mounts and consumes the pre-existing file system without creating or managing the storage itself. Only the `Filesystem` (FS) custom resource needs to be created in IBM Storage Scale Container Native.

The following prerequisites and configuration requirements apply:

- The remote IBM Storage Scale cluster must have the GUI configured and operational.
- An IBM Storage Scale Container Native service user must be created on the remote cluster and assigned to the `ContainerOperator` group.
- A Kubernetes secret must be created in IBM Storage Scale Container Native using the `ContainerOperator` user credentials.
- The secret must be referenced in the `gui.secretName` field of the `RemoteCluster` custom resource.

Local file system

Create a local file system by using local disks or shared volumes attached to the Kubernetes worker nodes. In this approach, IBM Storage Scale Container Native creates and manages the file system using locally available storage, removing dependency on an external IBM Storage Scale cluster.

Use the following steps to create a local file system:

1. Specify device names in the `Cluster` custom resource, if required.
2. Label Kubernetes nodes that have connectivity to the shared disks.
3. Create a `LocalDisk` custom resource for each disk used by the file system.
4. Create a `Filesystem` custom resource referencing the Local Disk resources.

> **Note:** If the disks use device names such as `sd*`, `hd*`, `vd*`, `scini*`, `pmem*`, `nvm*`, `dm-*`, `vpath*`, `dasd*`, or `emcpower*`, explicit device definition is not required. For other naming conventions, device names must be explicitly specified.

This table provides a summary of the basic difference between remote and local file systems.

Category	Remote File System	Local File System
Source of Storage	Uses an existing file system exported from a remote IBM Storage Scale cluster.	Uses local disks/volumes present on Kubernetes nodes.
Where Data Resides	Data resides on the external Storage Scale cluster.	Data resides on local node disks.
Required IBM Storage Scale Container Native Resources	Only Filesystem CR is created.	LocalDisk CR + Filesystem CR required.
Dependency on Existing Scale Cluster	Mandatory	Not required
Use Case	Reusing existing enterprise Scale storage.	Deploying Scale storage locally within Kubernetes.
Performance Characteristics	Dependent on network throughput.	Dependent on local disk hardware.
Setup Complexity	Lower	Higher
Scalability	Scales with remote cluster.	Scales with local node disk availability.
Fault Tolerance	Depends on remote cluster redundancy.	Depends on local node redundancy.
Typical Scenarios	Multi-cluster, enterprise environments.	Edge or independent Kubernetes clusters.

Key technical takeaways

This chapter established the architectural framework for IBM Storage Scale Container Native deployments on Kubernetes and Red Hat OpenShift. Central to this architecture is the strict separation between the management plane, provided by Kubernetes or Red Hat OpenShift, and the data plane, implemented by IBM Storage Scale. This separation ensures that data and metadata operations remain available and consistent even during control plane degradation, providing predictable behavior and strong fault isolation.

Declarative Kubernetes control constructs, combined with operator-driven reconciliation, enable centralized lifecycle management while preserving IBM Storage Scale's independent execution of metadata handling and I/O operations. Applications interact with storage through standard Kubernetes abstractions, while the underlying filesystem

maintains native semantics and behavior. Clearly defined functional boundaries across these layers are fundamental to reliable operations and effective problem determination.

Although Red Hat OpenShift and upstream Kubernetes follow the same architectural model, they differ in lifecycle management, security enforcement, observability, and operational rigor. These platform characteristics influence deployment practices, upgrade strategies, and support workflows and must be incorporated into architectural planning.

The supported storage layers, such as remote, local, and direct-attach, provide flexibility in how storage is sourced and managed, ranging from integration with existing enterprise Storage Scale environments to fully container-native deployments optimized for performance and locality.

The architectural principles and deployment models described in this chapter form the baseline for the installation, configuration, and operational procedures presented in the next chapter.

Deploying IBM Storage Scale Container Native on Kubernetes

This section provides a comprehensive guide to deploying IBM Storage Scale Container Native within a Kubernetes environment. It covers the complete deployment lifecycle, from infrastructure preparation and storage cluster configuration through platform validation, IBM Storage Scale Container Native installation, and troubleshooting.

This section demonstrates a hybrid architecture where Kubernetes serves as the control plane and application execution environment while an external Storage Scale cluster delivers scalable, high-performance storage services. Key topics include validating the Kubernetes platform baseline, preparing the external Storage Scale cluster for CSI integration, installing and configuring IBM Storage Scale Container Native operators and custom resources, and establishing both remote and local filesystem access. This section emphasizes prerequisite validations and systematic verification steps to ensure a successful deployment that enables Kubernetes workloads to dynamically consume Storage Scale storage through declarative APIs and CSI-based provisioning.

Infrastructure preparation

Virtualized infrastructure configuration

The validation and deployment of IBM Storage Scale Container Native was performed in a virtualized environment hosted on a Kernel-based Virtual Machine (KVM) server. The host system was provisioned with 32 virtual CPUs and 754 GB of memory, providing sufficient resources to simulate a production-like deployment topology.

Virtual machine layout

The environment consists of a combination of Kubernetes and IBM Storage Scale nodes, with clearly defined roles and resource allocations. The following table summarizes the resource allocation for each virtual machine type:

Node Role	vCPU	Memory
Control plane (master)	6	32 GB
Worker node 1	8	64 GB
Worker node 2	8	64 GB
Storage Scale node 1	4	64 GB
Storage Scale node 2	4	64 GB
Storage Scale node 3	4	64 GB

This configuration reflects a hybrid IBM Storage Scale Container Native deployment model, where Kubernetes control and compute nodes are separated from the external IBM Storage Scale storage cluster.

Virtual machine state verification

All virtual machines were verified to be in a running state using the `virsh` command-line interface:

```
virsh list --all

Example output:

Id   Name                State

----------------------------------

22   c676f5u24-master    running
23   c676f5u24-scale1    running
24   c676f5u24-scale2    running
25   c676f5u24-scale3    running
26   c676f5u24-worker1   running
27   c676f5u24-worker2   running
```

This confirms that all required nodes for both Kubernetes and Storage Scale clusters are operational.

Disk configuration and storage layout

Each virtual machine is provisioned with multiple block devices to support operating system, shared storage, and workload-specific data paths. The disk configuration was validated using the following command:

```
for vm in $(virsh list --all --name); do

echo "=== $vm ==="

virsh domblklist $vm

done
```

The resulting configuration highlights a consistent and structured storage design:

- Control plane and worker nodes:
  - vda: Operating system disk (Kubernetes node image)

- vdb: Shared disk (vg_shared_all) accessible across all nodes
- vdc: Kubernetes-specific shared disk (vg_shared_k8s)

Example:

```
vda  → /var/lib/libvirt/images/<node>.k8sprecnsa\_2
vdb  → /dev/vg\_shared\_all/disk\_shared\_all
vdc  → /dev/vg\_shared\_k8s/disk\_shared\_k8s
```

- Storage Scale nodes:
  - vda: Operating system disk (Storage Scale node image)
  - vdb: Shared disk (vg_shared_all) for cluster-wide coordination
  - vdc: Storage Scale–specific shared disk (vg_shared_scale)

  Example:

```
vda  → /var/lib/libvirt/images/<node>.scaleprecnsa
vdb  → /dev/vg\_shared\_all/disk\_shared\_all
vdc  → /dev/vg\_shared\_scale/disk\_shared\_scale
```

- Design considerations

  This storage layout enforces logical separation between the following resources:
  - Cluster-wide shared resources (vg_shared_all)
  - Kubernetes-specific storage domains (vg_shared_k8s)
  - Storage Scale–specific storage domains (vg_shared_scale)

  This separation ensures the following conditions:
  - Isolation between compute and storage workloads
  - Clear mapping of storage responsibilities
  - Flexibility for testing IBM Storage Scale Container Native integration scenarios

Summary

In summary, the described virtualized environment provides a controlled and scalable platform for deploying and validating IBM Storage Scale Container Native. By separating Kubernetes and Storage Scale roles while maintaining shared storage constructs, the environment closely models real-world hybrid deployment architectures.

IBM Storage Scale storage cluster configuration

This section describes the preparation of the IBM Storage Scale storage cluster to support integration with IBM Storage Scale Container Native and the Container Storage Interface (CSI) driver.

The IBM Storage Scale Container Native and CSI operators interact with the storage cluster through the IBM Storage Scale REST API, which is exposed by the GUI stack. To enable secure and controlled communication, a dedicated GUI user must be created with appropriate role-based access. In addition, specific cluster and file system configuration parameters must be set to ensure compatibility with CSI-based provisioning workflows.

Creating the IBM Storage Scale Container Native operator GUI user

For IBM Storage Scale version 6.0.0.x and later, a dedicated GUI user is required for IBM Storage Scale Container Native operations. This user is assigned the `ContainerOperator` role, which provides the minimum required privileges for operator-driven storage management.

To create the IBM Storage Scale Container Native operator user, run the following command on a Storage Scale node:

```
/usr/lpp/mmfs/gui/cli/mkuser cnsa_storage_gui_user -p cnsa_storage_gui_password -g ContainerOperator
```

Example output:

```
EFSSG0019I The user cnsa_storage_gui_user has been successfully created.
EFSSG1000I The command completed successfully.
```

This user is subsequently used by IBM Storage Scale Container Native to authenticate REST API requests to the storage cluster.

Configuring the storage cluster for CSI

To ensure proper operation of the IBM Storage Scale CSI driver, several configuration settings must be verified and adjusted on the storage cluster. These settings primarily address quota behavior, inode management, and filesystem reporting capabilities required for Kubernetes persistent volumes.

Verifying filesystem manager and quota settings

Begin by identifying the filesystem manager nodes:

```
mmlsmgr
```

Example output:

```
file system      manager node

---------------- ------------------

remotefs         c676f5u24-scale1
directdiskfs     c676f5u24-scale3
Cluster manager node: c676f5u24-scale1
```

Next, verify that per-fileset quota enforcement is disabled:

```
mmlsfs remotefs --perfileset-quota
mmlsfs directdiskfs --perfileset-quota
```

Expected output:

```
--perfileset-quota no
```

This confirms that per-fileset quota enforcement is not active, which is a prerequisite for CSI configuration.

Enabling quota enforcement

Quota support must be enabled at the filesystem level:

```
mmchfs remotefs -Q yes

mmchfs directdiskfs -Q yes
```

These commands propagate configuration changes across the cluster asynchronously.

Enabling root fileset quota enforcement

Enable quota enforcement for the root user:

```
mmchconfig enforceFilesetQuotaOnRoot=yes -i
```

This ensures that quota policies are consistently applied, including for administrative operations.

Configuring replication behavior for quotas

To prevent replication from affecting quota accounting, configure the following parameter:

```
mmchconfig ignoreReplicationForQuota=yes -i
```

This setting ensures accurate quota reporting in replicated environments.

Configuring SELinux extended attribute handling

Enable immutable extended attribute handling for SELinux:

```
mmchconfig controlSetxattrImmutableSELinux=yes -i
```

This setting is required for compatibility with containerized workloads running under SELinux enforcement.

Enabling fileset disk usage reporting

Enable fileset-level disk usage reporting:

```
mmchfs remotefs --filesetdf
mmchfs directdiskfs --filesetdf
```

This allows Kubernetes and CSI to accurately report storage consumption at the fileset level.

Enabling automatic inode expansion

To prevent inode exhaustion in dynamically provisioned persistent volumes, enable automatic inode expansion:

```
mmchfs remotefs --auto-inode-limit
mmchfs directdiskfs --auto-inode-limit
```

This configuration ensures that filesystems can scale inode capacity in response to workload demands.

Summary

Proper configuration of the IBM Storage Scale cluster is essential for successful IBM Storage Scale Container Native deployment. The steps outlined in this section produce the following results:

- Secure communication is established through a dedicated GUI user.

- Filesystem quota behavior is aligned with CSI expectations.
- Inode and space management support dynamic provisioning.
- Fileset-level reporting enables accurate storage visibility.

These configurations form the foundation for reliable integration between Kubernetes and IBM Storage Scale through IBM Storage Scale Container Native.

Kubernetes platform baseline and validation

Before installing IBM Storage Scale Container Native, the Kubernetes (or Red Hat OpenShift) platform must be validated to ensure it is healthy, stable, and capable of supporting storage integration workflows. IBM Storage Scale Container Native depends on core Kubernetes services, networking, and API availability; therefore, pre-installation validation is essential.

At a minimum, the following conditions must be satisfied:

- The Kubernetes API server is reachable and responsive
- CoreDNS is operational and correctly resolving internal and external domains
- The container network interface (CNI) is functioning correctly
- The installer has cluster-admin privileges
- Worker nodes have network connectivity to required external services and the Storage Scale cluster

Validating Kubernetes API server availability

Verify that all nodes are in a ready state and that the control plane is accessible:

```
kubectl get nodes
kubectl cluster-info
```

Example output:

```
NAME                STATUS   ROLES           AGE   VERSION
c676f5u24-master    Ready    control-plane   40d   v1.33.0
c676f5u24-worker1   Ready    <none>          40d   v1.33.0
c676f5u24-worker2   Ready    <none>          40d   v1.33.0
```

The expected result is that all nodes report a Ready status and the API server responds without delay.

Validating CoreDNS health and DNS resolution

CoreDNS provides essential name resolution services for Kubernetes workloads. Its availability and correctness must be verified.

Verify CoreDNS pods and services:

```
kubectl -n kube-system get pods -l k8s-app=kube-dns
kubectl -n kube-system get svc coredns
kubectl -n kube-system get endpoints coredns
kubectl -n kube-system get endpointslice -l kubernetes.io/service-name=coredns
```

Ensure that all CoreDNS pods are in a Running state.

Inspect CoreDNS logs:

```
kubectl -n kube-system logs -l k8s-app=kube-dns --since=10m --timestamps
```

No persistent errors should be observed.

Validate DNS resolution from within the cluster:

```
kubectl run dns-test \
    --image=busybox:1.36 \
    --restart=Never \
    --rm -it -- sh
```

Inside the pod:

```
nslookup kubernetes.default.svc.cluster.local
nslookup google.com
```

Successful resolution of both internal and external names confirms proper DNS functionality.

Validating container network interface

The container network must be fully operational to support pod-to-pod and pod-to-service communication:

```
kubectl -n kube-system get pods
```

Verify that networking components such as `calico-node` (or equivalent container network interface (CNI) plugin), `kube-proxy`, and `nodelocaldns` are in a Running state across all nodes.

Additionally, confirm node-level details:

```
kubectl get nodes -o wide
```

This ensures correct node IP assignment and runtime configuration.

Validating external network connectivity

Kubernetes nodes must be able to access external container registries and required services.

From the control plane node:

```
curl -I https://cp.icr.io
curl -I https://registry.k8s.io
curl -I https://quay.io
```

From worker nodes:

```
curl -I https://cp.icr.io
curl -I https://icr.io
```

Successful HTTP responses confirm outbound connectivity.

Validating connectivity to IBM Storage Scale cluster

IBM Storage Scale Container Native requires network connectivity from Kubernetes worker nodes to the Storage Scale cluster.

Validate TCP connectivity (port `1191` – mmfsd):

```
for i in 1 2 3; do
    nc -vz c676f5u24-scale$i 1191
done
```

Validate GUI (REST API) connectivity (port `443`):

```
nc -vz <scale-node-ip> 443
Validate connectivity from inside a pod:
kubectl run nettest --image=busybox:1.36 --restart=Never -- sleep 3600
kubectl exec nettest -- nc -vz c676f5u24-scale1 1191
kubectl exec nettest -- nc -vz <scale-node-ip> 443
```

Remove the test pod:

```
kubectl delete pod nettest
```

This step ensures that pod network paths to Storage Scale are functional, which is critical for CSI operations.

Validating Storage Scale REST API access

Verify that the Kubernetes environment can authenticate and communicate with the Storage Scale REST API using the IBM Storage Scale Container Native GUI user:

```
curl --insecure -u 'cnsa_storage_gui_user:cnsa_storage_gui_password' -X GET https://c676f5u24-scale1/scale

Expected output includes:

"status": {
    "code": 200,
    "message": "The request finished successfully."
}
```

A successful response confirms network connectivity to the GUI endpoint, valid IBM Storage Scale Container Native credentials, and proper REST API functionality.

Summary

This validation process ensures that the Kubernetes platform is fully prepared for IBM Storage Scale Container Native deployment. It confirms control plane availability, DNS resolution, container networking functionality, external connectivity, and end-to-end communication with the Storage Scale cluster. Completing these checks provides a stable foundation for IBM Storage Scale Container Native installation and reduces deployment risk.

Storage Scale baseline: External cluster

In deployments where IBM Storage Scale Container Native integrates with an external IBM Storage Scale cluster, the storage environment must be fully configured, operational, and healthy prior to Kubernetes integration. IBM Storage Scale Container Native does not provision, initialize, or repair the Storage Scale cluster; instead, it manages a remote Storage Scale cluster from which it uses a filesystem through the GUI REST API and GPFS services.

For this reason, validating the baseline health and configuration of the Storage Scale cluster is a critical prerequisite. Any instability or misconfiguration at the storage layer will directly impact IBM Storage Scale Container Native functionality, including remote cluster registration, filesystem access, and CSI-based provisioning.

Validating cluster configuration and node roles

The cluster configuration, node roles, and quorum design must be verified to ensure a stable distributed environment. All CNSA worker nodes must be able to communicate with the daemon nodes by name. Review the cluster information:

```
mmlscluster
```

Example output:

```
GPFS cluster information
========================
GPFS cluster name:         c676f5u24-scale1.pok.stglabs.ibm.com
GPFS cluster id:           3455730563579509587
GPFS UID domain:           c676f5u24-scale1.pok.stglabs.ibm.com
Remote shell command:      /usr/bin/ssh
Remote file copy command:  /usr/bin/scp
Repository type:           CCR

Node  Daemon node name                     IP address   Admin node name                      Designation

-----------------------------------------------------------------------------------------------------

1     c676f5u24-scale1.pok.stglabs.ibm.com  9.114.56.67  c676f5u24-scale1.pok.stglabs.ibm.com  quorum-mana
2     c676f5u24-scale2.pok.stglabs.ibm.com  9.114.56.68  c676f5u24-scale2.pok.stglabs.ibm.com  quorum-mana
3     c676f5u24-scale3.pok.stglabs.ibm.com  9.114.56.71  c676f5u24-scale3.pok.stglabs.ibm.com  quorum-mana
```

This output confirms that the cluster is operating in CCR mode, all nodes are participating in quorum, and node roles are properly distributed.

Validating node state and cluster membership

All nodes must be active and participating in the cluster. List the status of the nodes:

```
mmgetstate -aL
```

Example output:

```
Node number  Node name            Quorum    Nodes up   Total nodes  GPFS state    Remarks
-----------------------------------------------------------------------------------------

1            c676f5u24-scale1     2         3          3             active       quorum node
2            c676f5u24-scale2     2         3          3             active       quorum node
3            c676f5u24-scale3     2         3          3             active       quorum node
```

All nodes should report an active GPFS state, full cluster membership, and quorum participation.

Validating cluster health

Cluster health must be verified across all subsystems:

```
mmhealth cluster show
```

Example output:

```
Component           Total         Failed       Degraded       Healthy          Other

-------------------------------------------------------------------------------------------------------------
NODE                    3              0              0              0              3
GPFS                    3              0              0              0              3
NETWORK                 3              0              0              3              0
FILESYSTEM              2              0              0              2              0
DISK                    2              0              0              2              0
CES                     2              0              0              2              0
CESCLUSTER              1              0              0              1              0
FILESYSMGR              2              0              0              2              0
GUI                     1              0              0              1              0
PERFMON                 2              0              0              2              0
THRESHOLD               2              0              2              0              0
```

All critical components such as GPFS, NETWORK, FILESYSTEM, DISK, and GUI must be healthy. Threshold conditions may appear depending on configuration and do not necessarily indicate a failure.

Validating filesystem configuration and managers

Confirm that filesystems are defined and managed correctly:

```
mmlsmgr
```

Example output:

```
file system      manager node
---------------- ------------------
remotefs         9.114.56.67 (c676f5u24-scale1)
directdiskfs     9.114.56.71 (c676f5u24-scale3)

Cluster manager node: 9.114.56.67 (c676f5u24-scale1)
```

This verifies that each filesystem has an assigned manager node and that cluster manager election is functioning properly.

Validating filesystem mount state

All filesystems intended for IBM Storage Scale Container Native consumption must be mounted on all relevant nodes. List the current mounted filesystems:

```
mmlsmount all -L
```

Example output:

```
File system directdiskfs is mounted on 3 nodes:
    9.114.56.67     c676f5u24-scale1
    9.114.56.71     c676f5u24-scale3
    9.114.56.68     c676f5u24-scale2

File system remotefs is mounted on 3 nodes:
    9.114.56.67     c676f5u24-scale1
    9.114.56.71     c676f5u24-scale3
    9.114.56.68     c676f5u24-scale2
```

This ensures consistent filesystem availability across the cluster.

Validating filesystem version compatibility

Verify that all filesystems are running a supported version:

```
mmlsfs all -V
```

Example output:

```
File system attributes for /dev/directdiskfs:
=============================================
-V                 38.00 (6.0.0.0)

File system attributes for /dev/remotefs:
=========================================
-V                 38.00 (6.0.0.0)
```

The filesystem version must align with the Storage Scale release supported by IBM Storage Scale Container Native. A table showing supported filesystem format levels for each Storage Scale version can be viewed here.

Summary

A healthy external IBM Storage Scale cluster is a strict prerequisite for IBM Storage Scale Container Native deployments. These validation steps confirm cluster configuration and quorum integrity, node availability and GPFS daemon state, CES service readiness when applicable, filesystem availability and mount consistency, and overall subsystem health.

Ensuring that these conditions are met provides a stable and reliable storage backend for IBM Storage Scale Container Native, minimizing deployment issues and enabling consistent CSI operations.

Installing and configuring IBM Storage Scale Container Native on Kubernetes

This section describes the installation and configuration of IBM Storage Scale Container Native on a Kubernetes cluster, including operator deployment, cluster configuration, remote storage integration, and filesystem provisioning. These steps assume that both the Kubernetes platform and the external Storage Scale cluster have been validated as described in previous sections.

Installing prerequisite components

IBM Storage Scale Container Native relies on certificate management for secure communication between components. The cert-manager component must be installed prior to deploying the IBM Storage Scale Container Native operator:

```
kubectl apply -f https://github.com/cert-manager/cert-manager/releases/download/v1.17.4/cert-manager.yaml
```

This command installs the required Custom Resource Definitions (CRDs), services, and controllers for certificate lifecycle management within the cluster.

Preparing Kubernetes nodes

Worker nodes must be labeled appropriately so that IBM Storage Scale Container Native core pods can be scheduled correctly. In the following example, worker nodes are labeled as quorum nodes:

```
kubectl label node c676f5u24-worker1 scale.spectrum.ibm.com/designation=quorum
kubectl label node c676f5u24-worker2 scale.spectrum.ibm.com/designation=quorum
```

Verification:

```
kubectl get nodes --show-labels
```

Proper node labeling is critical because IBM Storage Scale Container Native uses node selectors to determine where GPFS daemon pods are deployed.

Installing the IBM Storage Scale Container Native operator

The IBM Storage Scale Container Native operator and required Kubernetes resources are installed by applying the installation manifest:

```
kubectl apply -f https://raw.githubusercontent.com/IBM/ibm-spectrum-scale-container-native/v6.0.0.x/genera
```

This step creates the following resources:

- Required namespaces (`ibm-spectrum-scale`, `ibm-spectrum-scale-operator`, `ibm-spectrum-scale-dns`, `ibm-spectrum-scale-csi`)
- Custom Resource Definitions (CRDs) for IBM Storage Scale Container Native resources
- RBAC roles and bindings
- Operator deployment and supporting services

Verification:

```
kubectl get namespaces | grep ibm-spectrum-scale
kubectl get pods -n ibm-spectrum-scale-operator
```

The operator pod must be in a Running state before proceeding.

Configuring access to container images

IBM Storage Scale container images are hosted in the IBM Cloud Container Registry (ICR). Access requires an entitlement key, which can be obtained by visiting the container library page here.

Export the entitlement key:

```
export ENTITLEMENT_KEY=<your_key>
```

Create pull secrets for each IBM Storage Scale Container Native namespace:

```
for namespace in ibm-spectrum-scale ibm-spectrum-scale-operator ibm-spectrum-scale-dns ibm-spectrum-scale-
    kubectl create secret docker-registry ibm-entitlement-key -n ${namespace} \
        --docker-server=cp.icr.io \
        --docker-username=cp \
     --docker-password=${ENTITLEMENT_KEY}
done
```

Verify:

```
kubectl get secrets -A | grep ibm-entitlement-key
```

These secrets allow Kubernetes to pull required container images during deployment.

Building the kernel portability layer

The IBM Storage Scale kernel portability layer must be compiled to match the host kernel. This is required for GPFS kernel modules to function correctly.

Create the build configuration:

```
kubectl create -f https://raw.githubusercontent.com/IBM/ibm-spectrum-scale-container-native/v6.0.0.x/gener
```

This configmap defines the logic used by the `mmbuildgpl` init container to build kernel modules dynamically.

Creating the cluster custom resource

The IBM Storage Scale Container Native cluster is defined through a Cluster custom resource. A sample configuration can be downloaded and modified:

```
curl -fs https://raw.githubusercontent.com/IBM/ibm-spectrum-scale-container-native/v6.0.0.x/generated/scal
```

In environments where DNS does not resolve external storage nodes, hostAliases can be defined:

```
spec:
  daemon:
    hostAliases:
      - ip: 9.114.56.67
        hostnames:
          - c676f5u24-scale1.pok.stglabs.ibm.com
      - ip: 9.114.56.68
        hostnames:
          - c676f5u24-scale2.pok.stglabs.ibm.com
      - ip: 9.114.56.71
        hostnames:
          - c676f5u24-scale3.pok.stglabs.ibm.com
```

Apply the cluster resource:

```
kubectl apply -f cluster.yaml
```

Verification:

```
kubectl get cluster
```

Configuring remote storage cluster access

To enable IBM Storage Scale Container Native to consume storage from an external Storage Scale cluster, a `RemoteCluster` resource must be defined.

Create credentials for the Storage Scale GUI user:

```
kubectl create secret generic cnsa-remote-mount-storage-cluster-1 --from-literal=username='cnsa\_storage\_
```

Label the secret:

```
kubectl label secret cnsa-remote-mount-storage-cluster-1 -n ibm-spectrum-scale product=ibm-spectrum-scale
```

Optionally create a CA certificate ConfigMap:

```
kubectl create configmap cacert-storage-cluster-1 --from-literal=storage-cluster-1.crt="$(openssl s\_clien
```

Download the `RemoteCluster` resource:

```
curl -fs https://raw.githubusercontent.com/IBM/ibm-spectrum-scale-container-native/v6.0.0.x/generated/scal
```

Update the fields that are specific to your installation, such as cluster name, contact nodes, and GUI-related values. Finally, apply the `RemoteCluster` resource:

```
kubectl apply -f remotecluster.yaml
```

Verification:

```
kubectl get remotecluster -n ibm-spectrum-scale
```

The `RemoteCluster` should report `READY=True` and `AVAILABLE=True`.

Verifying IBM Storage Scale Container Native cluster deployment

Verify that IBM Storage Scale Container Native pods are running:

```
kubectl get pods -n ibm-spectrum-scale -o wide
```

Validate GPFS state within pods:

```
kubectl exec -it -n ibm-spectrum-scale c676f5u24-worker1 -- mmgetstate -a
kubectl exec -it -n ibm-spectrum-scale c676f5u24-worker1 -- mmlscluster
```

All nodes should report an active GPFS state.

Configuring remote filesystems

To mount filesystems from the external Storage Scale cluster, define `Filesystem` custom resources.

Example for remotefs:

```
apiVersion: scale.spectrum.ibm.com/v1beta1
kind: Filesystem
metadata:
    name: remotefs
    namespace: ibm-spectrum-scale
spec:
    remote:
        cluster: remotecluster-c676f5u24-scale1
        fs: remotefs
```

Example for directdiskfs:

```
apiVersion: scale.spectrum.ibm.com/v1beta1
kind: Filesystem
metadata:
    name: directdiskfs
    namespace: ibm-spectrum-scale
spec:
    remote:
        cluster: remotecluster-c676f5u24-scale1
        fs: directdiskfs
```

Apply:

```
kubectl apply -f filesystem.remotefs.yaml
kubectl apply -f filesystem.directdiskfs.yaml
```

Verification:

```
kubectl get filesystem -n ibm-spectrum-scale -o wide
```

Both filesystems should show `ESTABLISHED=True`.

Configuring local storage

You can optionally configure local storage by using `LocalDisk` and `Filesystem` resources.

Create a `LocalDisk`:

```
apiVersion: scale.spectrum.ibm.com/v1beta1
kind: LocalDisk
metadata:
    name: disk0
    namespace: ibm-spectrum-scale
spec:
    device: /dev/vdc
    node: c676f5u24-worker1
    nodeConnectionSelector:
        matchExpressions:
        - key: kubernetes.io/hostname
                  operator: In
                  values:
                  - c676f5u24-worker1
                  - c676f5u24-worker2
```

Apply:

```
kubectl apply -f localdisk.yaml
Create a local filesystem:
apiVersion: scale.spectrum.ibm.com/v1beta1
kind: Filesystem
metadata:
    name: localfs
    namespace: ibm-spectrum-scale
spec:
    local:
        blockSize: 4M
        pools:
        - name: system
            disks:
            - disk0
        replication: 1-way
        type: shared
```

Apply:

```
kubectl apply -f filesystem.local.yaml
```

Verification:

```
kubectl get localdisk -n ibm-spectrum-scale

kubectl get filesystem -n ibm-spectrum-scale -o wide
```

The local filesystem should report `ESTABLISHED=True` and `HEALTHY=Healthy`.

Summary

This section demonstrated the complete installation and configuration workflow for IBM Storage Scale Container Native on Kubernetes. The process includes installing prerequisites, deploying the operator, configuring access to container images, building kernel modules, defining the IBM Storage Scale Container Native cluster, integrating with an external Storage Scale cluster, and provisioning both remote and local filesystems.

Following these steps ensures a fully functional IBM Storage Scale Container Native deployment capable of providing persistent storage services through the Container Storage Interface.

Deploying IBM Storage Scale Container Native on Red Hat OpenShift

This section describes the end-to-end deployment of IBM Storage Scale Container Native on Red Hat OpenShift Container Platform (OCP). It highlights Red Hat OpenShift-specific architectural requirements, operational differences, and installation steps. Unlike vanilla Kubernetes, Red Hat OpenShift introduces additional platform abstractions such as MachineConfig, MachineConfigPools (MCPs), CRI-O, Red Hat CoreOS (RHCOS), and a managed global image pull secret. These components influence how kernel dependencies, container images, and privileged workloads are deployed. This section documents those differences explicitly while preserving a consistent IBM Storage Scale Container Native deployment model.

The deployment approach follows a hybrid architecture where Red Hat OpenShift and an external IBM Storage Scale storage cluster operate as independent systems. IBM Storage Scale Container Native functions as the integration layer, exposing Storage Scale file systems to container workloads through operator-managed GPFS services and CSI interfaces.

Red Hat OpenShift platform overview and design model

The Red Hat OpenShift cluster provides the container orchestration and security framework, while the IBM Storage Scale cluster supplies enterprise-grade distributed storage. IBM Storage Scale Container Native bridges these environments through operator-managed GPFS daemon pods, REST API communication with the Storage Scale GUI, and CSI-based provisioning workflows.

In Red Hat OpenShift, worker nodes are immutable and based on RHCOS. As a result, kernel headers and development packages required by GPFS cannot be installed manually. Instead, they must be provisioned using MachineConfig objects that extend the worker MachineConfigPool. This distinction is one of the most critical behavioral differences from Kubernetes and is addressed early in the installation flow.

Infrastructure and node layout

The validation environment uses a virtualized Red Hat OpenShift cluster deployed on KVM infrastructure. The cluster consists of three control-plane nodes and three worker nodes, all running Red Hat Enterprise Linux CoreOS (RHCOS 9.6). The Red Hat OpenShift control plane manages node configuration through MachineConfigPools:

```
# oc adm top nodes
NAME                                 CPU(cores)   CPU(%)   MEMORY(bytes)   MEMORY(%)
master0.ocpcnsa2026.cp.fyre.ibm.com   649m         8%       6531Mi          45%
master1.ocpcnsa2026.cp.fyre.ibm.com   925m         12%      8629Mi          59%
master2.ocpcnsa2026.cp.fyre.ibm.com   617m         8%       6693Mi          46%
worker0.ocpcnsa2026.cp.fyre.ibm.com   622m         8%       8625Mi          59%
worker1.ocpcnsa2026.cp.fyre.ibm.com   209m         2%       6591Mi          45%
worker2.ocpcnsa2026.cp.fyre.ibm.com   451m         6%       7852Mi          54%

# oc get nodes -o wide
NAME                                 STATUS   ROLES                  AGE    VERSION   INTERNAL-IP    EXTE
master0.ocpcnsa2026.cp.fyre.ibm.com   Ready    control-plane,master   106d   v1.33.6   10.22.6.56     <non
master1.ocpcnsa2026.cp.fyre.ibm.com   Ready    control-plane,master   106d   v1.33.6   10.22.6.206    <non
master2.ocpcnsa2026.cp.fyre.ibm.com   Ready    control-plane,master   106d   v1.33.6   10.22.10.220   <non
worker0.ocpcnsa2026.cp.fyre.ibm.com   Ready    worker                 106d   v1.33.6   10.22.11.47    <non
worker1.ocpcnsa2026.cp.fyre.ibm.com   Ready    worker                 106d   v1.33.6   10.22.12.219   <non
worker2.ocpcnsa2026.cp.fyre.ibm.com   Ready    worker                 106d   v1.33.6   10.22.15.57    <non
```

The IBM Storage Scale cluster remains external and unchanged, running IBM Storage Scale 6.0.0.x and exposing both GPFS (port `1191`) and GUI REST API (port `443`).

This separation enforces clear demarcation of responsibilities:

- Red Hat OpenShift controls container scheduling, security, and lifecycle.
- Storage Scale controls data layout, disks, replication, and quotas.

IBM Storage Scale remote cluster configuration

IBM Storage Scale Remote Cluster Configuration is the same as for vanilla Kubernetes. For detailed steps, see IBM Storage Scale Storage Cluster Configuration.

Platform baseline and validation

Platform baseline and validation for Red Hat OpenShift is the same as upstream (vanilla) Kubernetes. For detailed steps, see Kubernetes Platform Baseline and Validation.

Remote Storage Scale cluster baseline: External cluster

Remote Storage Scale Cluster baseline and validation for Red Hat OpenShift is the same as vanilla Kubernetes. For detailed steps, see Storage Scale Baseline (External Cluster).

After both the Red Hat OpenShift platform and the external Storage Scale cluster have been validated, proceed with the configuration and installation of IBM Storage Scale Container Native.

MachineConfig for kernel development packages

IBM Storage Scale Container Native requires kernel headers matching the running kernel to compile the GPFS kernel portability layer. In Red Hat OpenShift, this requirement is fulfilled by applying a custom MachineConfig to the worker pool.

The MachineConfig installs the following packages:

- `kernel-devel`
- `kernel-headers`

After applying the MachineConfig, each worker node undergoes a controlled reboot. Progress must be monitored through the MachineConfigPool status:

```
# curl -O  https://raw.githubusercontent.com/IBM/ibm-spectrum-scale-container-native/v6.0.0.x/generated/sc
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   272  100   272    0     0   2029      0 --:--:-- --:--:-- --:--:--  2029

# oc apply  -f mco.yaml
machineconfig.machineconfiguration.openshift.io/00-worker-ibm-spectrum-scale-kernel-devel created
```

Verification: Ensure `UPDATED=True` and `DEGRADED=False` for the worker pool:

```
# oc get mcp
NAME     CONFIG                                             UPDATED   UPDATING   DEGRADED   MACHINECOUNT
master   rendered-master-bcb0b79ab017be18a6277b784125d212   True      False      False      3
worker   rendered-worker-9272a2d8df3e9ff312fe4304ba39ffa7   True      False      False      3
```

This step has no Kubernetes equivalent and is mandatory for Red Hat OpenShift-based deployments.

Installing the IBM Storage Scale Container Native operator on Red Hat OpenShift

The IBM Storage Scale Container Native operator is installed using the Red Hat OpenShift-specific installation manifest. This process creates the required namespaces, CRDs, RBAC roles, webhook configurations, and the controller manager deployment. The following namespaces are created:

- `ibm-spectrum-scale`
- `ibm-spectrum-scale-operator`
- `ibm-spectrum-scale-csi`
- `ibm-spectrum-scale-dns`

Verify the following conditions:

- Operator pod is in Running state.
- Required CRDs are present.

```
# kubectl apply -f https://raw.githubusercontent.com/IBM/ibm-spectrum-scale-container-native/v6.0.0.x/gene

# oc apply -f install.yaml
namespace/ibm-spectrum-scale created
namespace/ibm-spectrum-scale-csi created
namespace/ibm-spectrum-scale-dns created
namespace/ibm-spectrum-scale-operator created
[..]
validatingwebhookconfiguration.admissionregistration.k8s.io/ibm-spectrum-scale-validating-webhook-configur

# kubectl get namespaces | grep ibm-spectrum-scale
ibm-spectrum-scale                                Active   15s
ibm-spectrum-scale-csi                            Active   15s
ibm-spectrum-scale-dns                            Active   15s
ibm-spectrum-scale-operator                       Active   15s
# kubectl get pods -n ibm-spectrum-scale-operator
NAME                                                   READY   STATUS    RESTARTS   AGE
ibm-spectrum-scale-controller-manager-7bb96ccf5b-lgct4   1/1     Running   0          27s
```

Container image access and entitlement configuration

IBM Storage Scale container images are hosted in IBM Cloud Container Registry (cp.icr.io). Red Hat OpenShift supports two pull-secret models:

1. Namespace-level pull secrets
2. Global pull secret managed by the cluster

In this deployment, namespace-level pull secrets are created for all IBM Storage Scale Container Native namespaces. Additionally, the Red Hat OpenShift global pull secret is updated to include the IBM entitlement credentials. Entitlement keys can be obtained by visiting the container library page here:

```
#  export ENTITLEMENT_KEY=<your_key>
```

Create pull secrets for each IBM Storage Scale Container Native namespace:

```
#  for namespace in ibm-spectrum-scale ibm-spectrum-scale-operator ibm-spectrum-scale-dns ibm-spectrum-sca
  kubectl create secret docker-registry ibm-entitlement-key -n ${namespace} \
    --docker-server=cp.icr.io \
    --docker-username=cp \
    --docker-password=${ENTITLEMENT_KEY}
done
secret/ibm-entitlement-key created
secret/ibm-entitlement-key created
secret/ibm-entitlement-key created
secret/ibm-entitlement-key created
```

Verification:

```
# for namespace in ibm-spectrum-scale ibm-spectrum-scale-operator ibm-spectrum-scale-dns ibm-spectrum-scal
NAME                  TYPE                                  DATA   AGE
ibm-entitlement-key   kubernetes.io/dockerconfigjson   1      32m
NAME                  TYPE                                  DATA   AGE
ibm-entitlement-key   kubernetes.io/dockerconfigjson   1      32m
NAME                  TYPE                                  DATA   AGE
ibm-entitlement-key   kubernetes.io/dockerconfigjson   1      32m
NAME                  TYPE                                  DATA   AGE
ibm-entitlement-key   kubernetes.io/dockerconfigjson   1      32m
# unset ENTITLEMENT_KEY
```

> **Note:** `ENTITLEMENT_KEY` is no longer needed once image pull secrets have been created successfully.

Creating the IBM Storage Scale Container Native cluster custom resource

The `Cluster` custom resource defines the IBM Storage Scale Container Native cluster running inside Red Hat OpenShift. After applying the cluster CR, GPFS daemon pods are scheduled on worker nodes, and the internal IBM Storage Scale Container Native cluster forms automatically.

For a Red Hat OpenShift cluster:

```
# curl -fs https://raw.githubusercontent.com/IBM/ibm-spectrum-scale-container-native/v6.0.0.x/generated/sc
```

The cluster.yaml file must be edited to change `license.accept` to `true`. By default, `license.license` is set to `data-access`. It must be modified if you need features available under the `data-management` license.

The updated section should look like the following example:

```
  license:
    accept: true
    license: data-management
```

In environments where DNS does not resolve external storage nodes, hostAliases can be defined:

```
spec:
  daemon:
      hostAliases:
      - hostname: cnsaredbook1.fyre.ibm.com
        ip: 10.21.69.95
      - hostname: cnsaredbook2.fyre.ibm.com
        ip: 10.21.71.176
```

The next step applies a label to the nodes where the daemon runs, which is typically the Scale worker nodes. If the cluster size is small and if resources allow, you can also schedule pods on master nodes:

```
# oc label nodes -l node-role.kubernetes.io/worker= scale.spectrum.ibm.com/daemon-selector=
```

Apply the cluster custom resource by running the following command:

```
# oc apply -f cluster.yaml
```

Integrating a remote IBM Storage Scale cluster

To enable remote file system access, IBM Storage Scale Container Native authenticates to the external Storage Scale cluster using a `ContainerOperator` GUI user.

Required components include GUI credential secrets, CA certificate ConfigMaps, and a `RemoteCluster` custom resource.

The test environment includes Storage cluster v6.0.0.0. Hence, `CsiAdmin` GUI user is not required. Create a secret for the `ContainerOperator` GUI user defined on the storage cluster:

```
# oc create secret generic cnsa-remote-mount-cnsaredbook-cluster --from-literal=username='cnsa_storage_gui
secret/cnsa-remote-mount-cnsaredbook-cluster created
```

To label the secret, run the following command:

```
# oc label secret cnsa-remote-mount-cnsaredbook-cluster -n ibm-spectrum-scale product=ibm-spectrum-scale
secret/cnsa-remote-mount-cnsaredbook-cluster labeled
```

Verification: Once applied, the `RemoteCluster` resource transitions to `READY=True` and `AVAILABLE=True`.

IBM Storage Scale container native uses Transport Layer Security (TLS) verification to guarantee secure HTTPS communication with the storage cluster GUI. It verifies the server's certificate chain and host name. There are three options for customers to configure the security protocol:

1. CA certificate ConfigMap
2. Red Hat default CA
3. Skip verification

This test environment uses a CA certificate ConfigMap. The details of these options can be found in Configuring Certificate Authority (CA) certificates:

```
# oc create configmap cacert-cnsaredbook-cluster --from-literal=storage-cluster-1.crt="$(openssl s_client

configmap/cacert-cnsaredbook-cluster created
```

In this example, `cnsaredbook1.fyre.ibm.com` is the GUI host on the storage Cluster. Download a copy of `remotecluster.yaml`:

```
# curl -fs https://raw.githubusercontent.com/IBM/ibm-spectrum-scale-container-native/v6.0.0.x/generated/sc
```

The sample `remotecluster.yaml` uses `remotecluster-sample` as the resource name. Make sure to edit this file to give a suitable name for the `RemoteCluster` resource. It is crucial when multiple remote clusters are being configured. Also, ensure all other fields that are specific to your installation have been updated, such as contact nodes and GUI related values.

Apply the `remotecluster.yaml`as shown in the example below:

```
# oc apply -f remotecluster.yaml
```

Verification: Once configured, remotecluster resource looks like the following example:

```
# oc describe remotecluster -n ibm-spectrum-scale
Name:         cnsaredbook-remotecluster
Namespace:    ibm-spectrum-scale
[..]
  Local Key SHA Digest:    d3bf183069770c0043a6fc4eb46a690c5c81a299434bf5c2414dee7ba16c3a89
  Remote Key SHA Digest:   8bfeff80dac695049ae3c120b114a7f6c2176f6049d245dd15372cdff85c9efd
```

Remote filesystem configuration and validation

Remote filesystems are mounted by defining `Filesystem` custom resources that reference the `RemoteCluster` object. Remote IBM Storage Scale filesystems must have quota enabled. If quota is not enabled, IBM Storage Scale Container Native reports that the remote filesystem does not have quota enabled and reconciliation fails as shown in the example output:

```
# Events:
  Type     Reason  Age                 From        Message
  ----     ------  ----                ----        -------
  Warning  Failed  12s (x13 over 35s)  Filesystem  Remote filesystem 'FS01' does not have quota enabled.
```

Download a copy of the sample for Red Hat OpenShift:

```
curl -fs https://raw.githubusercontent.com/IBM/ibm-spectrum-scale-container-native/v6.0.0.x/generated/scal
```

The updated copy of `filesystem.remote.yaml` will look like the following example:

```
metadata:
  labels:
    app.kubernetes.io/instance: ibm-spectrum-scale
    app.kubernetes.io/name: cluster
  name: remote-fs1
  namespace: ibm-spectrum-scale
spec:
  remote:
    cluster: cnsaredbook-remotecluster
    fs: FS01
```

Remote filesystems show `ESTABLISHED=True` and `HEALTHY=NotSupported`. This is expected behavior.

Validation and health checks

Key validation includes the following commands:

- `oc get nodes`
- `oc get clusteroperators`
- `oc get pods -n ibm-spectrum-scale`
- `oc describe remotecluster`
- `oc describe filesystem`

GPFS state can be validated by executing GPFS commands inside daemon pods.

Deploying IBM Storage Scale Container Native on Red Hat OpenShift with direct-attach disks

This section describes the steps to implement IBM Storage Scale Container Native with shared SAN storage between the CNSA cluster and a remote cluster. The procedure begins with an existing remote cluster that is defined within CNSA.

Shared disk configuration

A shared-disk configuration for CNSA, also referred to as a direct-attached setup, requires SAN volumes that are accessible to both the CNSA cluster and the remote IBM Storage Scale cluster. The same SAN protocol (for example, Fibre Channel or iSCSI) must be used by both clusters. To provision shared SAN storage, any SAN array or block storage supported by Red Hat Enterprise Linux CoreOS can be used.

For validation and deployment, IBM SAN Volume Controller (SVC) is used to present two 99GB disks over the Fibre Channel protocol to the external cluster and to all worker nodes in the Red Hat OpenShift cluster.

Preparing the worker nodes

Red Hat Enterprise Linux CoreOS is an immutable OS. IBM Storage Scale Container Native requires kernel headers matching the running kernel to compile the GPFS kernel portability layer. In addition, worker nodes must have access to the required devices and therefore require operating system-level preparation.

In Red Hat OpenShift, this requirement is fulfilled by applying a custom MachineConfig to the worker pool. MachineConfig objects define node-level changes within Kubernetes and ensure that the configuration is applied consistently not only to existing nodes, but also to newly added or restored nodes. This approach ensures that the configuration remains persistent and resilient across the cluster. In a Red Hat OpenShift cluster, multiple MachineConfig objects are applied through MachineConfig pools. Custom pools can be defined to target specific nodes. In this example, the default master and worker pools are used. Because the demonstration environment is a compact cluster, only the master pool contains nodes. In contrast, When using a general-purpose operating system, such as Ubuntu, these changes can be applied directly on the host.

The kernel-devel machineconfig definition that is a prerequisite for the installation of CNSA is shown in the following example:

```
# oc get mcp
NAME     CONFIG                                             UPDATED   UPDATING   DEGRADED   MACHINECOUNT
master   rendered-master-2f014bf9415381c78d460f947856cbd5   False     True       True       3

# oc get mc
NAME                                          GENERATEDBYCONTROLLER                      IGNITIONVERS
00-master                                     2a29ea49e2625aac5e66e9675de04c2677689c7c   3.5.0
00-master-ibm-spectrum-scale-kernel-devel                                                3.2.0
...
```

In addition to configuring systemd units and applying kernel arguments, it can be used to create and manage files, apply NetworkManager configurations, and introduce other system-level changes required to tailor node behavior. This capability enables consistent, repeatable configuration of operating system features across the cluster. When defining additional machineconfigs with file content the content is base64 encoded to ensure the literal strings are sent. In Red Hat OpenShift the butane tool is used for managing machineconfig file content in a human-readable way. A generic way of creating files on Kubernetes nodes is using a base64 encoded string. Create a `multipath.conf` file that is suitable for the attached storage and encode it. The IBM storage in the demonstration environment does not need any modifications so the following `multipath.conf` file suffices:

```
defaults {
user_friendly_names yes
find_multipaths yes
}

blacklist {
}
```

Create an encoded string using the base64 command:

```
# cat multipath.conf | base64 -w 0
```

Insert the string in the `spec.config.storage.files.content.source` field as shown below:

`source: data:text/plain;charset=utf-8;base64,BASE64_ENCODED_STRING_HERE`

Each application of a machineconfig will restart the affected nodes. Combining multiple definitions in a single file is preferred over separate machineconfig files.

Based on guidance provided by the storage vendor, it may be necessary to introduce additional definitions within the `multipath.conf` file to ensure proper device handling and path management. In certain scenarios, adjustments to block device attributes are also required, which can be achieved through the implementation of custom udev rules. These configurations should be encapsulated within the same MachineConfig resource, enabling a consistent and centralized approach to applying storage-related host customizations across all targeted nodes. This ensures that multipathing behavior and device attribute modifications are uniformly enforced, aligning with best practices for reliable and predictable storage operations in the cluster environment. Requirements for attaching systems to hosts running the Linux

If SAN volumes attached to the Kubernetes nodes have been provisioned from IBM FlashSystem®, a file called 99-flashsystem-rules can be created with the following content:

```
SUBSYSTEM=="block", ACTION=="add", ENV{ID_VENDOR}=="IBM",ENV{ID_MODEL}=="2145", RUN+="/bin/sh -c 'echo 120
```

The following example shows the base64 encoded string:

```
# cat 99-flashsystem-rules | base64 -w 0
U1VCU1lTVEVNPT0iYmxvY2siLCBBQ1RJT049PSJhZGQiLCBFTlZ7SURfVkVORE9SfT09IklCTSIsRU5We0lEX01PREVMfT09IjIxNDUiLC
```

The MachineConfig that includes the multipath configuration, udev rules, and the systemd configuration required to start the multipath daemon is shown in the following example:

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: master
  name: 99-worker-multipath-configuration
spec:
  config:
    ignition:
      version: 3.2.0
    storage:
      files:
      - contents:
          source: data:text/plain;charset=utf-8;base64,IyBkZXZpY2UtbWFwcGVyLW11bHRpcGF0aCBjb25maWd1cmF0aW9
        mode: 422
        overwrite: true
        path: /etc/multipath.conf
      - contents:
          source: data:text/plain;charset=utf-8;base64, U1VCU1lTVEVNPT0iYmxvY2siLCBBQ1RJT049PSJhZGQiLCBFTl
        overwrite: true
        path: /etc/udev/rules.d/99-flashsystem-rules
    systemd:
      units:
        - name: multipathd.service
          enabled: true
    osImageURL: ""
```

Apply this machineconfig and wait until all nodes in the machine config pool have been drained, rebooted, and made available again.

Next, verify that the disks are visible on the worker nodes. All available disks are automatically available inside the gpfs container of the Scale core component on each node:

```
# oc debug node/node1.inno2.amsiic.ibm.com -- chroot /host multipath -ll
Temporary namespace openshift-debug-9b74d is created for debugging node...
Starting pod/node1inno2amsiicibmcom-debug-9hxp2 ...
To use host binaries, run `chroot /host`. Instead, if you need to access host namespaces, run `nsenter -a
mpatha (360050768018e03885000000000000304) dm-3 IBM,2145
size=99G features='1 queue_if_no_path' hwhandler='1 alua' wp=rw
|-+- policy='service-time 0' prio=50 status=active
| `- 34:0:0:0 sdc 8:32  active ready running
`-+- policy='service-time 0' prio=10 status=enabled
  `- 33:0:0:0 sdb 8:16  active ready running
mpathb (360050768018e03885000000000000305) dm-2 IBM,2145
size=99G features='1 queue_if_no_path' hwhandler='1 alua' wp=rw
|-+- policy='service-time 0' prio=50 status=active
| `- 33:0:0:1 sde 8:64  active ready running
`-+- policy='service-time 0' prio=10 status=enabled
  `- 34:0:0:1 sdd 8:48  active ready running
```

Above output shows that two disk drives are visible on the worker node. The UUID, that is stated after the mpath device, is also visible in the SVC.

Validation: Validate the disks visible on the GPFS container of the core CNSA pod using the following command:

```
#  oc rsh -n ibm-spectrum-scale -c gpfs node1 lsblk
NAME        MAJ:MIN RM   SIZE RO TYPE  MOUNTPOINTS
loop0         7:0    0   9.8M  1 loop
...
sdb           8:16   0    99G  0 disk
`-mpatha    253:3    0    99G  0 mpath
sdc           8:32   0    99G  0 disk
`-mpatha    253:3    0    99G  0 mpath
sdd           8:48   0    99G  0 disk
`-mpathb    253:2    0    99G  0 mpath
sde           8:64   0    99G  0 disk
`-mpathb    253:2    0    99G  0 mpath
sdf           8:80   0    10G  0 disk
```

The multipath driver is not accessible from the GPFS container itself, but all block devices and device-mapper devices are visible and accessible.

Defining the filesystem on the external cluster

On the external cluster, create NSDs and a filesystem. In the example setup, two 99 GB disks are assigned to the external scale cluster, which consists of two nodes. Output of the multipath command on the first node:

```
# multipath -ll
...
mpathb (360050768018e03885000000000000304) dm-5 IBM,2145
size=99G features='1 queue_if_no_path' hwhandler='1 alua' wp=rw
|-+- policy='service-time 0' prio=50 status=active
| `- 33:0:0:0 sdd 8:48  active ready running
`-+- policy='service-time 0' prio=10 status=enabled
  `- 34:0:0:0 sdi 8:128 active ready running
mpathc (360050768018e03885000000000000305) dm-6 IBM,2145
size=99G features='1 queue_if_no_path' hwhandler='1 alua' wp=rw
|-+- policy='service-time 0' prio=50 status=active
| `- 34:0:0:1 sdj 8:144 active ready running
`-+- policy='service-time 0' prio=10 status=enabled
  `- 33:0:0:1 sde 8:64  active ready running
```

In IBM Storage Scale environments, Network Shared Disks (NSDs) are defined centrally but leveraged cluster-wide. NSD definitions are required on only a single cluster node; however, device naming conventions can vary across nodes due to differences in hardware enumeration or operating system behavior.

To ensure consistency, the device specified for each NSD must correspond to the correct block device as identified on the first node listed in the `servers` stanza. This node acts as the reference point for NSD creation. Subsequent nodes in the cluster automatically discover and associate the NSDs by using descriptor data derived from their locally attached devices, rather than relying on identical device names.

As a recommended best practice, all NSDs should initially be created with the first cluster node designated as the primary node (that is, the first entry in the `servers` variable). This approach simplifies initial deployment and ensures predictable ownership and control during configuration. If operational requirements change, the NSD stanza can later be updated to assign a different primary node for specific NSDs. Such updates can be applied dynamically by using the `mmchnsd` command, allowing for flexible reconfiguration without requiring full redeployment.

In this environment, the first node remains the primary node for all NSDs. This configuration is appropriate because all Red Hat OpenShift worker nodes have direct access to the storage, and I/O processing is not performed by the local cluster.

```
# cat localfs.stanza
%nsd: device=dm-5
           nsd=localnsd1
           servers=cluster1_node1,cluster1_node2
           usage=dataAndMetadata
           failureGroup=1
           pool=system
           thinDiskType=auto

%nsd: device=dm-6
           nsd=localnsd2
           servers=cluster1_node1,cluster1_node2
           usage=dataAndMetadata
           failureGroup=1
           pool=system
           thinDiskType=auto

# mmcrnsd -F localfs.stanza
```

Now create the filesystem and mount it:

```
# mmcrfs localfs -F localfs.stanza --perfileset-quota --filesetdf --auto-inode-limit -Q yes -m 1 -M 2 -r 1

The following disks of localfs will be formatted on node jump.inno2.amsiic.ibm.com:
    localnsd1: size 101376 MB
    localnsd2: size 101376 MB
Formatting file system ...
Disks up to size 830.99 GB can be added to storage pool system.
Creating Inode File
Creating Allocation Maps
Creating Log Files
Clearing Inode Allocation Map
Clearing Block Allocation Map
Formatting Allocation Map for storage pool system
Completed creation of file system /dev/localfs.
mmcrfs: mmsdrfs propagation completed.
# mmmount localfs -a
Mon Feb 23 01:57:36 PM CET 2026: mmmount: Mounting file systems ...
```

The filesystem is now available for use.

Defining the external filesystem in the CNSA cluster

You can define the remote filesystem in the Red Hat OpenShift Cluster. The following example YAML creates a `localfs` filesystem on the CNSA cluster with the same name as is used in the REMOTE_CLUSTER_NAME cluster:

```
apiVersion: scale.spectrum.ibm.com/v1beta1
kind: Filesystem
metadata:
  labels:
    app.kubernetes.io/instance: ibm-spectrum-scale
    app.kubernetes.io/name: cluster
  name: localfs
  namespace: ibm-spectrum-scale
spec:
  remote:
    cluster: REMOTE_CLUSTER_NAME
    fs: localfs
  seLinuxOptions: {}
```

For Red Hat OpenShift clusters, replace `seLinuxOptions` with the following content:

```
eLinuxOptions:
    level: s0
    role: object_r
    type: container_file_t
    user: system_u
```

Verification: Run the following command to confirm that the filesystem was created in the CNSA cluster:

```
# oc get filesystems -n ibm-spectrum-scale
NAME             ESTABLISHED   HEALTHY        AGE
localfs          True          NotSupported   48s


# oc rsh -n ibm-spectrum-scale -c gpfs NODE_NAME mmremotefs show localfs
Local Name  Remote Name  Cluster name       Mount Point        Mount Options    Automount  Drive  Priority
localfs     localfs      cluster1_node1.amsiic.ibm.com /mnt/localfs       rw,context="system_u:object_r:co
```

In IBM Storage Scale environments, each time a file system is mounted, a discovery process is initiated in which all locally attached disk devices are scanned for the unique identifier i.e NSD ID written to the disk at the time it was defined as an NSD. These NSD IDs are then correlated with the identifiers recorded in the file system metadata. When a match is established, I/O operations are performed directly against the local block device, thereby optimizing performance through local access paths.

In the event that direct access to the device is no longer available--for example, due to path failure or node-level disruption--I/O operations are transparently redirected over the network to the designated primary NSD server. This failover mechanism ensures continued data availability and maintains application continuity without requiring manual intervention.

To verify the location where I/O is being serviced for NSDs within a given file system, the `mmlsdisk` command can be used with the `-M` option. The following example illustrates how to determine whether I/O is being performed locally or remotely:

```
# oc rsh -n ibm-spectrum-scale -c gpfs node1 mmlsdisk localfs -M

Disk name     IO performed on node     Device             Availability
------------  -----------------------  -----------------  ------------
localnsd1     localhost                /dev/dm-3          up
localnsd2     localhost                /dev/dm-2          up
```

In this example, the `IO performed on node` column indicates that I/O operations are being serviced locally (`localhost`), confirming that the nodes have direct access to the underlying block devices. This validation step is essential for confirming optimal data path utilization and ensuring that the Storage Scale deployment is operating in accordance with design expectations.

Summary

This chapter presented comprehensive deployment workflows for IBM Storage Scale Container Native on Red Hat OpenShift Container Platform, covering both remote cluster integration and direct-attached storage configurations. While the foundational IBM Storage Scale Container Native architecture remains consistent with Kubernetes deployments, Red Hat OpenShift introduces additional considerations around kernel management, image authentication, node immutability, and storage device preparation through MachineConfig objects.

The chapter detailed the complete remote cluster deployment process, including MachineConfig implementation for kernel development packages, operator installation, container image access through IBM entitlement keys, cluster custom resource creation, and remote filesystem integration with quota requirements. Additionally, it covered direct-attached storage scenarios where shared SAN volumes are accessible to both the CNSA cluster and external Storage Scale cluster, including multipath configuration, udev rules, NSD definition, and I/O path verification.

By leveraging MachineConfig for node-level preparation, integrating with Red Hat OpenShift's image registry model, and following operator-driven workflows, administrators can deploy IBM Storage Scale Container Native reliably in secure, enterprise-grade Red Hat OpenShift environments. This chapter complements the Kubernetes deployment guide and provides a unified reference for multi-platform IBM Storage Scale Container Native installations across both remote and direct-attached storage architectures.

Troubleshooting and debugging IBM Storage Scale Container Native deployment

This section describes practical troubleshooting and debugging techniques used during the installation and configuration of IBM Storage Scale Container Native in a Kubernetes environment. Because IBM Storage Scale Container Native spans Kubernetes, the IBM Storage Scale Container Native operator, the IBM Storage Scale cluster, and the Container Storage Interface (CSI), effective debugging requires a layered and structured approach that correlates symptoms across these components.

Debugging strategy and workflow

Troubleshooting IBM Storage Scale Container Native deployments should follow a consistent validation sequence:

- Validate Kubernetes platform health (nodes, pods, networking, DNS)
- Validate IBM Storage Scale Container Native operator and custom resources (CRDs)
- Validate connectivity to the Storage Scale cluster (GUI and GPFS)
- Validate filesystem and storage resource state
- Correlate errors across logs from Kubernetes, operator, and Storage Scale

This layered model reflects the IBM Storage Scale Container Native architecture, where Kubernetes orchestrates, the operator reconciles desired state, and IBM Storage Scale executes storage operations.

Verifying pod health and status

Begin by confirming that all IBM Storage Scale Container Native-related pods are running and stable:

```
# kubectl get pods -n ibm-spectrum-scale -o wide
# kubectl get pods -n ibm-spectrum-scale-operator
```

Pods should be in a `Running` state with minimal restarts. Investigate any pods in `Pending`, `CrashLoopBackOff`, or `ImagePullBackOff`.

For deeper inspection:

```
# kubectl describe pod <pod-name> -n ibm-spectrum-scale
# kubectl logs <pod-name> -n ibm-spectrum-scale
```

For multi-container pods:

```
# kubectl logs <pod-name> -c <container-name> -n ibm-spectrum-scale
```

Debugging the IBM Storage Scale Container Native operator and custom resources

The IBM Storage Scale Container Native operator manages the lifecycle of cluster and storage resources. Failures during installation are often reflected in operator logs and resource status.

Check operator logs:

```
# kubectl logs -n ibm-spectrum-scale-operator deployment/ibm-spectrum-scale-controller-manager
```

Verify CRDs and resource state:

```
# kubectl get crds | grep spectrum
# kubectl get cluster
# kubectl describe cluster
# kubectl get remotecluster -n ibm-spectrum-scale
# kubectl describe remotecluster <name> -n ibm-spectrum-scale
```

Focus on status conditions and error messages, such as authentication failures or reconciliation errors.

Validating authentication and secrets

IBM Storage Scale Container Native uses Kubernetes secrets to authenticate to the IBM Storage Scale GUI REST API. Incorrect credentials or roles are a common source of failure:

```
# kubectl get secrets -n ibm-spectrum-scale
# kubectl describe secret cnsa-remote-mount-storage-cluster-1 -n ibm-spectrum-scale
```

Validate access manually:

```
# curl --insecure -u 'cnsa_storage_gui_user:cnsa_storage_gui_password' https://<scale-node>/scalemgmt/v2/c
```

A `401 Unauthorized` response indicates invalid credentials or insufficient permissions.

Debugging filesystem resources

Filesystem custom resources reflect the integration state between IBM Storage Scale Container Native and the Storage Scale cluster:

```
# kubectl get filesystem -n ibm-spectrum-scale -o wide
# kubectl describe filesystem <fs-name> -n ibm-spectrum-scale
```

The following key observations apply:

- `ESTABLISHED=False` indicates connectivity or mount issues.
- `HEALTHY=NotSupported` is expected for remote filesystems.
- Status conditions provide detailed diagnostic messages.

Debugging LocalDisk and storage provisioning

Local storage provisioning requires correct node mapping and device availability:

```
# kubectl get localdisk -n ibm-spectrum-scale
# kubectl describe localdisk <disk-name> -n ibm-spectrum-scale
```

The following common issues can occur:

- Node not recognized as a daemon node
- Incorrect node naming (must match Kubernetes node name)
- Device path not present or accessible

Verify node names:

```
# kubectl get nodes
```

Ensure that IBM Storage Scale Container Native daemon pods are running on the target nodes and that the specified device exists.

Validating network connectivity

IBM Storage Scale Container Native requires connectivity to both GPFS (port `1191`) and the GUI REST API (port `443`).

From nodes:

```
# nc -vz <scale-node> 1191
# nc -vz <scale-node> 443
```

From within a pod:

```
# kubectl run nettest --image=busybox:1.36 --restart=Never -- sleep 3600
# kubectl exec nettest -- nc -vz <scale-node> 1191
# kubectl exec nettest -- nc -vz <scale-node> 443
# kubectl delete pod nettest
```

Failures indicate network, routing, firewall, or policy issues.

Resolving DNS and hostname issues

Hostname resolution failures can prevent IBM Storage Scale Container Native from communicating with the Storage Scale cluster:

```
# kubectl run dns-test --image=busybox:1.36 --rm -it -- nslookup <scale-node>
```

If DNS resolution fails, configure `hostAliases` in the Cluster resource:

```
spec:
    daemon:
        hostAliases:
        - hostname: <scale-node>
        ip: <ip-address>
```

Debugging the Storage Scale cluster

If Kubernetes and operator checks succeed, several commands can help you investigate the Storage Scale cluster directly:

```
# mmgetstate -a
# mmhealth cluster show
# mmlsmount all -L
# mmlsmgr
```

Review logs:

```
# tail -f /var/adm/ras/mmfs.log.latest
```

Focus on the following items:

- Authentication attempts from IBM Storage Scale Container Native
- GPFS daemon communication issues
- Filesystem and NSD status

Common issues and observations

Several recurring patterns are observed during IBM Storage Scale Container Native deployments:

- Remote filesystems report `HEALTHY=NotSupported`, which is expected.
- LocalDisk failures are often caused by incorrect node naming or missing daemon nodes.
- GUI authentication issues result in HTTP 401 errors.
- DNS or hostname resolution issues prevent remote cluster registration.
- Operator logs provide the most actionable diagnostic information.

Summary

Troubleshooting IBM Storage Scale Container Native deployments requires coordinated validation across Kubernetes, the IBM Storage Scale Container Native operator, and the IBM Storage Scale cluster. By following a structured approach that progresses from platform validation to storage-layer analysis, issues can be isolated efficiently and resolved with minimal disruption.

This methodology improves time to resolution and ensures a reliable deployment of IBM Storage Scale Container Native in Kubernetes environments.

IBM Storage Scale Container Storage Interface (CSI) on Red Hat OpenShift

This section describes the design, configuration, and operational behavior of the IBM Storage Scale Container Storage Interface (CSI) driver when deployed on Red Hat OpenShift implicitly with CNSA. It uses verified command output from a functional environment to illustrate both lightweight (directory-based) and fileset-based dynamic provisioning models.

Introduction to IBM Storage Scale CSI

IBM Storage Scale Container Storage Interface (CSI) provides a standardized mechanism for storage vendors to expose persistent storage services to containerized workloads. IBM Storage Scale CSI enables containers running on

Red Hat OpenShift to consume enterprise-grade IBM Storage Scale with advanced capabilities such as ReadWriteMany (RWX) access, high availability, and centralized management.

The CSI driver abstracts Storage Scale constructs such as file systems, filesets, and directories, and maps them to Kubernetes resources including StorageClasses, PersistentVolumeClaims (PVCs), and PersistentVolumes (PVs).

CSI operator deployment and verification

The IBM Storage Scale CSI Operator manages installation and lifecycle of the CSI driver components. A healthy operator is required for dynamic provisioning.

Verification command:

```
# oc get csiscaleoperator -n ibm-spectrum-scale-csi
NAME                     VERSION   SUCCESS
ibm-spectrum-scale-csi   3.0.1     True
```

The sample environment reports IBM Storage Scale CSI version 3.0.1 with status `SUCCESS=True`, confirming that the CSI services are operational.

StorageClass design models

IBM Spectrum Scale CSI exposes backend provisioning behavior through Kubernetes StorageClasses. Each StorageClass determines how Storage Scale objects are created and managed. This table shows the Storage classes used in test environment and their typical uses.

StorageClass	Provisioning Model	Backend Object	Typical Use Case
ibm-spectrum-scale-csi-lt	Lightweight (Directory)	Directory	Rapid provisioning, shared trees
ibm-spectrum-scale-csi-fileset	Fileset-based	IBM Storage Scale filesets	Isolation, quotas, governance

Lightweight (directory-based) volumes

Lightweight provisioning maps a PVC to a directory created under a predefined base path within the Spectrum Scale file system. This model reduces GPFS metadata overhead.

Creating storage class

The lightweight (LT) `StorageClass` provisions a directory under a shared parent path. This is suitable for workloads that require fast provisioning and minimal GPFS object creation overhead. Below is the YAML file used to create a storage class for Lightweight (Directory-Based) volumes (PVs):

```
# cat storageClass_Lightweight.yaml

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
    name: ibm-spectrum-scale-csi-lt
provisioner: spectrumscale.csi.ibm.com
parameters:
    volBackendFs: "remote-fs1"
    volDirBasePath: "share/" # relative path from filesystem mount point for
```

Command: `# oc create -f storageClass_Lightweight.yaml`

Creating PVC

The following code is an example of a typical YAML file for a lightweight dynamic PVC:

```
#  cat lt-pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: scale-lt-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: ibm-spectrum-scale-csi-lt
```

Command: `# oc create -f lt-pvc.yaml`

Verification:

```
# oc get sc
NAME                            PROVISIONER                    RECLAIMPOLICY   VOLUMEBINDINGMODE      ALL
ibm-spectrum-scale-csi-lt       spectrumscale.csi.ibm.com      Delete          Immediate              fal
# oc get pvc
NAME                                    STATUS   VOLUME                                     CAPA
scale-lt-pvc                            Bound    pvc-9f31e7dc-7a87-4592-badb-4bedcd96a926   1Gi

# oc get pv/pvc-9f31e7dc-7a87-4592-badb-4bedcd96a926
NAME                                     CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
pvc-9f31e7dc-7a87-4592-badb-4bedcd96a926   1Gi        RWX            Delete           Bound    ibm-spectru
```

Provisioning flow:

```
PVC (scale-lt-pvc) → StorageClass (ibm-spectrum-scale-csi-lt) → PersistentVolume → /gpfs/fs01/share/<pvc-u
```

The PVC transitions to the `Bound` state after the CSI provisioner successfully creates the backend directory.

Fileset-based volumes

Fileset-based provisioning creates a dedicated GPFS fileset per PersistentVolumeClaim, providing isolation, independent inode space, and compatibility with Spectrum Scale quotas.

Creating storage class

The fileset-based StorageClass dynamically creates a dedicated Spectrum Scale fileset for each PVC. This approach provides strong isolation and quota management at the fileset level. You can use the following code example in the YAML file to create a storage class for Fileset-Based volumes (PVs):

```
# cat storageClass_fileset.yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
    name: ibm-spectrum-scale-csi-fileset
provisioner: spectrumscale.csi.ibm.com
parameters:
    volBackendFs: "remote-fs1"
reclaimPolicy: Delete
```

Command: `# oc create -f storageClass_fileset.yaml`

Creating PVC

The following code sample is a typical YAML file for a fileset-based dynamic PVC:

```
# cat fileset-pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: scale-fileset-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: ibm-spectrum-scale-csi-fileset
```

Command: `# oc create -f fileset-pvc.yaml`

Verification:

```
# oc get sc/ibm-spectrum-scale-csi-fileset
NAME                            PROVISIONER                RECLAIMPOLICY   VOLUMEBINDINGMODE   ALLOWVOLU
ibm-spectrum-scale-csi-fileset   spectrumscale.csi.ibm.com   Delete          Immediate           false
#oc get pvc/scale-fileset-pvc
NAME                STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGEC
scale-fileset-pvc   Bound    pvc-9eb9fb4d-53a0-4721-aa91-f1da6f04049b   1Gi        RWX            ibm-spec

# oc get pv/pvc-9eb9fb4d-53a0-4721-aa91-f1da6f04049b
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
pvc-9eb9fb4d-53a0-4721-aa91-f1da6f04049b   1Gi        RWX            Delete           Bound    ibm-spectru
```

```
Provisioning flow: PVC (scale-fileset-pvc) → StorageClass (ibm-spectrum-scale-csi-fileset) → PersistentVol
```

The `scale-fileset-pvc` demonstrates fileset-based provisioning. Upon PVC creation, the CSI driver creates a dedicated fileset inside the backend Storage Scale file system.

PVC-PV-Spectrum Scale mapping

The CSI driver maintains a strict mapping between Kubernetes objects and backend Spectrum Scale resources, encoded within the CSI `volumeHandle`.

Kubernetes Object	Example Name	Storage Scale Object	Backend Location
PVC	scale-lt-pvc	Directory	pvc-9f31e7dc-7a87-4592-badb-4bedcd96a926
PVC	scale-fileset-pvc	Fileset	pvc-9eb9fb4d-53a0-4721-aa91-f1da6f04049b

Backend validation

Storage Scale CLI and Openshift CLI commands can be used to validate backend object creation. The `mmlsfileset` command confirms CSI-created filesets, and directory-based PVCs are verified using standard filesystem tools.

Summary

IBM Spectrum Scale CSI tightly integrates Kubernetes persistent storage with Spectrum Scale enterprise capabilities. Lightweight provisioning offers speed and simplicity, while fileset-based provisioning offers enhanced isolation and governance, enabling flexible deployment architectures.

Failure scenarios and troubleshooting

This section describes common failure scenarios encountered when using IBM Storage Scale Container Storage Interface (CSI) on Red Hat OpenShift. Each scenario outlines observable symptoms, potential root causes, and recommended troubleshooting steps.

PersistentVolumeClaim remains in pending state

A PersistentVolumeClaim (PVC) that remains in the `Pending` state indicates that Kubernetes is unable to dynamically provision a corresponding PersistentVolume.

The following symptoms may be observed:

- PVC remains in `Pending` state
- Events indicate `ExternalProvisioning` or waiting for volume creation

Use the following troubleshooting steps when necessary:

- Verify CSI operator and provisioner pods are running
- Validate StorageClass parameters such as `volBackendFs` and `volDirBasePath`
- Inspect PVC events and CSI controller logs
- Confirm Storage Scale filesystem availability

PVC bound but pod fails to mount volume

In this scenario, the PVC and PV are bound successfully, but application pods fail to mount the volume. You might see the following symptoms:

- Pod stuck in `ContainerCreating` state
- `MountVolume.SetUp failed` errors in pod events

Use the following troubleshooting steps:

- Verify Storage Scale is mounted on all worker nodes
- Ensure `Storage-scale-csi-node` pods are running
- Review kubelet and CSI node logs
- Validate POSIX permissions and UID/GID consistency

Backend fileset or directory not created

Provisioning may appear successful in Kubernetes but fail to create the expected backend object.

The following symptoms may be observed:

- PVC is `Bound` but backend directory or fileset is missing
- `mmlsfileset` output does not list expected fileset

Use the following troubleshooting steps:

- Validate `volBackendFs` configuration

- Review CSI controller logs
- Check Storage Scale inode and metadata availability

Volume deletion or reclaim failures

When `reclaimPolicy` is set to `Delete`, CSI must remove backend resources during PVC deletion.

The following symptoms may be observed:

- PV stuck in `Terminating` state
- Backend directory or fileset remains after PVC deletion

Use the following troubleshooting steps:

- Inspect PV finalizers
- Review CSI deletion logs
- Manually clean up backend objects if required

Performance or RWX access issues

Shared RWX volumes may experience performance or permission challenges if application access patterns are not aligned with Storage Scale semantics.

The following symptoms may be observed:

- Intermittent I/O errors
- Permission denied errors across pods

Use the following troubleshooting steps:

- Validate file ownership and permissions
- Monitor Storage Scale performance metrics
- Use fileset-based provisioning for higher isolation

Summary

Effective troubleshooting of IBM Storage Scale Container Native deployments requires a systematic approach that recognizes the architectural separation between Kubernetes management operations and Storage Scale data plane execution. The debugging methodology progresses through distinct validation layers, beginning with Kubernetes platform health verification, advancing through operator and custom resource state analysis, and culminating in Storage Scale cluster-side diagnostics. This structured progression enables efficient problem isolation by establishing clear boundaries between control plane orchestration failures and storage layer execution issues.

Authentication and network connectivity represent the most common sources of deployment failure. GUI REST API authentication issues manifest as HTTP 401 errors and prevent operator reconciliation, while network connectivity problems between Kubernetes nodes and Storage Scale services disrupt both management operations and data access paths. DNS resolution failures can prevent remote cluster registration entirely, requiring explicit hostname configuration through hostAliases when standard resolution mechanisms are insufficient. Operator logs provide the most actionable diagnostic information during troubleshooting, as they capture reconciliation attempts, API interactions, and error conditions that may not be visible through resource status alone.

The IBM Storage Scale Container Storage Interface driver introduces additional complexity through its integration with Kubernetes persistent storage abstractions. Dynamic provisioning failures typically originate from StorageClass misconfiguration, CSI operator unavailability, or backend filesystem accessibility issues. PersistentVolumeClaim binding failures require validation of both Kubernetes-side resources and Storage Scale filesystem state, as successful binding depends on coordination across both layers. Mount failures often indicate node-level Storage

Scale daemon issues or POSIX permission misalignments that prevent container runtimes from accessing provisioned volumes.

Common failure patterns observed across IBM Storage Scale Container Native deployments include remote filesystems reporting `HEALTHY` as `NotSupported`, which is expected behavior rather than an error condition. LocalDisk provisioning failures frequently result from incorrect node naming that fails to match Kubernetes node identifiers or from daemon pods not being scheduled on target nodes. Operator reconciliation stalls often indicate finalizer conflicts, insufficient RBAC permissions, or resource dependencies that cannot be satisfied. Understanding these patterns accelerates diagnosis and reduces time to resolution by focusing investigation on known failure modes rather than exhaustive system analysis.

Troubleshooting effectiveness depends on maintaining clear operational boundaries between platform validation, operator behavior analysis, and storage layer diagnostics. By following a methodical validation sequence that progresses from Kubernetes control plane health through operator reconciliation state to Storage Scale cluster operations, issues can be isolated to specific architectural layers and resolved with minimal disruption. This approach ensures that diagnostic efforts remain focused and that remediation actions target the actual source of failure rather than symptoms manifested in dependent components.

Application use cases

This section explores practical application use cases for IBM Storage Scale Container Native and provides guidance on selecting appropriate storage configurations for containerized workloads. It begins by examining persistent volume options, including the three types of filesystem access (remote via network, remote with direct disk access, and local filesystem) and the various persistent volume types (lightweight, dependent fileset, independent fileset, and consistency groups). It then presents detailed use cases for enterprise applications, including IBM FileNet® Content Manager and PostgreSQL databases, demonstrating how IBM Storage Scale Container Native can be configured to meet specific application requirements for performance, high availability, disaster recovery, and data management.

This section emphasizes the trade-offs between different storage approaches and provides recommendations for selecting the optimal configuration based on features needed, scalability requirements, and operational considerations.

Persistent Volume options

Where the IBM Storage Scale Container Native cluster provides access to IBM Storage Scale filesystems, the CSI driver defines the Storage Class, Persistent Volumes, and Persistent Volume Claims. Depending on the use case, a different choice can be made for both the cluster type or the Persistent Volume type.

IBM Storage Scale cluster types

As discussed in more detail in the section "Architecture of Red Hat OpenShift and Kubernetes in an IBM Storage Scale Container Native Deployment", there are three types of filesystem access in an IBM Storage Scale Container Native cluster:

1. Remote filesystem with NSD access via TCP/IP network
2. Remote filesystem with NSD access via shared accessible block devices
3. Local filesystem with NSD access via local block devices

Option one allows for a central storage cluster that can provide multiple IBM Storage Scale Container Native clusters with their own filesystem. Option one has the following distinguishing characteristics:

- A separate team can manage the storage, apart from the Kubernetes management team that just provisions storage.

- It allows for the use of Storage Scale System appliances, reducing the knowledge needed in managing IBM Storage Scale.
- It only requires Ethernet connectivity for the IBM Storage Scale Container Native cluster. A dedicated network on the Kubernetes worker nodes may be necessary to increase performance.

Option two adds disk access to the Kubernetes worker nodes so the remote cluster is only used for coordination and management, and not for I/O. The I/O is performed directly on the disks that are shared with the remote cluster. Option two has the following additional characteristics:

- Better performance by providing a direct path to disk. Additional Ethernet or FC adapters may be required in the Kubernetes worker nodes.
- Instead of using a Storage Scale System appliance, regular block storage such as IBM FlashSystem® FC disks can be assigned as shared storage to both the CNSA cluster and the remote Storage Scale cluster.
  - Performance benefits are that the I/O path for the worker node is now FC based which unloads the Red Hat OpenShift Ethernet.
  - Less load on the Kubernetes worker nodes as management, maintenance, and AFM activity is handled by the remote Storage Scale cluster.
  - Possible synergy with IBM CSI block driver to deliver block PVCs to Red Hat OpenShift: See IBM Block Storage CSI driver documentation

Option three eliminates the need for a remote cluster by defining a local file system in the IBM Storage Scale Container Native cluster itself. This does require shared storage assigned through a network to Kubernetes worker nodes, because local (internal) storage is not supported. This solution resembles option two with several distinctions:

- Storage Scale cluster and filesystem fully managed and updated through Kubernetes.
- The Storage Scale file systems are only available inside the Kubernetes cluster.
- Similar setup to the IBM Fusion Access for SAN product, but not limited to Red Hat OpenShift, and missing IBM Fusion DR options.

Persistent Volume types

Within a Storage Scale filesystem files can be stored in three different ways:

1. In a simple directory like any filesystem, called lightweight, quick to deploy.
2. In a dependent fileset. This is a separate grouping of files within a filesystem that can be created, mounted, unmounted and deleted as a whole. It uses inodes for its files from the filesystem or its parent independent fileset.
3. In an independent fileset. This is like a dependent fileset, but with its own administration of inodes, which makes it independent from the main filesystem. It uses the same storage as the main filesystem like the dependent fileset, but its own set of inodes.

Each of these options has a corresponding PVC type in the Scale CSI driver.

Special mention is the Consistency Group, which is a construct within the Scale CSI driver, not the file system itself. It is an independent fileset that is created per namespace with several dependent filesets used as Persistent Volumes defined therein to provide a single point of control for creating snapshots and clones. By performing the snapshot action against the base independent fileset, all dependent filesets inside it are automatically included as well.

When defining Persistent Volumes the options above can be provisioned in a dynamic way or in a static way. With the dynamic way the filesets, names and locations in the file system are generated by the CSI driver. This is the recommended option.

When defining Persistent Volumes in a static way the filesets need to be defined manually using Storage Scale commands before creating the Persistent Volumes. This allows for more control on the filesets created but means

some features are not supported by the CSI driver, such as cloning, expansion, and snapshotting. Static provisioning is therefore only recommended for very specific use cases like accessing existing data and building Disaster Recovery set ups.

This table shows an overview of the features of the dynamically provisioned Persistent Volumes:

PV Type	Limit (per file system)	Snapshot	Cloning	Compression	Quota	Tiering
Consistency Group	3000+10000	yes	yes	yes	yes*	yes
independent fileset	3000	yes	yes	yes	yes	yes
dependent fileset	10000	no	no	yes	no*	yes
lightweight	unlimited	no	no	no	no	no

> * The fileset quota is set to be the requested size of the Persistent Volume. However, a dependent fileset cannot have an enforced quota like the independent fileset does. The parent fileset is therefore the real limit for the size of a dependent fileset.

The type of Persistent Volume that is chosen depends on the features needed and the number of expected PVs. The independent fileset limitation of 3000 per filesystem might be an issue in some use cases.

The recommendation is to use independent fileset based persistent volumes unless the Consistency Group feature of a consistent snapshot across multiple PVs is needed.

IBM FileNet Content Manager

This section describes how IBM Storage Scale Container Native can be used for IBM FileNet Content Management. FNCM provides repository services to capture, manage, and store digital assets. Multiple repositories (object stores) can be created to satisfy business requirements. Each store has its own database and storage areas. FNCM can run directly on a server platform or in containers on a CNCF certified Kubernetes platform.

FileNet storage requirements

Each FileNet store requires a database and capacity to store content. The database storage area can also store content in the format of Binary Large Objects (BLObs). For large environments separate storage areas are recommended.

Content can be stored in either of the following storage areas:

- File storage areas
  - Can be NTFS, UNIX based filesystem, or a GPFS filesystem.
- Fixed storage areas
  - Uses a file storage area for database and staging.
  - Uses an external content device such as an S3 object store, Storage Protect, etc for permanent storage. GPFS is not supported.

- Advanced storage areas
  - Uses replication to provide High Availability and Disaster Recovery for database, file, and fixed storage areas.

A FNCM storage setup can be as simple as just a database storage area to multiple File and Fixed storage area for a store. Having many storage areas does complicate the storage strategy from a FileNet perspective, as knowledge about the virtual or physical storage is then needed to make informed decisions on which area to use and create policies to place and migrate data from one area to another.

IBM Storage Scale can provide a single file storage area where the placement of data to hot or warm tiers can be guided using policies on the Scale cluster instead of FileNet. Offloading of cold files to an external object store using `cacheVolumes` is possible, negating the need for a fixed storage area.

IBM Storage Scale cluster

When deploying FileNet Content Manager in a Red Hat OpenShift cluster there are two options to access IBM Storage Scale Container Native:

1. Local cluster (or Fusion Access for SAN)
   - Ideal for single, dedicated Red Hat OpenShift deployments
   - Multiple repositories supported
2. Remote cluster
   - Provides central storage for multiple FNCM instances
   - Easy scalability as Storage Scale filesystem is outside of Red Hat OpenShift
   - Can use appliances like SSS 6000 with heat map based storage migration

When using IBM Storage Scale Container Native, cross site High Availability and Disaster Recovery are not provided by the Red Hat OpenShift cluster itself. Stretching a cluster across sites still leaves the Red Hat OpenShift cluster as a Single Point Of Failure (SPOF). If a maintenance action such as a Red Hat OpenShift upgrade fails, it cannot be reverted, which leaves both sites inoperable. Having the application provide the redundancy using data replication between two Red Hat OpenShift clusters each with its own instance of FileNet is a more solid solution. Using Postgres database replication and Advanced Storage Areas provides an active-active solution without SPOFs. For more information, see Replication models for advanced storage areas

Increasing the pagepool (core pod memory size) may improve performance of the filesystems as more files will be cached. The default cache size is 4 GB. More cache is better, though exactly how much depends on the number of files that were read again from cache, not the size of the cache.

Cache volume instead of fixed storage area

When using Kubernetes storage in a file storage area a special kind of Persistent Volume can be interesting to use, which is a cache volume. This is a fileset with a limited amount of disk (NVME/HDD) based capacity, but a transparent offload to an external S3 object store. It resembles the Fixed Content storage area in that respect, which has a staging area and several external storage options.

The advantage of using a cache volume is that the compatibility of the external object store is done by IBM Storage Scale and not by FileNet. This places the responsibility for the S3 storage connections with the storage administrator, not the FileNet administrator.

A limitation is that FileNet can no longer take advantage of some of the features external fixed content devices can provide, such as object locking and scrubbing.

Persistent Volume type options

Both static and dynamic volumes are supported by FileNet.

When using dynamic volumes, three Scale based storage classes need to be specified in FileNet. A slow, medium, and fast tier. These can for instance be a cache volume based tier, an HDD based tier, and an NVME based tier.

Static volumes allow for more flexibility in defining the fileset, but require predefining of filesets in the Scale filesystem and defining Persistent Volumes and Persistent Volume Claims. Using a static fileset allows for instance highly targeted migration policies in Storage Scale to move data to its optimal location, which can even be on a tape device.

As many PVCs need to be created the recommendation is to use a dedicated GPFS filesystem for all PVCs, and use dynamic provisioning to create them.

For online backups, a consistent snapshot of all PVCs needs to be taken. IBM Storage Scale Container Native provides this using the Consistency Group Storage Class. A snapshot of any PVC in the class means the snapshot is for all PVCs in that class. All PVCs in a consistency group are in the same filesystem tier.

If online backups are not a requirement, independent fileset based storage classes are recommended for maximum flexibility.

PostgreSQL database

This section describes how IBM Storage Scale Container Native can be used for PostgreSQL (also called Postgres), though the recommendations are applicable to other databases as well.

PostgreSQL deployments can range from small local databases using the open source distribution to large multi-cluster setups with support from EDB.

With PostgreSQL, perform replication by using the built-in Write-Ahead Log (WAL) shipping mechanism. This allows for synchronous or asynchronous replication to multiple target clusters.

Postgres storage requirements

Each PostgreSQL instance needs a Persistent Volume Claim for PGDATA, with optionally a separate PVC for the WAL device. More PVCs can be defined for tablespaces. Using tablespace PVCs allows differentiation of performance classes or tiers in the database, and the creation of a "temporary" tablespace.

Full online backups of PostgreSQL are performed through PVC snapshots. If additional tablespaces are created all need to be snapshotted at the same time. Point in Time recovery is then achieved using the WAL archives.

For high-performance Postgres databases direct access to disk is recommended using either local drives or FC attached drives. Storage provided as FC storage devices is preferred as this includes write cache and RAID protected storage.

IBM Storage Scale cluster

Each replica of a PostgreSQL instance should be located on a different Storage Scale filesystem, preferably also on a different Storage Scale cluster. This is to create isolation between the replicas of the database. The use of a local cluster or a remote cluster with shared FC access is the recommended solution to achieve optimal performance.

Within a local cluster multiple filesystems can be created to have a separation of the replicas within a Red Hat OpenShift cluster. When replicating to another Red Hat OpenShift cluster a local cluster with storage from a different SAN storage server is recommended to achieve redundancy not only at the Red Hat OpenShift level, but also at the storage level.

Depending on the performance requirements, a remote Scale cluster with network access can be sufficient, also taking the network topology and speed into account.

Persistent Volume type options

If a single PGDATA PVC is used by PostgreSQL for all storage needs, then an independent fileset type PVC is recommended.

With multiple PVCs for the database, if a hot backup is needed then a consistency group Storage Class is required so the snapshot is made for all PVCs at the same time. When the backup is taken cold (usually done from a replica) independent fileset type PVCs are recommended for maximum flexibility.

IBM DB2 Data Warehouse

This section describes how IBM Storage Scale Container Native can be used for IBM DB2® Data Warehouse. IBM provides two options for deployment:

1. Containerized deployment on Red Hat OpenShift or Kubernetes
2. P10 Private Cloud Rack for DB2 Data Warehouse

The DB2 Data Warehouse containerized deployment is suitable for small to enterprise size container environments. It can be deployed as a single container or as a multi container managed by the DB2 operator.

The Private Cloud Rack is custom designed high-performance infrastructure and software offering that delivers a validated DB2 solution based on Red Hat OpenShift. It contains IBM POWER based servers and IBM FlashSystem storage which can be scaled from 3 worker nodes to 77 nodes, and from 74 TB of net storage to 2587 TB. The storage is assigned to DB2 pods using IBM Storage Scale Container Native connecting to an independent IBM Storage Scale cluster with all IBM FlashSystem volumes assigned to all worker nodes using Fibre Channel SAN.

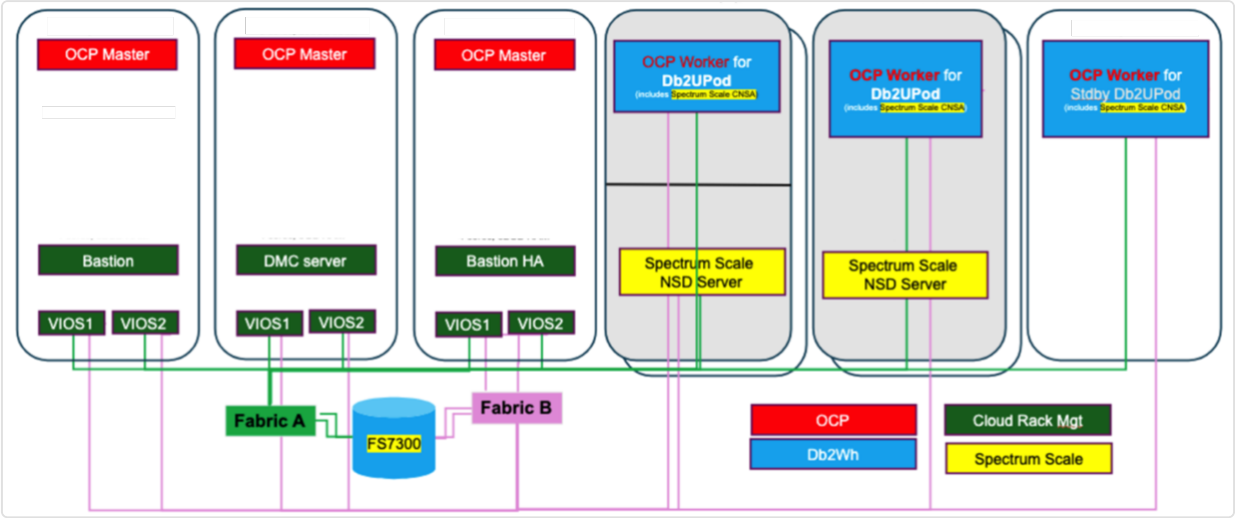

Figure: DB2 Private Cloud Rack Architecture

DB2 storage requirements

Both DB2 solutions require storage for the following uses:

- System & Backup storage [Shared with RWX]
  - Db2 or Db2 Warehouse instance home directory
  - Diagnostic logs
  - Other global configuration directories
  - Backups, Restore or Load locations
- User storage [Exclusive with RWO]
  - Database storage paths
  - Transaction logs

The database storage needs a high performance, scalable solution. The IBM Storage Scale solution provides this by using (virtual) block access and multi-server model, as opposed to solution such as NFS which use file access and single server.

The P10 Cloudrack solution uses IBM Storage Scale Container Native with all IBM FlashSystem disks assigned to all worker nodes. This is the optimal setup for performance as the DB2 pods can directly access the IBM FlashSystem FC block devices on their local worker node directly. This minimizes the I/O path which reduces the latency of operations.

The backup storage is defined as a Persistent Volume as well, and this can be an NFS export when restore times are not critical. For large deployments or the P10 Cloud Rack an external Scale cluster based on IBM Storage Scale System 6000 is recommended. This allows for the most cost effective and best performing solution when deployed in a hybrid configuration with internal NVMe drives and NLSAS HDD drives in expansion drawers. The maximum SSS 6000 configuration can have 911 TB of NVMe Flash Storage and 15497 TB of NLSAS storage, with a total read throughput of 175 GB/s and a write throughput of 144 GB/s. Because a Scale System server simply provides a remote Storage Scale filesystem to the CNSA cluster over the network, the creation of backup PVCs is a standard activity for the Kubernetes/Red Hat OpenShift Administrator.

IBM Storage Scale cluster choices

The DB2 solution creates a database over multiple worker nodes in a single Kubernetes cluster. High Availability across sites is achieved using Q Replication to another DB2 instance in a near synchronous active-active configuration. A stretched Kubernetes cluster or special Disaster Recovery setup is therefore not needed.

Storage access for the DB2 database is recommended to use Fibre Channel storage on the worker nodes, preferably using the local cluster setup in CNSA. An external cluster such as a Scale System 6000 or custom Scale cluster through Ethernet is also possible.

For larger databases the core pod of CNSA should be configured with more memory to create a larger file cache (pagepool) size. Instead of the default 4 GB a size of 8 GB is used in the DB2 Cloudrack solution.

Persistent Volume type options

As the number of PVCs for DB2 is not large, the default recommendation of using independent filesets is applicable to DB2. The dynamic allocation method is the preferred way to create the PVCs, as the special advantages of static PVCs, like name control and DR options are not applicable for DB2.

Virtual machine storage

This section describes how IBM Storage Scale Container Native can be used for virtual machine storage. Kubernetes Virtualization uses Kubevirt which is built on the KVM hypervisor. To provide storage to VMs, the following options are the most common:

- Internal block storage: LVM, Rook/ODF/FDF block
- External connected block storage: FC, iSCSI, Ceph block
- Filesystem storage: NFS, IBM Storage Scale Container Native

Using the internal drives of a worker node using Logical Volumes is only really feasible for test environments, as it does not support high availability. Opensource Rook, Red Hat OpenShift Data Foundation, and IBM's Fusion Data Foundation create a highly available block storage solution that adds snapshots and Disaster Recovery capabilities. However, these rely on internal drives in the Kubernetes nodes, and a fast network to replicate the data across the nodes. Hosting the storage inside the Kubernetes cluster causes a considerable load on the nodes.

Mimicking the physical setup of traditional servers, block devices provided by FC or iSCSI can be used as storage for the VMs. The redundancy for the block devices is handled by the storage server, so the load on the Kubernetes nodes is minimal. iSCSI uses Ethernet to connect to a storage server, whereas a separate FC network provides proper redundancy and reliable performance. A drawback of external block storage is the sheer number of Linux

devices that are present on the worker nodes. This can hit limits in the storage backend. External Ceph is also connected through network to the worker nodes and offloads most of the processing to the Ceph cluster. As external Ceph is not the dedicated, curated, and managed solution as an internal Ceph Rook/ODF/FDF cluster is, this does add complexity to the overall solution.

The block storage for the VMs can also be presented in the form of image files, and this is what Storage Scale can do. A Scale storage cluster can either be deployed on external nodes or an appliance like a Scale Server 6000, or on the Kubernetes nodes themselves as a local cluster or Fusion Access for SAN cluster. Having a separation of the size of the storage solution to the size of the Kubernetes cluster is especially important for Virtual Machine storage, where ratio of CPU and Memory to Storage can be very dependent on the situation.

Virtual machine recommended storage requirements

The PVCs that are used for Virtual Machines must have the following characteristics:

1. RWX: This allows a volume to move from one Kubernetes node to another to support Live Migration.
2. Snapshots: This allows online backups to be made.
3. Quota: This needs to be set to limit the use of storage by a Virtual Machine.
4. Volume expansion: Increasing the size of a disk when needed.
5. Cloning: Capable of duplicating.

These requirements can be fulfilled with dynamically provisioned PVCs based on independent filesets.

IBM Storage Scale cluster

For high performance, the use of Direct FC connections is recommended. This allows direct access by the Linux I/O stack of the devices which results in the lowest possible latency.

Remote clusters can be used if the network is fast enough and PVCs are mostly needed for capacity or streaming I/O performance. Using a mix of Scale clusters on different types of storage is possible, and distinguishing between boot drives and data drives this can provide optimization not achievable with other storage provisioners.

Summary

IBM Storage Scale Container Native provides flexible storage provisioning options that enable organizations to align storage architecture with specific application requirements and operational constraints. The choice between remote filesystem access, direct disk attachment, and local filesystem deployment determines the balance between centralized management, performance characteristics, and operational independence. Remote filesystems with network-based NSD access enable centralized storage administration and support for multiple Kubernetes clusters, while direct disk attachment through shared block devices delivers superior performance by eliminating network latency from the data path. Local filesystem deployments provide complete autonomy within the Kubernetes cluster but require shared storage infrastructure and place full Storage Scale management responsibility within the container platform.

Persistent volume provisioning strategies must account for both feature requirements and scalability constraints. Independent filesets provide comprehensive functionality including snapshots, cloning, compression, and quota enforcement, but are limited to 3000 instances per filesystem. Dependent filesets extend capacity to 10000 instances per filesystem but sacrifice individual quota enforcement and snapshot capabilities. Lightweight directory-based volumes offer unlimited scalability without filesystem object overhead but lack advanced data management features. Consistency groups address the requirement for coordinated snapshots across multiple volumes by organizing dependent filesets within a parent independent fileset, enabling atomic backup operations for multi-volume applications while maintaining the dependent fileset scalability advantage.

Enterprise content management systems such as IBM FileNet Content Manager benefit from Storage Scale's unified namespace and policy-driven data placement capabilities, which simplify storage area management compared to traditional multi-tier architectures. Cache volumes provide transparent offloading to external object storage, effectively

replacing fixed content storage areas while consolidating S3 integration responsibility with storage administrators rather than application teams. Database workloads including PostgreSQL and DB2 Data Warehouse require careful consideration of replication topology, backup consistency, and I/O path optimization. High-performance database deployments favor direct Fibre Channel attachment to minimize latency, while replica isolation across separate filesystems or clusters ensures fault independence and supports active-active configurations without introducing single points of failure.

Virtual machine storage presents distinct requirements that emphasize ReadWriteMany access for live migration support, snapshot capabilities for operational backups, and quota enforcement to prevent resource exhaustion. Independent fileset-based persistent volumes satisfy these requirements while providing volume expansion and cloning capabilities essential for VM lifecycle management. The separation of storage cluster sizing from Kubernetes cluster capacity becomes particularly important in virtualization scenarios, where compute-to-storage ratios vary significantly based on workload characteristics and consolidation density.

Storage architecture decisions must balance performance requirements, operational complexity, and feature availability across the deployment lifecycle. Direct disk attachment configurations deliver optimal performance but increase infrastructure complexity and reduce flexibility in storage resource allocation. Remote cluster architectures simplify storage administration and enable resource sharing across multiple Kubernetes environments but introduce network dependencies that affect both performance and failure domain boundaries. Understanding these trade-offs and aligning storage provisioning strategies with application-specific requirements ensures that IBM Storage Scale Container Native deployments deliver both immediate operational value and long-term architectural sustainability.

Accelerating GenAI with IBM Storage Scale Container Native

This section demonstrates how IBM Storage Scale Container Native can accelerate generative AI workloads by providing high-performance shared storage for large language model (LLM) inference systems. It focuses on integrating IBM Storage Scale Container Native with vLLM Production Stack and LMCache, an open-source key-value cache engine that extends standard GPU-based caching to multiple tiers including remote storage. By leveraging Storage Scale's distributed file system capabilities, multiple vLLM instances can share KV cache across GPUs and nodes, reducing redundant computation and improving inference performance. This section presents a proof-of-concept architecture where IBM Storage Scale Container Native serves as the backend storage layer for KV cache persistence, enabling cache reuse across requests and providing fault tolerance. It covers the environment setup, including Storage Scale cluster configuration, Kubernetes deployment with IBM Storage Scale Container Native operator, and the integration of vLLM Production Stack components to demonstrate practical performance benefits for AI inference workloads.

Introduction to GenAI and LLMs

With the wide adoption of LLM (large language model), many software vendors are embedding GenAI in their applications. Enhancing the customer experience via an intelligent chatbot, summarizing reports, translating languages, assisting developers to code and solving mathematical problems by asking questions. LLM comes in various sizes and is trained in different domains and formats. Building an infrastructure to host LLM requires GPU and depending on the size of the LLM, this can be expensive. To make matters worse, one LLM might not be enough for some use case. One LLM Agent can only handle a specific type of complex unscripted task. Recent developments include creating a group of LLM Agents collaborating as a team to accomplish an even more complex task such as creating an interactive learning dashboard for chemistry reaction with a certain compound, build this and test it.

vLLM is one of the popular open-source serving engine for LLM. This is a fast throughput LLM inference and serving engine developed at UC Berkeley’s Sky Computing Lab. It supports better memory efficiency with PagedAttention, loading quantified models, integration with Hugging Face models and provides an OpenAI-compatible server interface for model serving.

LMCache is an open-source KV(Key-Value) cache engine for LLM inference. Standard KV cache only works within a single request on a single GPU. When the request ends, the cache is discarded. LMCache extends this by storing the cache in multiple tiers: GPU memory, CPU memory and storage. This allows cache sharing across multiple GPUs and nodes; reusing them for any future request that contains matching text segments. The size of GPU memory is usually limited, and it's reserved for loading the model. Tiering to CPU memory solves this problem and allows sharing of cache between GPUs in the same node. Tiering to a remote storage allows cache sharing across GPUs and nodes.

vLLM Production Stack is an open-source project to allow users to easily deploy a full stack inference system making use of vLLM with LMCache to speed up inference, prefix-aware routing, observability features for monitoring and autoscaling in Kubernetes.

IBM Storage Scale is a high-performance distributed storage system that can be used as a remote shared storage for LMCache. In this section, a proof of concept is built to demonstrate how to make use of IBM Storage Scale Container Native with vLLM Production Stack to accelerate the performance of LLM, enabling more KV cache hits and provides fault tolerance to the architecture.

Architecture overview

The reference architecture demonstrates how IBM Storage Scale Container Native integrates with vLLM Production Stack to provide shared, persistent KV cache storage across multiple inference instances. The deployment separates storage management from compute orchestration, with an external Storage Scale cluster delivering high-performance distributed filesystem capabilities to a Kubernetes environment running containerized vLLM workloads.

This is a high level architecture overview of the setup.

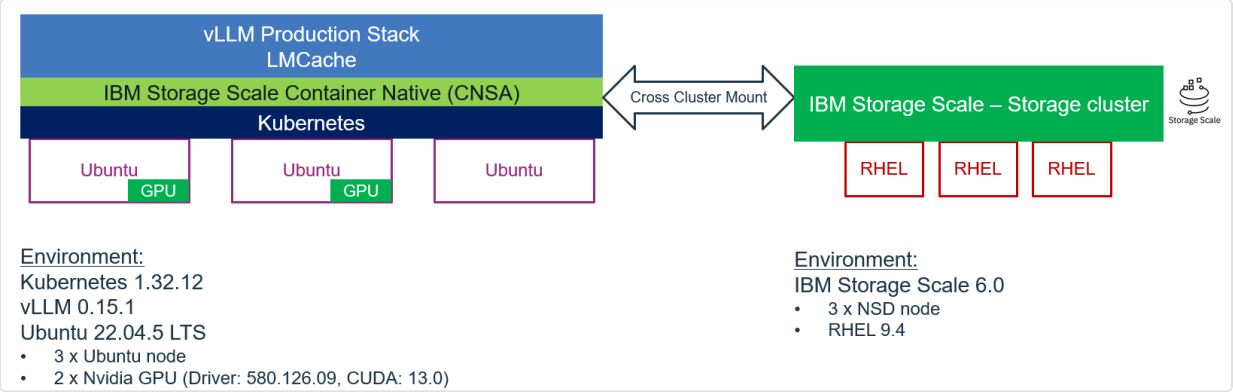

Figure: vLLM Production Stack Architecture with IBM Storage Scale Container Native

A three-node IBM Storage Scale cluster running on Red Hat Enterprise Linux is deployed to deliver a high-performance, POSIX-compliant distributed file system with parallel I/O capabilities.

A separate three-node Kubernetes cluster is provisioned on Ubuntu, configured with CRI-O as the runtime and Calico for the CNI network overlay. The IBM Storage Scale Container Native operator is installed within the Kubernetes environment to automate the mounting, management, and lifecycle operations of the Storage Scale file systems across worker nodes. IBM Storage Scale Container Native uses Storage Scale CSI capabilities to dynamically present the file system to multiple containers via a shared mount point.

The vLLM Production Stack is configured to use IBM Storage Scale Container Native as the backend storage layer for KV-cache persistence, allowing the system to leverage the high-throughput, low-latency characteristics of Storage Scale. The KV cache directory is mounted via CSI using a shared, RWX (ReadWriteMany) access mode to support multi-pod access.

Two vLLM instances are deployed as Kubernetes containers, each configured to access the same IBM Storage Scale Container Native-backed file system path. This enables shared KV cache usage between replicas, reducing redundant computation, improving model response performance, and supporting more scalable inference workloads.

The shared mount ensures that both vLLM instances can read and write cache entries concurrently, leveraging Storage Scale's parallel data access to maintain performance and consistency across nodes.

Environment

The proof-of-concept environment consists of two independent clusters: a three-node IBM Storage Scale cluster providing distributed filesystem services and a three-node Kubernetes cluster hosting the containerized inference workloads. Each cluster is configured with the necessary operators and components to support the integration, including IBM Storage Scale Container Native operator, GPU support, and vLLM Production Stack components.

The two clusters are set up by following the installation steps below:

Setting up a Storage Scale cluster

To view the steps to install IBM Storage Scale using the installation toolkit, see Installing IBM Storage Scale on Linux nodes with the installation toolkit

Setting up a Kubernetes cluster

To view the steps to install Kubernetes, see Getting started

Setting up a Node Feature Discovery

To view the steps to install a Node Feature Discovery operator, see Node Feature Discovery

Setting up an Nvidia GPU operator

To view the steps to install an Nvidia GPU operator, see Installing the IBM Storage Scale container native operator and cluster

Setting up an IBM Storage Scale Container Native

To view the steps to install IBM Storage Scale Container Native, see

Installing the IBM Storage Scale container native operator and cluster

Defining all Kubernetes nodes as control-plane and worker

Run the following 'kubectl' commands after logging into the Kubernetes cluster.

Define the nodes as control-plane and worker:

```
kubectl label node u20-1.testlab.net node-role.kubernetes.io/control-plane=""
kubectl label node u20-1.testlab.net node-role.kubernetes.io/worker=""

kubectl label node u20-2.testlab.net node-role.kubernetes.io/control-plane=""
kubectl label node u20-2.testlab.net node-role.kubernetes.io/worker=""

kubectl label node u20-3.testlab.net node-role.kubernetes.io/control-plane=""
kubectl label node u20-3.testlab.net node-role.kubernetes.io/worker=""
```

Verify that control-plane taints are removed, so they can run workloads:

```
kubectl taint nodes --all node-role.kubernetes.io/control-plane- || true
kubectl taint nodes --all node-role.kubernetes.io/master- || true

kubectl get nodes
NAME                STATUS   ROLES                  AGE     VERSION
u20-1.testlab.net   Ready    control-plane,worker   7d10h   v1.32.12
u20-2.testlab.net   Ready    control-plane,worker   7d10h   v1.32.12
u20-3.testlab.net   Ready    control-plane,worker   7d10h   v1.32.12
```

Preparation

Before deploying vLLM instances, the environment requires persistent storage for both the large language model files and the KV cache directory. This preparation phase creates the necessary persistent volume claims using IBM Storage Scale Container Native CSI driver and populates the model storage with the selected LLM from Hugging Face.

Creating a PVC to store LLM

The following code sample creates a PVC to store the LLM:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: model-storage-pvc4
    labels:
    app: model-downloader
spec:
    accessModes:
        - ReadWriteMany
    resources:
        requests:
            storage: 50Gi
```

Creating a Kubernetes job to download LLM into the PVC

Create a Hugging Face token secret:

```
kubectl create secret generic hf-token-secret --from-literal=token=<YOUR HUGGING FACE TOKEN>
```

For more information about Hugging Face access tokens, see Hugging Face User Access Tokens.

Create a Kubernetes job with the following commands:

```
apiVersion: batch/v1
kind: Job
metadata:
    name: model-downloader
    labels:
        app: model-downloader
    spec:
        ttlSecondsAfterFinished: 300
        template:
            metadata:
                labels:
                    app: model-downloader
spec:
    restartPolicy: OnFailure
    containers:
        - name: downloader
        image: python:3.11-slim
        command:
            - sh
            - -c
            - |
            set -e
            echo "Installing huggingface\_hub..."
            pip install -q huggingface\_hub hf\_transfer
            echo "Starting model download..."
            python3 -c "
            import os
            from huggingface\_hub import snapshot\_download

            model\_id = '<LLM MODEL>'
            local\_dir = '/data/hub/models--<LLM MODEL>'
```

Determine if it is already downloaded to avoid re-downloading on job retry:

```
marker = local\_dir + '/.download\_complete'
if os.path.exists(marker):
    print(f'Model already downloaded at {local\_dir}, skipping.')
else:
    print(f'Downloading {model\_id} to {local\_dir}...')
    snapshot\_download(
        repo\_id=model\_id,
        local\_dir=local\_dir,
        local\_dir\_use\_symlinks=False,
        ignore\_patterns=['*.msgpack', '*.h5'],
    )
    open(marker, 'w').write('done')
    print('Download complete.')
"
env:
    - name: HUGGING\_FACE\_HUB\_TOKEN
        valueFrom:
            secretKeyRef:
                name: hf-token-secret
                key: token
    - name: HF\_HUB\_ENABLE\_HF\_TRANSFER
        value: "1"        # faster downloads via hf\_transfer
    - name: HF\_HUB\_DISABLE\_PROGRESS\_BARS
        value: "1"
 volumeMounts:
    - name: model-storage
    mountPath: /data
resources:
    requests:
        cpu: "2"
        memory: "8Gi"
    limits:
        cpu: "4"
        memory: "16Gi"
    volumes:
        - name: model-storage
        persistentVolumeClaim:
            claimName: model-storage-pvc4
```

Creating a PVC to store KV cache

The following code sample creates a PVC to store the KV cache:

```
# nano lmcache-pvc.yaml:

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: lmcache-pvc
  namespace: default
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: ibm-spectrum-scale
  resources:
    requests:
      storage: 100Gi

kubectl apply -f lmcache-pvc.yaml
```

Starting a vLLM instance

The following code sample starts a vLLM instance:

```
# nano lmcache-config.yaml

apiVersion: v1
kind: ConfigMap
metadata:
  name: lmcache-config
  namespace: default
data:
  lmcache_config.yaml: |
    chunk_size: 256
    local_cpu:
      enabled: false
      max_cache_size: 0
    local_disk: "file:///lmcache-disk"
    max_local_disk_size: 80.0     # GiB — leave headroom below the 100Gi PVC
    save_unfull_chunk: true    # cache trailing partial chunks

# kubectl apply -f lmcache-config.yaml

# nano vllm.yaml

servingEngineSpec:
  runtimeClassName: ""
  modelSpec:
  - name: "mistral"
    repository: "lmcache/vllm-openai"
    tag: "v0.3.15"
    modelURL: "<LLM MODEL>"
    replicaCount: 1
    requestCPU: 10
    requestMemory: "40Gi"
    requestGPU: 1
    pvcName: "model-storage-pvc4"
    vllmConfig:
      enablePrefixCaching: false
      maxModelLen: 16384
      quantization: "awq"
    lmcacheConfig:
      enabled: true
      cpuOffloadingBufferSize: "0"
      diskOffloadingBufferSize: "80"

    env:
      - name: LD_LIBRARY_PATH
        value: "/lib/x86_64-linux-gnu:/usr/local/cuda/lib64"
      - name: LMCACHE_CONFIG_FILE
        value: "/etc/lmcache/lmcache_config.yaml"
      # MUST be an env var — cannot live inside the config file
      - name: LMCACHE_USE_EXPERIMENTAL
        value: "True"
      - name: LMCACHE_LOG_LEVEL
        value: "DEBUG"
    extraEnv:
      - name: HF_HUB_OFFLINE
        value: "1"
      - name: HF_HUB_DISABLE_SYMLINKS_WARNING
        value: "1"
      - name: HUGGING_FACE_HUB_TOKEN
        valueFrom:
          secretKeyRef:
            name: hf-token-secret
            key: token
    extraVolumes:
```

```
      - name: lmcache-shared
        persistentVolumeClaim:
          claimName: lmcache-pvc
      - name: lmcache-config
        configMap:
          name: lmcache-config
          items:
            - key: lmcache_config.yaml
              path: lmcache_config.yaml
    extraVolumeMounts:
      - name: lmcache-shared
        mountPath: /lmcache-disk       # must match local_disk path in ConfigMap
      - name: lmcache-config
        mountPath: /etc/lmcache

cacheserverSpec:
  replicaCount: 1
  containerPort: 8080
  servicePort: 81
  serde: "naive"
  repository: "lmcache/vllm-openai"
  tag: "v0.3.15"
  env:
    - name: LD_LIBRARY_PATH
      value: "/lib/x86_64-linux-gnu:/usr/local/cuda/lib64"
  resources:
    requests:
      cpu: "4"
      memory: "8G"
    limits:
      cpu: "4"
      memory: "10G"
  labels:
    environment: "cacheserver"
    release: "cacheserver"

routerSpec:
  replicaCount: 0

# helm repo add vllm https://vllm-project.github.io/production-stack
# helm install vllm vllm/vllm-stack -f vllm.yaml
```

This Helm chart provisions a single vLLM instance within a containerized environment.

Check the pod status:

```
kubectl get pods
```

Ensure the vLLM container has fully initialized before submitting any prompts.

Demonstration with 1 vLLM instance

A test script is prepared to transmit a context of approximately 8,000 tokens along with a 20-token prompt to the LLM. It runs one cold phase followed by three warm phases. During Phase 2, the 8,000-token pre-fill is bypassed because of KV Cache served from Storage Scale PVC, and only the 20-token query is processed. The time to first token (TTFT) is recorded, and the corresponding speedup is calculated:

```
git clone [https://github.com/siaut/vllm-demo.git](https://github.com/siaut/vllm-demo.git)

cd vllm-demo

python3 test\_lmcache.py --base-url http://<EXTERNAL IP OF vLLM>/v1 --model <LLM MODEL>

================================================================

LMCache TTFT Demonstration
URL            : http://<EXTERNAL IP OF vLLM>/v1
Model          : <LLM MODEL>
Shared context : ~2393 tokens  (9 cache chunks of 256)
Unique question: ~20 tokens  (changes per user)
max\_tokens     : 1  (TTFT-only measurement)
Warm passes    : 3

PHASE 1 — COLD: all 1 prompts, no cache anywhere

PHASE 2-4 — WARM: same prompts, LMCache disk serves the hits

================================================================

────────────────────────────────────────────────────────────

PHASE 1 — COLD  (GPU cache disabled, disk cache empty)

────────────────────────────────────────────────────────────

#  User      Status       TTFT  Question (truncated)

---  --------  -------  --------  ----------------------------------

1  user1     OK         1387ms  What is the formula for scaled dot-product at...

────────────────────────────────────────────────────────────

PHASE 2 — WARM  (LMCache disk hits expected)

────────────────────────────────────────────────────────────

#  User      Status       TTFT  Question (truncated)

---  --------  -------  --------  ----------------------------------

1  user1     OK          129ms  What is the formula for scaled dot-product at...

────────────────────────────────────────────────────────────

PHASE 3 — WARM  (LMCache disk hits expected)

────────────────────────────────────────────────────────────

#  User      Status       TTFT  Question (truncated)

---  --------  -------  --------  ----------------------------------

1  user1     OK          125ms  What is the formula for scaled dot-product at...

────────────────────────────────────────────────────────────

PHASE 4 — WARM  (LMCache disk hits expected)

────────────────────────────────────────────────────────────
```

```
#  User      Status       TTFT  Question (truncated)

---  --------  -------  --------  ---------------------------------

1  user1     OK          133ms  What is the formula for scaled dot-product at...

================================================================

LMCACHE PERFORMANCE RESULTS  (15:51:53)

================================================================

Shared context  : ~2393 tokens (cached by LMCache on disk)
Question length : ~20 tokens (unique per request, not cached)
max\_tokens      : 1 (TTFT measurement mode)
Requests        : 1 prompts × 2 phases = 2 total
Cold successes  : 1 / 1
Warm successes  : 1 / 1

Metric                          COLD (no cache)  WARM (disk hit)
──────────────────────────  ─────────────────  ────────────────

Min TTFT                                1387ms            129ms

Median TTFT                             1387ms            129ms

p95 TTFT                                1387ms            129ms

Max TTFT                                1387ms            129ms

SPEEDUP  ████████████████████████  10.8X

√  10.8X TTFT improvement — LMCache 10X target achieved!

Per-request breakdown:

#  User       Cold TTFT   Warm TTFT   Speedup

---  --------  ----------  ----------  --------

1  user1          1387ms        129ms     10.8X

================================================================
```

The initial cold run recorded a TTFT of 1387 ms. Subsequent warm runs achieved a TTFT of 129 ms, representing a 10.8x improvement in speed.

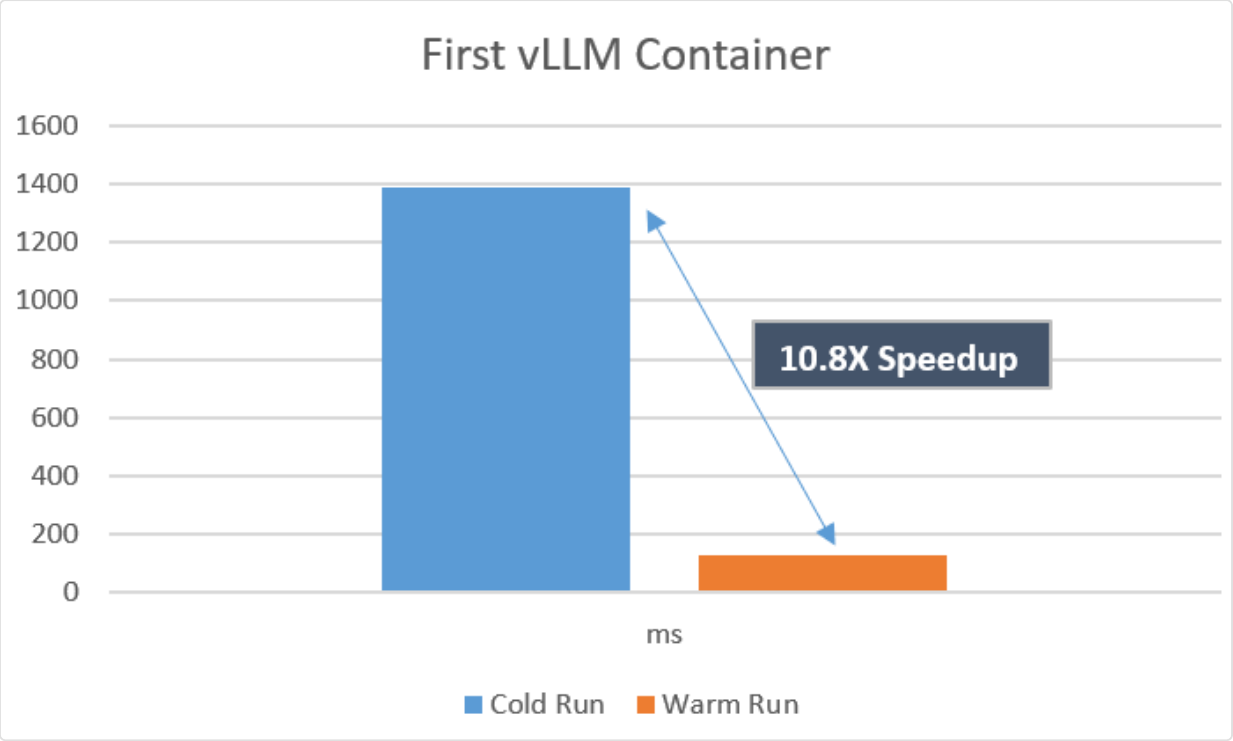

Figure: LMCache performance results showing 10.8x TTFT improvement

Restart the vLLM container and run the `test_lmcache.py` script again:

```
# python3 test_lmcache.py --base-url http://<EXTERNAL IP OF vLLM>/v1 --model <LLM MODEL>

================================================================
  LMCACHE PERFORMANCE RESULTS  (15:56:25)
================================================================
  Shared context  : ~2393 tokens (cached by LMCache on disk)
  Question length : ~20 tokens (unique per request, not cached)
  max_tokens      : 1 (TTFT measurement mode)
  Requests        : 1 prompts × 2 phases = 2 total
  Cold successes  : 1 / 1
  Warm successes  : 1 / 1

  Metric                          COLD (no cache)  WARM (disk hit)
  ──────────────────────────────  ─────────────────  ─────────────────
  Min TTFT                                 447ms             92ms
  Median TTFT                              447ms             92ms
  p95 TTFT                                 447ms             92ms
  Max TTFT                                 447ms             92ms

  SPEEDUP  ██████████  4.8X

  ✗  4.8X — LMCache disk cache may not be hitting.
     Check: kubectl logs <vllm-pod> | grep 'External prefix cache hit rate'
     Check: kubectl exec -it <vllm-pod> -- du -sh /lmcache-disk

  Per-request breakdown:
    #  User       Cold TTFT   Warm TTFT   Speedup
  ---  --------  ----------  ----------  --------
    1  user1           447ms         92ms      4.8X

================================================================
```

Note that the first (cold) run took 447 ms TTFT. The subsequent (warm) run took only 92 ms TTFT. This is a 4.8 times TTFT improvement. This demonstrates that when the container is restarted, the KV cache stored in the IBM Storage Scale PVC is persistent. This helps improve TTFT even when the container restarts.

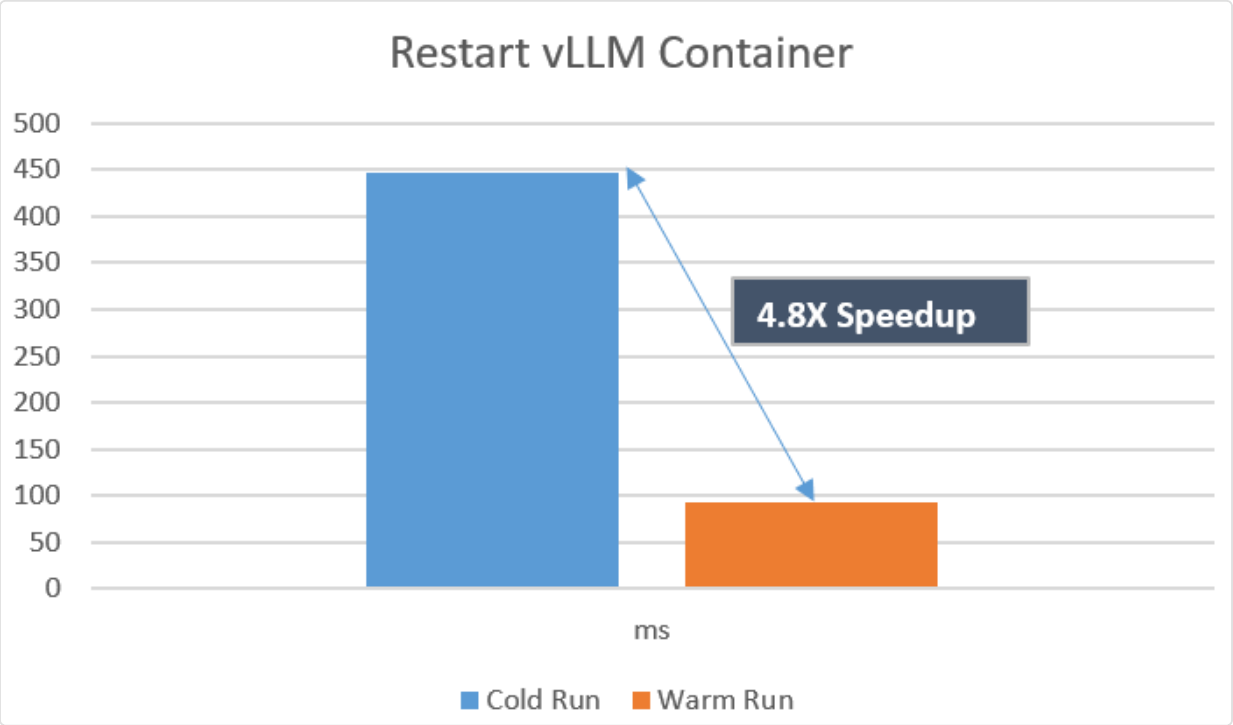

Figure: LMCache performance after container restart showing persistent cache

Demonstration with 2 vLLM instances

Edit the file `vllm.yaml` to increase the vLLM replica count to 2:

```
# helm upgrade vllm vllm/vllm-stack -f vllm.yaml
```

Run the `test_lmcache.py` script again:

```
# python3 test_lmcache.py --base-url http://<EXTERNAL IP OF vLLM>/v1 --model <LLM MODEL>

================================================================
  LMCACHE PERFORMANCE RESULTS  (16:10:54)
================================================================
  Shared context  : ~2393 tokens (cached by LMCache on disk)
  Question length : ~20 tokens (unique per request, not cached)
  max_tokens      : 1 (TTFT measurement mode)
  Requests        : 1 prompts × 2 phases = 2 total
  Cold successes  : 1 / 1
  Warm successes  : 1 / 1

  Metric                          COLD (no cache)  WARM (disk hit)
  ──────────────────────────────  ───────────────  ────────────────
  Min TTFT                                 757ms            134ms
  Median TTFT                              757ms            134ms
  p95 TTFT                                 757ms            134ms
  Max TTFT                                 757ms            134ms

  SPEEDUP  ████████████  5.6X

  ~  5.6X improvement — increase context length for 10X target.
     Tip: SHARED_CONTEXT needs to be longer relative to the question.

  Per-request breakdown:
    #  User       Cold TTFT   Warm TTFT   Speedup
  ---  --------  ----------  ----------  --------
    1  user1           757ms        134ms      5.6X

================================================================
```

The first run took 757 ms TTFT, which is still less than the 1387 ms of the initial cold run.

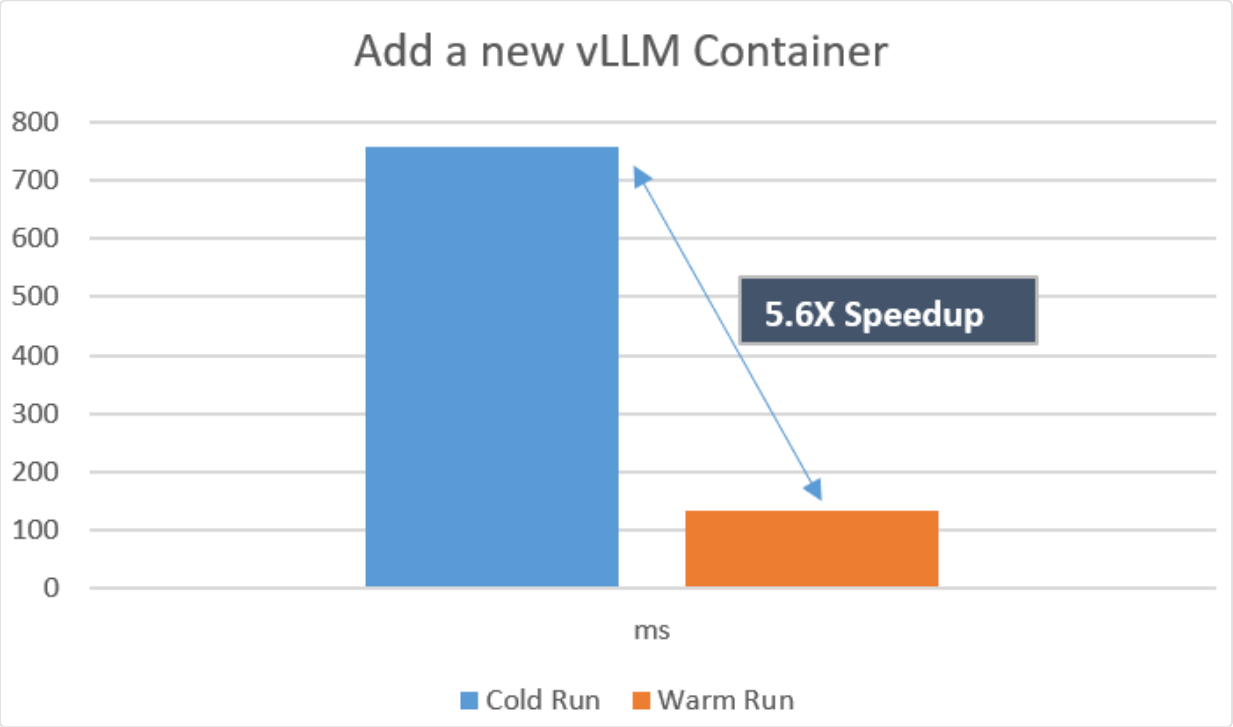

Figure: Performance results chart with 2 vLLM replicas showing 5.6x TTFT improvement (757ms cold vs 134ms warm) and shared cache access across containers

The subsequent run took 134 ms TTFT, indicating that the second vLLM container can access the shared KV cache stored in the IBM Storage Scale PVC.

In summary, IBM Storage Scale Container Native delivers high-performance shared storage that can be mounted seamlessly across multiple vLLM containers for KV cache storage. It also enables independent scaling of the storage layer and supports persistent KV cache retention across container and node restarts.

Summary

IBM Storage Scale Container Native accelerates generative AI workloads by providing high-performance shared storage for key-value cache persistence across multiple vLLM inference instances. The integration with vLLM Production Stack and LMCache extends KV cache beyond single-GPU memory constraints to a distributed storage tier, enabling cache reuse across requests, containers, and nodes. This architectural approach addresses fundamental limitations in standard GPU-based caching by providing persistent storage that survives container restarts and supports concurrent access from multiple inference instances.

Performance validation demonstrates substantial improvements in time-to-first-token metrics when cache hits are served from Storage Scale-backed persistent volumes. Single-instance deployments achieved a 10.8x speedup between cold and warm runs, reducing TTFT from 1387 ms to 129 ms by bypassing prefill computation through cached key-value retrieval. Container restart scenarios validated cache persistence with a 4.8x speedup immediately following initialization, eliminating cache warm-up penalties. Multi-instance deployments extended these benefits through shared cache access, where a second vLLM container immediately leveraged cache entries from the first instance without independent population.

The architectural separation between Storage Scale clusters and Kubernetes compute environments provides operational flexibility for independent scaling and lifecycle management. Storage capacity and performance can be adjusted independently of inference instance scaling, while persistent cache retention across failures improves system resilience and reduces recovery time. This integration demonstrates how IBM Storage Scale Container Native serves as a critical performance acceleration layer for AI inference workloads, enabling organizations to optimize GPU utilization and reduce inference latency through intelligent cache management backed by enterprise-grade distributed storage infrastructure.

High availability, data sharing, and disaster recovery

This section explores high availability, data sharing, and disaster recovery strategies for IBM Storage Scale Container Native deployments. It examines three dual-site scenarios: a stretched Storage Scale cluster providing storage to multiple independent Kubernetes clusters, local Scale clusters on each site using cached S3 storage for data sharing, and full disaster recovery with replicated block storage. It addresses the challenges of achieving site redundancy while avoiding single points of failure, including the limitations of stretched Kubernetes clusters and the complexities of coordinating storage replication with application failover.

This section demonstrates practical implementations using static persistent volume provisioning to enable data sharing across sites, discusses the trade-offs between synchronous and asynchronous replication approaches, and provides configuration examples for building resilient multi-site architectures that balance availability, performance, and operational complexity.

Introduction to disaster recovery

IBM Storage Scale clusters can be configured in a stretched setup, with half of the nodes on one site, and half of the nodes on another site. A third site is needed for a quorum node, which can be a small VM. The replication of data is performed by Scale, with the filesystem fully replicated across the two sites. Again, a filesystem quorum disk (without data) is needed on the third site.

This dual datacenter setup is supported with restrictions on latency as the replication is synchronous. What this setup does not solve is redundancy in case of cluster wide issues or filesystem corruption. Backups are still required.

A method that is more like Disaster Recovery-style is to have all Storage Scale storage on external block storage such as IBM FlashSystem® and leave the IBM FlashSystem to replicate either synchronously or asynchronously. On the remote site the similar Scale nodes need to be present to import the filesystem using a definition file into a new cluster. This method is more of a manual approach. It solves the cluster issue, but not the filesystem issue.

Kubernetes can be stretched across two sites, with distance limitations posed by latency requirements for etcd communication. However, using a single Kubernetes cluster means that any issue with the cluster is also present on the other site. This is particularly perilous when upgrading, as a failed upgrade affects both sites, and a full re-deployment may be the only solution.

Combining a stretched cluster on two sites with a separate DR cluster on a third site would solve both issues. The CNSA Scale cluster is currently (v6.0.0) not supported to be defined in a Kubernetes stretched cluster. The issue is with quorum settings which need a third site with a worker node, not just an arbiter node. For Red Hat OpenShift, it is possible to use two separate clusters and use Red Hat Advanced Cluster Manager (ACM) to coordinate the pods active on each cluster. However, ACM only supports ODF/FDF with asynchronous storage replication. This section describes three dual site scenarios:

1. A single stretched Scale cluster providing storage to two Kubernetes clusters
2. Local Scale clusters on each site using cached S3 storage for data sharing
3. Full Disaster Recovery with replicated block storage

Stretched Scale cluster

Having an external Storage Scale cluster that is stretched across two sites provides site redundancy for the data. When each site has its own independent Kubernetes cluster, each can access the same Scale filesystem using IBM Storage Scale Container Native. When defining PVCs in a dynamic way the control over the Storage Scale filesets that provide the PV data source is with the Kubernetes cluster that creates them, allowing for volume expansion, cloning, and snapshotting. Using static volumes requires the Kubernetes administrator to define the Storage Scale fileset manually, and then define a PV and PVC using manifests. This means all control is with the administrator, but Kubernetes then cannot manage the PVs anymore.

When static PVCs are defined in the Kubernetes cluster on one site, the filesets they are based on are also available on the other site as the underlying Storage Scale cluster is stretched across the two sites with a quorum node on a third site.

DR is performed by defining the static PVCs on the primary and secondary and re-deploying the pods. The YAML files for this must be prepared in advance. It is up to the administrator to make sure the pod is only active on a single cluster and the global router points to the correct cluster.

This diagram shows the relationships between the external Storage Scale cluster and the two Kubernetes clusters:

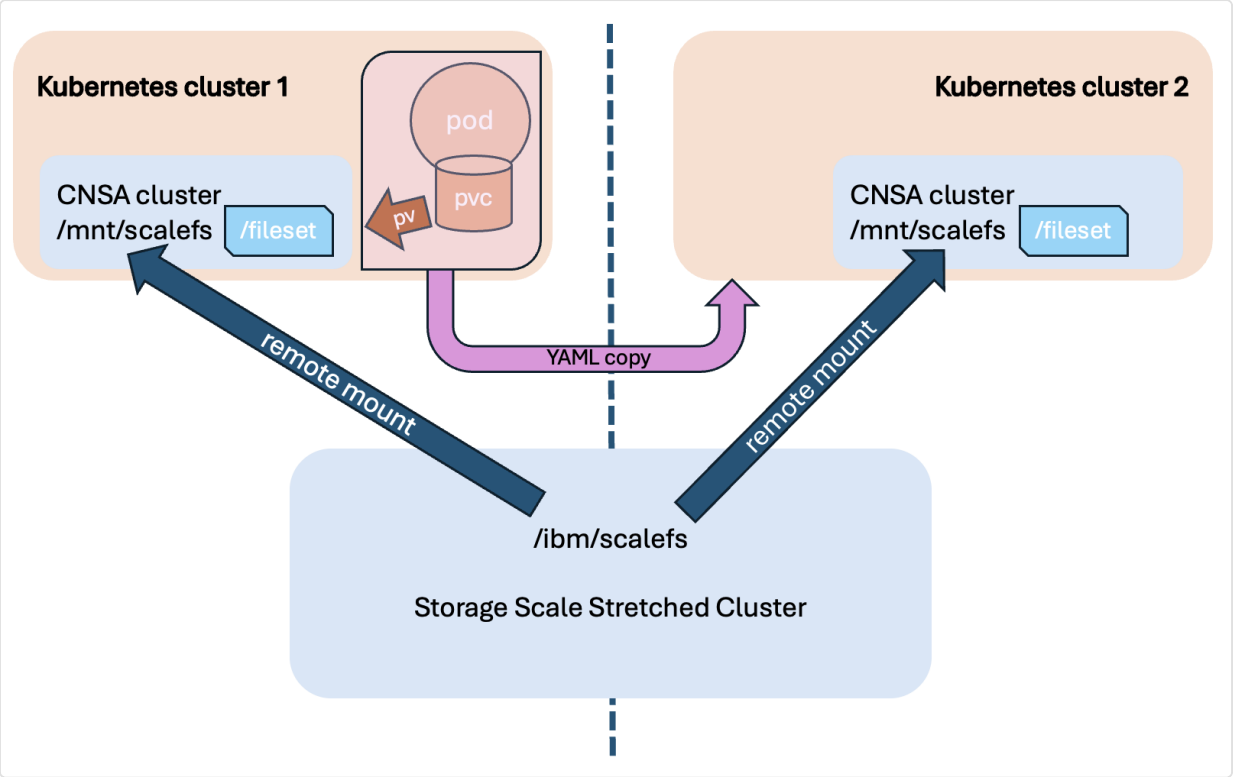

Figure: Topology diagram showing IBM Storage Scale stretched cluster configuration across two sites with quorum node on third site

To build the above example, define the external scale cluster on both Kubernetes nodes, define the remote filesystem (keep the name the same for clarity). This section skips the creation of a Scale remote cluster in CNSA, and starts with defining a static storage class defined on both Kubernetes clusters:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
    name: sc-scalefs-static
provisioner: spectrumscale.csi.ibm.com
parameters:
    volBackendFs: scalefs
    existingVolume: "yes"
reclaimPolicy: Retain
```

On the external scale cluster, the fileset that is to be used by the static Persistent Volume needs to be defined and linked (mounted).

```
mmcrfileset scalefs scalefs-static-fs1 --inode-space=new
mmlinkfileset scalefs scalefs-static-fs1 -J /ibm/scalefs/scalefs-static-fs1
mmsetquota scalefs:scalefs-static-fs1 --block 5G:5G
```

The fileset details are needed for the fileset to be properly identified in the Persistent Volume Definition:

- Get the cluster ID of the external scale cluster:

```
mmlscluster | grep id:
```

- Get the filesystem UID:

```
mmlsfs scalefs --uid
```

To create the Persistent Volume, an identifying `volumeHandle` string must be constructed. It contains the following parameters:

```
0;[Volume type];[Cluster ID];[Filesystem UUID];;[Fileset name];[Path to the directory or fileset linkpath]
```

To create the PV/PVC pair, apply the following YAML with the customized `volumeHandle` string:

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: scalefs-static-pv1
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteMany
  csi:
    driver: spectrumscale.csi.ibm.com
    volumeHandle: 0;2;6662971415901116805;638E11AC:695E1A80;;scalefs-static-fs1;/mnt/scalefs/scalefs-stati
  storageClassName: sc-scalefs-static
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: scalefs-static-pvc1
  namespace: testnamespace
spec:
  volumeName: scalefs-static-pv1
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 5Gi
  storageClassName: sc-scalefs-static
```

The method described here is the manual way to create the manifests. A more automated method is described in the documentation and uses a script called `generate_static_provisioning_yamls.sh`

For more information, see Static provisioning using manifests

To demonstrate the usage of the static PVCs, the following manifest creates an application that appends timestamps to a log file:

```
apiVersion: v1
kind: Pod
metadata:
  name: app-static
  namespace: testnamespace
spec:
  containers:
  - name: app
    image: nginx
    command: ["/bin/sh"]
    args: ["-c", "while true; do echo $MY_NODE_NAME:$(date -u) >> /test/log.txt; sleep 10; done"]
    volumeMounts:
    - name: scalefs-static-pvc1
      mountPath: /test
    env:
      - name: MY_NODE_NAME
        valueFrom:
          fieldRef:
            fieldPath: spec.nodeName
  volumes:
  - name: scalefs-static-pvc1
    persistentVolumeClaim:
      claimName: scalefs-static-pvc1
```

Deploying the PV/PVC/Pod manifest on the second cluster shows the power of using a shared Scale filesystem for storing pod data. Both pods can write to the log file. On the external scale cluster the file can be read and the output would be similar to the following:

```
tail -f /ibm/scalefs/scalefs-static-fs1/log.txt
node1.cluster1.amsiic.ibm.com:Thu Feb 5 12:51:58 UTC 2026
node3.cluster2.amsiic.ibm.com:Thu Feb 5 12:52:04 UTC 2026
node1.cluster1.amsiic.ibm.com:Thu Feb 5 12:52:08 UTC 2026
node3.cluster2.amsiic.ibm.com:Thu Feb 5 12:52:14 UTC 2026
```

While simultaneous writing may be a use case for some applications, this is not desirable for other applications such as databases. To avoid corruption, use proper procedures to make sure only one pod is active.

Caching volume to object store for data sharing

A Cache volume is a Storage Scale fileset where all data is uploaded to an S3 object store. If the data does not fit into the volume, file data will be evicted and only file metadata remains in the cache volume. If the data is accessed the file data is retrieved from the object store. This caching behavior can accelerate data access times from an S3 object store by as much as a factor of 10.

A cache volume can be configured in these four caching modes:

- ReadOnly: Use this mode when an application needs to only read from an object storage bucket.
- Exclusive: Use this mode when an application needs to both read from and write to an object storage bucket. Make sure that only a single application writes to the object storage bucket in this case.
- Parallel: Use this mode when multiple applications need to both read from and write to an object storage bucket.
- Detached: Use this mode when an application needs to cache from an object storage bucket and modify the local cached copy without writing the changes back to the bucket.

The Scale filesystem can be provided by either a local cluster or a remote cluster. Additional CPU and memory resources are needed to manage the cache and queuing to the S3 gateway. See cluster custom resource definition and the client role section in the IBM Storage Scale Container Native manual for more details.

Cache volumes can be used in Parallel mode within a single Kubernetes cluster, but also across multiple clusters.

To create a shared cache volume a replicated S3 object store, or an object store that is stretched across two sites can be used. The IBM Ceph multizone RGW configuration is a good example for a replicated object store, while IBM Storage Scale can deliver a stretched object store using S3 protocol services.

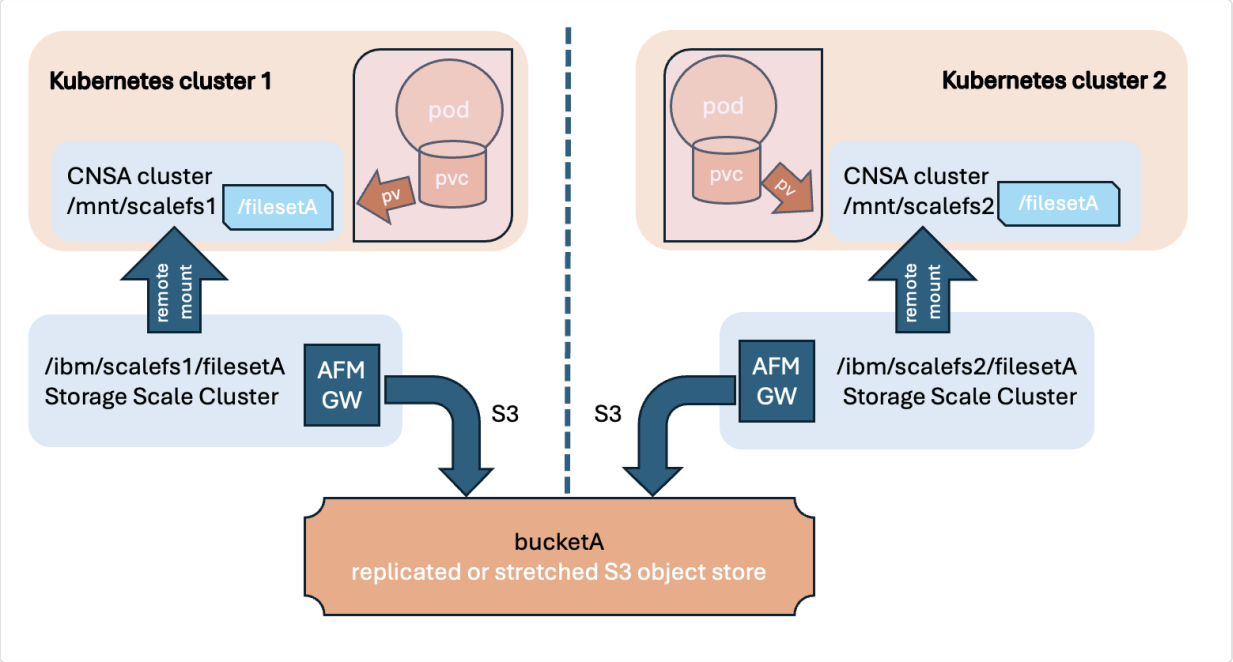

Figure: Diagram showing cache volume configuration with S3 object store backend for cross-cluster data sharing and disaster recovery

This setup is primarily intended to be used for data sharing, though Disaster Recovery use cases may also be covered. Make sure proper procedures are in place to prevent dual access in case simultaneous writing to the fileset causes data corruption.

To set up the cache volume system, first create a bucket in the S3 object store, then create a secret containing the bucket definition, the access key and the secret key:

```
kubectl create secret generic SECRETNAME -n NAMESPACE --from-literal=endpoint= https://s3server --from-lit
```

Create a new storage class for this bucket:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: STORAGE-CLASS
provisioner: spectrumscale.csi.ibm.com
parameters:
  volBackendFs: "FILESYSTEM-NAME"
  volumeType: "cache"
  cacheMode: "Parallel"
  csi.storage.k8s.io/provisioner-secret-name: SECRETNAME
  csi.storage.k8s.io/provisioner-secret-namespace: NAMESPACE
reclaimPolicy: Delete
allowVolumeExpansion: true
Then create the PVC:
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: PVC-NAME
  namespace: NAMESPACE
spec:
  storageClassName: STORAGE-CLASS
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 100Gi
```

File data is fetched on demand, or by using a `CacheVolumeOperation`. For more information, see CacheVolumeOperation Spec. By specifying a `contentFilter` option, files that match the filter are pre-fetched. Eviction of file data starts when 70% of the quota is reached, so size accordingly. Eviction can also be done manually using the `CacheVolumeOperation`.

Full disaster recovery using storage-based replication

In case of a major disaster where one site fails completely the recovery mechanism cannot be dependent on anything happening, not happening, or in an unknown state on that site. Therefore the recovery needs to be performed using independent resources, so one Scale and one Kubernetes cluster per site.

IBM Storage Scale provides for this in the way of an active-passive filesystem setup, where each site has a Scale cluster, and the filesystem definition is transferred from the production cluster to the recovery cluster. The disks that the filesystem is built on need to be replicated by the storage server that is providing them. In case of disaster, the replication is stopped for the disks on the recovery site and disks are then assigned to the Scale cluster.

The Kubernetes namespace definitions need to be recovered as well. There are two main options for this:

1. GitOps redeploy
2. Restore from backup

The Gitops method relies on the storage (PV/PVCs) being present on the recovery cluster before redeployment. If there is full control of the application this is the preferred method.

The Restore from backup uses an external S3 object store which is accessible from both the production and recovery clusters. Among the methods available are:

1. For Kubernetes: Velero
2. For Red Hat OpenShift: OADP
3. For IBM Fusion: Fusion Backup & Restore

These methods not only backup the namespace definition, but can also backup the content of the Persistent Volumes. Depending on the amount of data, the speed of restore, the Recovery Time Objective (RTO), and the Recovery Point Objective (RPO) using backup and restore may be sufficient for some environments.

However, if the RPO needs to be zero, or close to zero, and the amount of data in the Persistent Volumes combined with the RTO makes a full restore too slow, a continuous replication mechanism is needed.

This diagram shows how the data and definition streams work in this scenario:

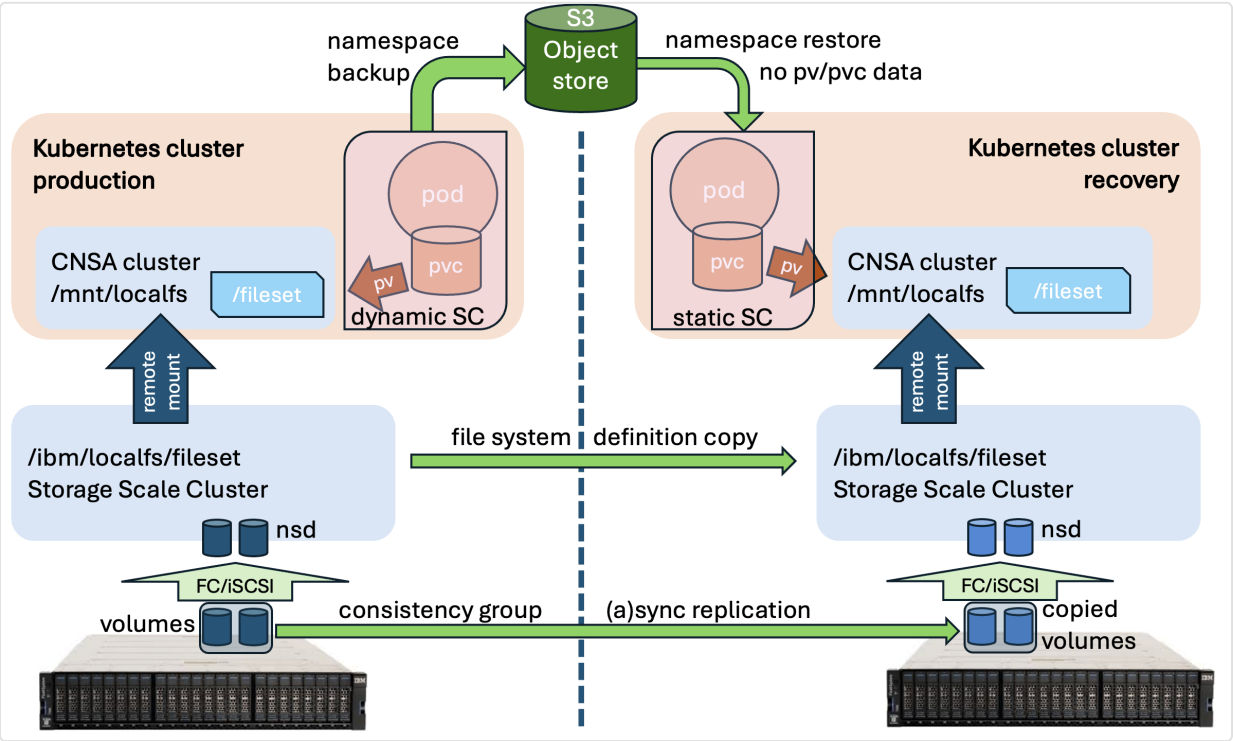

Figure: Architecture diagram illustrating full disaster recovery setup using storage-based replication with primary and recovery clusters

For more information, see Active-passive Storage Scale cluster replication.

Scale storage details

IBM Storage Scale filesystems are self contained on the storage. The definition of which Network Shared Disks (NSDs) contain the filesystem is stored in a text file. To import a filesystem into another cluster, the text file and the disks must be available. Each NSD has a unique identifier written to it at creation time, this enables Storage Scale to detect the disk when it is present on the local Linux server as defined in the filesystem definition text file.

Only when the number of disks in a filesystem changes, do the filesystem definitions need to be imported again in the recovery cluster. The size of the disks and the definitions inside the filesystems such as filesets are determined when the file system is activated on the recovery site.

From the storage interface such as IBM FlashSystem, use Policy based HA or Policy based replication to make FC attached drives available on a remote site. Once the decision is made to switch to the other Kubernetes cluster Disaster Recovery is performed by stopping the FC disk replication and assigning the disks to the other Storage Scale cluster and importing the filesystem. Then the filesystem can be defined in Kubernetes and the static PVCs can be defined and application pods redeployed.

All information for all PVs and PVCs need to be copied over to the DR site as well, as both the static and the dynamically provisioned PVCs that were used on the originating cluster need to be redefined as static PVCs and PVs.

Disaster recovery demonstration setup

This section provides a practical demonstration of disaster recovery procedures using IBM Storage Scale Container Native with storage-based replication. The demonstration environment consists of paired production and recovery clusters across both Kubernetes and Storage Scale layers, with SAN storage replication providing the foundation for data protection and recovery capabilities.

To demonstrate the full disaster recovery mechanism, the following setup is used:

- Two Kubernetes clusters, one production, one recovery cluster, each with an IBM Storage Scale Container Native cluster.
- Two Storage Scale clusters, one production, one recovery cluster, with SAN storage.
- Two Storage Servers which use replication for the Scale disks in a consistency group.
- The Recovery cluster runs MinIO S3 service to store the namespace backups.
- Velero is installed on both clusters, pointing to a MinIO S3 bucket.

The recovery procedure is as follows:

1. Stop storage replication and assign volumes to Scale nodes in the recovery cluster.
2. Activate the Scale file system.
3. Recreate Storage Classes with the same name with static provisioning.
4. Recreate all PVs and PVCs for each namespace.
5. Restore the namespace backup from Velero.

All recovery actions are performed on a management server that can run commands on the Kubernetes cluster.

The disaster recovery process is split into three sections:

1. Preparation: This needs to be performed only once.
2. Replication: This needs to be repeated when there is a structural change.
3. Recovery.

Preparation step 1: Creating Scale resources

In Storage Scale we need a filesystem that is hosted on a SAN storage server. In the demonstration environment we have a filesystem called `localfs` that uses two disks from the storage server. On the production storage server these two volumes are configured in a consistency group and this is replicated to two volumes on the recovery storage server. This demonstration does not go into detail on how the replication is set up on the SAN storage server.

On the production Scale cluster there are two nodes called nsdserver1 and nsdserver2 with two shared disks. In Storage Scale this setup looks like the following configuration:

```
# /usr/lpp/mmfs/bin/mmlsnsd
 File system   Disk name       NSD servers
---------------------------------------------------------
 localfs       localnsd1       nsdserver1,nsdserver2
 localfs       localnsd2       nsdserver2,nsdserver1

# /usr/lpp/mmfs/bin/mmlsdisk localfs -m
Disk name     IO performed on node     Device             Availability
------------  -----------------------  -----------------  ------------
localnsd1     localhost                /dev/dm-5          up
localnsd2     localhost                /dev/dm-6          up

# multipath -ll
mpathb (360050768018e0388500000000000000304) dm-5 IBM,2145
size=99G features='1 queue_if_no_path' hwhandler='1 alua' wp=rw
|-+- policy='service-time 0' prio=50 status=active
| `- 33:0:0:0 sdd 8:48  active ready running
`-+- policy='service-time 0' prio=10 status=enabled
  `- 34:0:0:0 sdi 8:128 active ready running
mpathc (360050768018e0388500000000000000305) dm-6 IBM,2145
size=99G features='1 queue_if_no_path' hwhandler='1 alua' wp=rw
|-+- policy='service-time 0' prio=50 status=active
| `- 34:0:0:1 sdj 8:144 active ready running
`-+- policy='service-time 0' prio=10 status=enabled
  `- 33:0:0:1 sde 8:64  active ready running
```

The recovery cluster in this demonstration setup uses nsdserver3 and nsdserver4 as cluster nodes. These nodes are described in the next section.

On the Kubernetes cluster we assume for this demo that the Scale cluster has already been defined in the production Kubernetes cluster. We do need to define a storage class for the `localfs` filesystem using this manifest:

```
apiVersion: scale.spectrum.ibm.com/v1beta1
kind: Filesystem
metadata:
  labels:
    app.kubernetes.io/instance: ibm-spectrum-scale
    app.kubernetes.io/name: cluster
  name: localfs
  namespace: ibm-spectrum-scale
spec:
  remote:
    cluster: cluster1
    fs: localfs
  seLinuxOptions:
    level: s0
    role: object_r
    type: container_file_t
    user: system_u

NB: seLinuxOptions applicable to Red Hat OpenShift only, for Kubernetes replace with:

seLinuxOptions: {}
```

You must have a dynamic storage class, a namespace, a PVC/PV, and an app on the production cluster to test the recovery. Create those with the following manifest:

```
Define the storage class first:

---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: sc-localfs-independent
provisioner: spectrumscale.csi.ibm.com
parameters:
  volBackendFs: localfs
  filesetType: independent
  uid: "1000"
  gid: "1000"
  shared: "true"
reclaimPolicy: Retain
allowVolumeExpansion: true

Define the namespace and application next:
---
apiVersion: v1
kind: Namespace
metadata:
  name: local-ns
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: localapp-pvc
  namespace: local-ns
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 33Gi
  storageClassName: sc-localfs-independent
---
apiVersion: v1
kind: Pod
metadata:
  name: localapp
  namespace: local-ns
spec:
  containers:
  - name: localapp
    image: nginx
    command: ["/bin/sh"]
    args: ["-c", "while true; do echo $MY_NODE_NAME:$(date -u) >> /test/log.txt; sleep 10; done"]
    volumeMounts:
    - name: localtestpv
      mountPath: /test
    env:
      - name: MY_NODE_NAME
        valueFrom:
          fieldRef:
            fieldPath: spec.nodeName
  volumes:
  - name: localtestpv
    persistentVolumeClaim:
      claimName: localapp-pvc
```

Preparation step 2: Creating an object store

The S3 object store used by velero can be any compatible store. It can be hosted on a public cloud such as IBM's cloud, or it can be a local Store like IBM's Storage Ceph.

For a full list of supported providers, see Providers.

In this demonstration setup, the MinIO store is used because it can be simply deployed inside the recovery Kubernetes cluster.

MinIO is a simple setup if we just need some storage for the objects. The following YAML file deploys a MinIO instance and makes it accessible from outside the Kubernetes cluster. Modify the highlighted items:

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: minio
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: minio-pvc
  namespace: minio
spec:
  storageClassName: sc-cluster2-independent
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: minio
  namespace: minio
spec:
  replicas: 1
  selector:
    matchLabels:
      app: minio
  template:
    metadata:
      labels:
        app: minio
    spec:
      containers:
        - name: minio
          image: quay.io/minio/minio:latest
          args:
            - server
            - /data
          env:
            - name: MINIO_ROOT_USER
              value: "minio"
            - name: MINIO_ROOT_PASSWORD
              value: "secure-minio-password"
            - name: MINIO_CONSOLE_ADDRESS
              value: ":9001"
          ports:
            - containerPort: 9000
            - containerPort: 9001
          volumeMounts:
            - name: data
              mountPath: /data
      volumes:
        - name: data
          persistentVolumeClaim:
            claimName: minio-pvc
---
apiVersion: v1
kind: Service
metadata:
  name: minio-api-service
  namespace: minio
```

```
spec:
  ports:
    - port: 9000
      targetPort: 9000
      protocol: TCP
  selector:
    app: minio
  type: NodePort

---
apiVersion: v1
kind: Service
metadata:
  name: minio-webconsole-service
  namespace: minio
spec:
  ports:
    - port: 9001
      targetPort: 9001
      protocol: TCP
  selector:
    app: minio
  type: NodePort
---
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: minio-api-route
  namespace: minio
spec:
  to:
    kind: Service
    name: minio-api-service
  port:
    targetPort: 9000
  tls:
    termination: edge
    insecureEdgeTerminationPolicy: Allow
  wildcardPolicy: None
---

For Red Hat OpenShift, define a route to expose the service:

apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: minio-webconsole-route
  namespace: minio
spec:
  to:
    kind: Service
    name: minio-webconsole-service
  port:
    targetPort: 9001
  tls:
    termination: edge
    insecureEdgeTerminationPolicy: Allow
  wildcardPolicy: None
```

Create a `credentials-minio-velero` file in your home directory to access MinIO. This file is needed by Velero.

```
cat << EOF > ~/credentials-minio-velero
[minio-velero]
aws_access_key_id = minio
aws_secret_access_key = secure-minio-password
EOF
```

Create a bucket by using either the `aws` CLI or the MinIO console that is accessible at:

```
https://minio-webconsole-route-minio.apps.yourcluster.yourcompany.com

cat ~/credentials-minio-velero >> ~/.aws/credentials

aws --profile minio-velero --endpoint http://minio-api-route-minio.apps.yourcluster.yourcompany.com s3 mb
```

Preparation step 3: Installing Velero

Velero is a tool to backup and restore Kubernetes resources. The client program is installed on a management system where `velero` commands are then executed. Velero needs to be installed on both the production and the recovery Kubernetes cluster.

For more information, see Velero.

To install Velero, download the appropriate executable from GitHub

Extract the tar file and store the `velero` executable in `/usr/local/bin`.

Install Velero into Kubernetes with the following command: (update the s3URL with the exposed minio service)

```
velero install \
  --provider aws \
  --plugins velero/velero-plugin-for-aws:v1.9.2 \
  --bucket velero-backup\
  --backup-location-config s3Url=http://minio-api-route-minio.apps.yourcluster.yourcompany.com \
  --secret-file ./credentials-minio-velero
```

Check the status of Velero:

```
kubectl get deployments -l component=velero --namespace=velero

NAME     READY   UP-TO-DATE   AVAILABLE   AGE

velero   1/1     1            1           57s
```

Full backups of namespaces can now be made with the `velero` command:

```
velero backup create NAMESPACE-BACKUP --include-namespaces NAMESPACE
```

Because the PVCs on the production site are of the dynamic type, snapshots are supported, so a full backup by Velero is possible. On the recovery cluster the PVCs need to be defined as static, so snapshots are not supported. This means that any backup created on the recovery cluster needs to exclude the persistent volumes with this option: `--exclude-resources pv,pvc`.

Replication step 1: Creating clean persistent volume manifests

All persistent Volumes and Persistent Volume Claims in a namespace on the production cluster need to be stored and copied to the management server of the recovery cluster. A Storage Class with the same name as is used in the production cluster also needs to be present on the recovery cluster. But in the definition of the storage class add a line `existingVolume: "yes"` to the parameters section to turn the dynamic storage class into a static provisioning storage class.

For example:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: sc-localfs-independent
provisioner: spectrumscale.csi.ibm.com
parameters:
  volBackendFs: localfs
  filesetType: independent
  existingVolume: "yes"
  uid: "1000"
  gid: "1000"
  shared: "true"
reclaimPolicy: Retain
allowVolumeExpansion: true
```

The PV and PVC manifests on the production cluster can be exported as YAML manifests. Before these manifests can be applied to the recovery cluster, they need to be stripped of unique information. To make this easier these two Python scripts are useful:

```
cat << EOF > /usr/local/bin/cleanup_pvc_data.py
import yaml
import os
import argparse
from yaml.loader import SafeLoader

parser = argparse.ArgumentParser()
parser.add_argument('-i', '--inputfile',
                    type=argparse.FileType('r'),
                    help="Input YAML filename",
                    required=True)
args = parser.parse_args()
outfile = os.path.splitext(args.inputfile.name)[0] + '_cleaned.yaml'

data = yaml.load(args.inputfile, Loader=SafeLoader)
del data['metadata']['finalizers']
del data['metadata']['resourceVersion']
del data['metadata']['uid']
del data['metadata']['annotations']
del data['status']
print(yaml.dump(data))
with open(outfile, 'w') as f:
  result = yaml.dump(data, f, sort_keys=False, default_flow_style=False)
EOF
cat << EOF > /usr/local/bin/cleanup_pv_data.py

import yaml
import os
import argparse
from yaml.loader import SafeLoader

parser = argparse.ArgumentParser()
parser.add_argument('-i', '--inputfile',
                    type=argparse.FileType('r'),
                    help="Input YAML filename",
                    required=True)
args = parser.parse_args()
outfile = os.path.splitext(args.inputfile.name)[0] + '_cleaned.yaml'

data = yaml.load(args.inputfile, Loader=SafeLoader)
del data['metadata']['creationTimestamp']
del data['metadata']['finalizers']
del data['metadata']['resourceVersion']
del data['metadata']['uid']
del data['spec']['csi']['volumeAttributes']['storage.kubernetes.io/csiProvisionerIdentity']
del data['spec']['claimRef']['resourceVersion']
del data['spec']['claimRef']['uid']
del data['status']
print(yaml.dump(data))
with open(outfile, 'w') as f:
  result = yaml.dump(data, f, sort_keys=False, default_flow_style=False)
```

These Python files are used by the `clean_ns_storage.sh` shell script that uses them on PV/PVC manifests inside your home directory on the production site:

```
cat << EOF > /usr/local/bin/clean_ns_storage.sh
#!/bin/sh
STORAGECLASS=$1
NAMESPACE=$2

if [ "$NAMESPACE" = "" ]
then
   echo "Usage: clean_ns_storage.sh STORAGECLASS NAMESPACE"
   exit 1
fi

mkdir -p ~/manifests/$NAMESPACE/$STORAGECLASS

PVS=$(kubectl get pv | grep -w $STORAGECLASS | grep " $NAMESPACE/" | awk '{print $1}')
for i in $PVS
do
    kubectl get pv $i -o yaml > ~/manifests/$NAMESPACE/$STORAGECLASS/$i.yaml
    python /usr/local/bin/cleanup_pv_data.py -i ~/manifests/$NAMESPACE/$STORAGECLASS/$i.yaml
    echo "stored PV $i in ~/manifests/$NAMESPACE/$STORAGECLASS"
done

PVCS=$(kubectl get pvc -n $NAMESPACE | grep $STORAGECLASS | awk '{print $1}')
for i in $PVCS
do
    kubectl get pvc $i -o yaml > ~/manifests/$NAMESPACE/$STORAGECLASS/$i.yaml
    python /usr/local/bin/cleanup_pvc_data.py -i ~/manifests/$NAMESPACE/$STORAGECLASS/$i.yaml
    echo "stored PVC $i in ~/manifests/$NAMESPACE/$STORAGECLASS"
done
```

Run the `clean_ns_storage.sh` script for each namespace and storage class combination in the production cluster:

```
chmod +x /usr/local/bin/clean_ns_storage.sh
clean_ns_storage.sh sc-localfs-independent local-ns
```

The storage manifests need to be copied to the recovery management server. One option is to upload them to the S3 object store, another is to use the `scp` command.

Replication step 2: Creating the file system definition on the Scale recovery cluster

For the Storage Scale filesystem definition replication the production Scale cluster requires remote command access to the recovery cluster. The default is to have root SSH permission from the node running the `mmfsctl` command to all contact nodes. Use of `sudo` wrappers and `scaleadm` API calls are also possible as a transfer mechanism, see the `mmfsctl` manual page for more information.

On the production Scale cluster, create a contact node file with all recovery cluster nodes: (replace nodes with your nodenames)

```
cat << EOF > /var/mmfs/etc/recovery-contact-nodes
nsdserver3.company.com
nsdserver4.company.com
EOF
```

Create a map file for each NSD server that is used in the production cluster to a NSD server in the recovery cluster as these servers will be different:

Format of this file is `production-nsd-servername recovery-nsd-servername`. For example:

```
cat << EOF > /var/mmfs/etc/nsdserver-mapfile
nsdserver1.company.com nsdserver3.company.com
nsdserver2.company.com nsdserver4.company.com
EOF
```

We can use this mapfile to create text replacement commands for use with sed, and then create a specfile that is used by the `mmfsctl` command to modify the filesystem/NSD definition when importing the filesystem on the recovery cluster.

```
cat << EOF > /usr/local/bin/sync_fs_definition.sh
fsname=$1
if [ "$fsname" = "" ]
then
   echo "Usage: sync_fs_definition.sh scale-fs-name"
   exit 1
fi

cat /var/mmfs/etc/nsdserver-mapfile | while read prodnode recovnode
do
   echo "s/\\b$prodnode\\b/$recovnode/g"
done > /var/mmfs/etc/nsdserver-mapfile.sed

/usr/lpp/mmfs/bin/mmlsnsd | grep -w $fsname | while read fs nsd servers; do
   remoteservers=$(echo $servers | sed -f /var/mmfs/etc/nsdserver-mapfile.sed)
   echo "%nsd:
   nsd=$nsd
   servers=$remoteservers
"
done > /var/mmfs/etc/${fsname}-specfile

/usr/lpp/mmfs/bin/mmfsctl $fsname syncFSconfig -n /var/mmfs/etc/recovery-contact-nodes -S /var/mmfs/etc/$f
```

We can use the `sync_fs_definition.sh` script to create the definition of the local filesystem on the remote site.

```
# chmod +x /usr/local/bin/sync_fs_definition.sh
# sync_fs_definition.sh localfs

localfs-specfile

100%  122   196.2KB/s   00:00
mmfsctl:  Exporting file system information from the source cluster . . .

mmexportfs: Processing file system localfs ...
mmfsctl:  Importing file system information into the target cluster on node nsdserver3.company.com . . .

mmimportfs: Processing file system localfs ...
mmimportfs: Processing disk localnsd1
mmimportfs: Processing disk localnsd2

mmimportfs: Committing the changes ...

mmimportfs: The following file systems were successfully imported:
    localfs
mmimportfs: mmsdrfs propagation completed.
```

The file system created on the recovery Scale cluster may not be able to access the disks yet, and will show errors when accessed. If the configuration of the NSDs in the filesystem changes, the definition needs to be refreshed. To do this, first delete the filesystem and NSDs on the remote side, then run the `sync_fs_definition.sh` script again.

In the demonstration environment, this would mean running these commands on the recovery Scale cluster:

```
mmdelfs localfs -p
mmdelnsd localnsd1
mmdelnsd localnsd2
```

Replication step 3: Creating a namespace backup by using Velero

On the management server in the production environment run the following `velero` command:

```
# velero backup create local-ns-1 --include-namespaces local-ns --exclude-resources pv,pvc
```

Check the progress of the backup with:

```
# velero get backups
NAME             STATUS             ERRORS   WARNINGS   CREATED                         EXPIRES   STORAGE
local-ns-1       Completed          0        0          2026-02-24 10:42:43 +0100 CET   28d       default
```

Each time something changes in the namespace such as additional containers, secret, container attributes, etc, the backup needs to be run again. For more information, see Velero backup reference.

Recovery step 1: Making the SAN volumes available in Scale

In case of a disaster or planned disaster test, the replication of the volumes between production and recovery site needs to be stopped on the SAN storage server. For some systems the replicated volumes also need to be assigned to the recovery Scale cluster. Once this is done, run the following commands on all NSD servers in the recovery Scale cluster:

```
# rescan-scsi-bus.sh
# iscsiadm -m node -R
# /usr/lpp/mmfs/bin/mmdevdiscover
```

In the demonstration recovery cluster the two volumes become visible:

```
# multipath -ll
mpathc (360050768018e03885000000000000321) dm-6 IBM,2145
size=99G features='1 queue_if_no_path' hwhandler='1 alua' wp=rw
|-+- policy='service-time 0' prio=50 status=active
| `- 33:0:0:3 sdj 8:144 active ready running
`-+- policy='service-time 0' prio=10 status=enabled
  `- 34:0:0:3 sdk 8:160 active ready running
mpathd (360050768018e03885000000000000320) dm-7 IBM,2145
size=99G features='1 queue_if_no_path' hwhandler='1 alua' wp=rw
|-+- policy='service-time 0' prio=50 status=active
| `- 34:0:0:2 sdi 8:128 active ready running
`-+- policy='service-time 0' prio=10 status=enabled
  `- 33:0:0:2 sdh 8:112 active ready running
```

These volumes are now detected by the Scale cluster:

```
# mmlsnsd -m

 Disk name       NSD volume ID      Device          Node name or Class       Remarks
-------------------------------------------------------------------------------------
 localnsd1       AC118E6D699C4AAD   /dev/dm-7       nsdserver3.company.com server node
 localnsd2       AC118E6D699C4AAE   /dev/dm-6       nsdserver4.company.com server node
```

Mount the filesystem on all nodes and check access:

```
/usr/lpp/mmfs/bin/mmmount localfs -a
```

Mounting the filesystem performs a log replay that will fix any issues with open files. Depending whether or how the production Scale cluster was stopped and the replication method, either synchronous or asynchronous, a filesystem check may be needed. Validate the filesystem in any case using the online check:

```
# /usr/lpp/mmfs/bin/mmfsck localfs -n -o
Checking "localfs" on Thu Feb 26 11:46:57 2026
   5 % complete on Thu Feb 26 11:46:57 2026
Starting Inode phase
 100 % complete on Thu Feb 26 11:46:59 2026

             25952256 subblocks
               306203   allocated
                    0   unreferenced
                    0   deallocated

                  574 addresses
                    0   suspended
File system is clean.
Fsck completed in 0 hours 0 minutes 3 seconds

If errors are detected, a full filesystem check is required until the errors are gone:
/usr/lpp/mmfs/bin/mmfsck localfs -y --threads 4 -N all
```

If the file system is not found to be clean and errors are detected, a full filesystem check is required until the errors are gone:

```
/usr/lpp/mmfs/bin/mmfsck localfs -y --threads 4 -N all
```

Recovery step 2: Defining the Scale filesystem in Kubernetes

For the demonstration environment the recovery Scale cluster is assumed to be already defined in the recovery Kubernetes cluster. Check this with:

```
# kubectl get remotecluster -n ibm-spectrum-scale
NAME                  GUI VERSION   READY   AVAILABLE   AGE
cluster2              6.0.0-0       True    True        70d
```

The definition of the replicated filesystem needs to be done using this YAML:

```
apiVersion: scale.spectrum.ibm.com/v1beta1
kind: Filesystem
metadata:
  labels:
    app.kubernetes.io/instance: ibm-spectrum-scale
    app.kubernetes.io/name: cluster
  name: localfs
  namespace: ibm-spectrum-scale
spec:
  remote:
    cluster: cluster2
    fs: localfs
  seLinuxOptions:
    level: s0
    role: object_r
    type: container_file_t
    user: system_u

# seLinuxOptions applicable to Red Hat OpenShift only, for Kubernetes replace with:

seLinuxOptions: {}
```

Next is to create the `StorageClass` with the same name as the one used in the production cluster, but defined as a static class by adding the `ExistingVolume` parameter:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: sc-localfs-independent
provisioner: spectrumscale.csi.ibm.com
parameters:
  volBackendFs: localfs
  filesetType: independent
  ExistingVolume: "yes"
  uid: "1000"
  gid: "1000"
  shared: "true"
reclaimPolicy: Retain
allowVolumeExpansion: true
```

Recovery step 3: Defining the persistent volumes

The Persistent Volumes and Persistent Volume Claims can now be recreated using the cleaned manifests that were created using the replication step. Because the Persistent Volumes are now stored in the recovery cluster, the production Scale cluster ID needs to be replaced with the recovery Scale clusterid.

The cluster id can be determined on a Scale cluster node:

```
mmlscluster | grep id:
```

The Persistent Volume YAML has a definition line `volumeHandle`:

```
volumeHandle: 0;2;14005285791585032265;6D8E11AC:699C4BA4
```

The third value in the line is the cluster ID. This can be swapped as follows:

```
for manifest in ~/manifests/local-ns/*/*_cleaned.yaml
do
    sed -i "s/old-cluster-id/new-cluster-id/"
done
```

First create the namespace manually in the recovery Kubernetes cluster, then define all PVs and PVCs:

```
kubectl new-project local-ns
kubectl apply -f ~/manifests/local-ns/*/*_cleaned.yaml
```

Ensure all the PV YAML files in the directory have the old cluster ID swapped with the new, and all PVCs are created as static volumes using the already present fileset in the recovered Scale filesystem.

Recovery step 4: Defining the namespace

The last step in the recovery process is to restore all resources in the namespace using Velero. First check the available backups:

```
# velero get backups
NAME             STATUS            ERRORS   WARNINGS   CREATED                         EXPIRES   STORAGE
local-ns-1       Completed         0        0          2026-02-24 10:42:43 +0100 CET   28d       default
Then restore the backup:
velero restore create --from-backup local-ns-1 --exclude-resources pv,pvc
```

Check the progress with the `describe` or `log` option, or view the result:

```
# velero get restore
NAME                            BACKUP           STATUS       STARTED                         COMPLETED
local-ns-1-20260224142507       local-ns-1       InProgress   2026-02-24 14:25:07 +0100 CET   <nil>

kubectl get pods -n local-ns
NAME       READY   STATUS    RESTARTS   AGE
localapp   1/1     Running   0          7m27s
```

When looking at the output in the `test.txt` file of the localapp pod, you can view the moment that the replication stopped and the app restarted in the recovery cluster:

```
kubectl exec --stdin --tty localapp -n local-ns -- tail /test/test.txt
node3.cluster1.amsiic.ibm.com: Tue Feb 24 12:30:42 UTC 2026
node3.cluster1.amsiic.ibm.com: Tue Feb 24 12:30:52 UTC 2026
node3.cluster1.amsiic.ibm.com: Tue Feb 24 12:31:02 UTC 2026
node2.cluster2.amsiic.ibm.com: Tue Feb 24 13:25:17 UTC 2026
node2.cluster2.amsiic.ibm.com: Tue Feb 24 13:25:27 UTC 2026
node2.cluster2.amsiic.ibm.com: Tue Feb 24 13:25:37 UTC 2026
```

This concludes the disaster recovery demonstration.

Reverting back to the production cluster would require the following steps:

1. Reverse the replication direction of the volumes in the SAN storage server.
2. If the original production cluster is still present and defined:
    a. Make sure the volume sync is ready, then stop activity on the recovery Kubernetes cluster.
    b. Stop the replication process for the volumes.

c. Mount the filesystem in the production Scale cluster.

d. Start the activity on the production Kubernetes cluster.

3. If the original production site needed to be rebuilt from scratch:

   a. Reinstall the Scale cluster.

   b. Reinstall the Kubernetes cluster and IBM Storage Scale Container Native cluster.

   c. Use the replication and recovery procedures to rebuild Kubernetes resources.

Summary

IBM Storage Scale Container Native deployments require careful architectural planning to achieve site redundancy and disaster recovery capabilities while avoiding single points of failure. The fundamental challenge lies in coordinating storage replication with application failover across independent infrastructure layers, as stretched Kubernetes clusters introduce operational risks where control plane failures affect both sites simultaneously. Three distinct dual-site approaches address different availability and recovery requirements, each with specific trade-offs between operational complexity, synchronization characteristics, and failure domain isolation.

Stretched Storage Scale clusters provide synchronous data replication across sites while supporting independent Kubernetes clusters at each location. This architecture enables data sharing through static persistent volume provisioning, where manually defined filesets remain accessible from both sites through the distributed filesystem. The approach requires careful coordination to prevent simultaneous write access and depends on proper routing mechanisms to ensure application pods remain active on only one cluster at a time. Latency constraints limit geographic separation, and the shared storage layer represents a potential single point of failure despite site redundancy.

Cache volumes with object storage backends enable data sharing across completely independent clusters through parallel caching modes that synchronize content via replicated or stretched S3 infrastructure. This model provides greater operational independence and supports asynchronous data propagation, making it suitable for scenarios where eventual consistency is acceptable. The cache layer accelerates access to object-stored data while maintaining local modifications, though proper procedures remain essential to prevent corruption from concurrent writes. Resource requirements increase due to cache management overhead, and recovery time depends on cache population characteristics rather than immediate filesystem availability.

Full disaster recovery using storage-based replication establishes active-passive configurations where filesystem definitions transfer between independent Storage Scale clusters following storage server replication cutover. This approach achieves near-zero recovery point objectives through continuous block-level replication while maintaining complete operational independence between sites. Recovery procedures involve stopping replication, assigning volumes to the recovery cluster, activating filesystems, and recreating Kubernetes resources through either GitOps redeployment or backup restoration using tools like Velero. Static persistent volume provisioning becomes mandatory in recovery scenarios, requiring advance preparation of volume manifests and careful tracking of filesystem definitions across infrastructure changes.

The selection among these approaches depends on recovery time objectives, recovery point objectives, acceptable operational complexity, and tolerance for shared infrastructure dependencies. Stretched clusters minimize recovery time but introduce shared failure domains, cache volumes provide operational independence with eventual consistency, and storage-based replication delivers the strongest isolation with manual recovery procedures. All approaches require disciplined operational practices to prevent data corruption from concurrent access and to maintain consistency between storage state and Kubernetes resource definitions across site boundaries.

Abbreviations

This section contains acronyms and abbreviations used throughout this document, organized alphabetically.

Acronym	Full Name	Definition
ACM	Advanced Cluster Manager	Red Hat's multi-cluster management platform for Kubernetes and OpenShift environments. Provides centralized management, governance, and application lifecycle capabilities across multiple clusters.
AFM	Active File Management	IBM Storage Scale feature that enables automated data movement and caching between file systems, supporting disaster recovery and multi-site data management.
AI	Artificial Intelligence	Computer systems designed to perform tasks that typically require human intelligence, including learning, reasoning, and problem-solving. IBM Storage Scale optimizes data access for AI workloads.
API	Application Programming Interface	Set of protocols and tools for building software applications. Defines how software components should interact.
BCM	Base Command Manager	NVIDIA's cluster management platform that simplifies provisioning, configuring, and monitoring GPU-accelerated clusters for HPC, AI, and data science environments.
BLOb	Binary Large Object	A collection of binary data stored as a single entity in a database management system.
BMC	Baseboard Management Controller	Hardware management interface for out-of-band management of computer systems.
CES	Cluster Export Services	IBM Storage Scale services that provide protocol support for file sharing.
CIFS	Common Internet File System	Network file sharing protocol, also known as SMB.
Calico	Calico	Open-source networking and network security solution for containers, virtual machines, and native host-based workloads. Provides CNI implementation for Kubernetes.
CLI	Command-Line Interface	Text-based interface for interacting with software and operating systems. BCM provides cmsh CLI for administration.
CNCF	Cloud Native Computing Foundation	Organization that hosts and promotes cloud-native open source projects.

Acronym	Full Name	Definition
CCR	Cluster Configuration Repository	IBM Storage Scale repository type that stores cluster configuration data in a distributed manner across cluster nodes, providing high availability and consistency.
Ceph	Ceph	Open-source software-defined storage platform that provides object, block, and file storage in a unified system. Supports S3-compatible object storage through RADOS Gateway.
CNI	Container Network Interface	Standard interface for configuring network interfaces in Linux containers.
CNSA	Container Native Storage Access	IBM Storage Scale's containerized implementation for Kubernetes and Red Hat OpenShift. Runs as pods and integrates with Container Storage Interface (CSI) for persistent volumes.
CoreDNS	CoreDNS	DNS server that chains plugins and provides DNS services for Kubernetes clusters. Essential for service discovery and name resolution within the cluster.
CRI-O	Container Runtime Interface - Open	Lightweight container runtime for Kubernetes.
CSI	Container Storage Interface	Standard interface for exposing storage systems to containerized workloads on Kubernetes and other container orchestration platforms.
CUDA	Compute Unified Device Architecture	NVIDIA's parallel computing platform and programming model for GPU acceleration. Enables developers to use GPUs for general-purpose processing.
DB2	Database 2	IBM's relational database management system.
DHCP	Dynamic Host Configuration Protocol	Network protocol that automatically assigns IP addresses and network configuration to devices. Used during BCM node provisioning.
DR	Disaster Recovery	Process of restoring systems and data after a catastrophic event.
EDB	EnterpriseDB	Company providing enterprise-class PostgreSQL database solutions.
ESS	Elastic Storage Server	IBM's scalable storage solution (now part of Storage Scale System).
etcd	etcd	Distributed key-value store used by Kubernetes to store all cluster data. Serves as Kubernetes' backing store for all cluster state and configuration.

Acronym	Full Name	Definition
FC	Fibre Channel	High-speed network technology primarily used for storage networking.
FlashSystem	IBM FlashSystem	IBM's enterprise-class all-flash and hybrid flash storage arrays that provide high-performance block storage with advanced data services.
Fusion	IBM Fusion	IBM's family of software-defined storage solutions including Fusion Data Foundation and Fusion Access for SAN, designed for hybrid cloud and container environments.
FDF	Fusion Data Foundation	IBM's software-defined storage solution based on Red Hat OpenShift Data Foundation.
FNCM	FileNet Content Manager	IBM's enterprise content management system.
GenAI	Generative AI	Artificial intelligence systems capable of generating new content, including text, images, code, and other media, based on learned patterns from training data.
GPFS	General Parallel File System	IBM's high-performance clustered file system, now known as IBM Storage Scale. Provides parallel access to files from multiple nodes with high throughput and scalability.
GPL	GPFS Portability Layer	Loadable kernel module that enables IBM Storage Scale daemon to interact with the operating system. Must be rebuilt when kernel version changes.
Helm	Helm	Package manager for Kubernetes that helps define, install, and upgrade complex Kubernetes applications using charts.
Hugging Face	Hugging Face	Platform and community for machine learning that provides tools, models, and datasets for natural language processing and other AI tasks.
GPU	Graphics Processing Unit	Specialized processor designed for parallel processing, widely used for AI/ML training and inference. IBM Storage Scale supports GPUDirect Storage for optimized data access.
GUI	Graphical User Interface	Visual interface that allows users to interact with software through graphical elements. BCM provides Base View GUI.

Acronym	Full Name	Definition
HA	High Availability	System design approach that ensures a certain level of operational performance for a higher than normal period.
HDD	Hard Disk Drive	Traditional magnetic storage device.
HPC	High-Performance Computing	Computing approach that aggregates computing power to deliver higher performance than typical desktop or workstation systems. Used for complex calculations, simulations, and data analysis.
HTTP	Hypertext Transfer Protocol	Application protocol for distributed, collaborative, hypermedia information systems. BCM uses HTTP for image and file transfers.
ILM	Information Lifecycle Management	Policies and processes for managing data throughout its lifecycle.
I/O	Input/Output	Communication between an information processing system and the outside world.
IPMI	Intelligent Platform Management Interface	Standardized interface for out-of-band management of computer systems. BCM uses IPMI for remote power management and monitoring.
iPXE	Internet Preboot Execution Environment	Enhanced version of PXE that adds support for booting from HTTP, iSCSI, and other protocols. Used by BCM for flexible node provisioning.
iSCSI	Internet Small Computer Systems Interface	IP-based storage networking standard for linking data storage facilities.
Kubernetes	Kubernetes	Open-source container orchestration platform for automating deployment, scaling, and management of containerized applications. Foundation for both upstream Kubernetes and Red Hat OpenShift.
KV	Key-Value	Data storage paradigm designed for storing, retrieving, and managing associative arrays.
KVM	Kernel-based Virtual Machine	Virtualization infrastructure for the Linux kernel.
LDAP	Lightweight Directory Access Protocol	Protocol for accessing and maintaining distributed directory information services. BCM integrates with LDAP for user authentication and authorization.
LLM	Large Language Model	AI model trained on vast amounts of text data to understand and generate human-like text.

Acronym	Full Name	Definition
LMCache	Language Model Cache	Open-source key-value cache engine for LLM inference that extends standard GPU-based caching to multiple tiers including CPU memory and remote storage, enabling cache sharing across GPUs and nodes.
LPAR	Logical Partition	Virtualization technology primarily used in IBM mainframe and Power Systems that divides a single physical server into multiple isolated virtual servers, each running its own operating system.
LSF	Load Sharing Facility	IBM's workload management platform for distributed computing environments. Supports job scheduling across HPC clusters.
LVM	Logical Volume Manager	Device mapper framework that provides logical volume management for the Linux kernel.
MCP	MachineConfigPool	OpenShift resource that groups nodes together for configuration management. Enables consistent OS-level configuration across sets of nodes through Machine Config Operator.
MCO	Machine Config Operator	OpenShift operator that manages operating system configuration on cluster nodes.
mmfsd	mmfsd	IBM Storage Scale daemon process that runs on each cluster node and handles all filesystem operations, metadata management, and cluster communication.
ML	Machine Learning	Subset of AI that enables systems to learn and improve from experience without explicit programming. Requires high-performance storage for training data access.
NFS	Network File System	Distributed file system protocol that allows remote file access over a network. IBM Storage Scale supports NFS protocol for client access.
NLSAS	Near-Line Serial Attached SCSI	Type of hard disk drive designed for high-capacity storage with lower cost per gigabyte.
NSD	Network Shared Disk	IBM Storage Scale component that provides access to storage devices.
NTFS	New Technology File System	Proprietary file system developed by Microsoft.

Acronym	Full Name	Definition
NVMe	Non-Volatile Memory Express	Interface specification for accessing solid-state drives attached through PCIe bus.
OADP	OpenShift API for Data Protection	Red Hat OpenShift operator that provides backup and restore capabilities for cluster resources and persistent volumes using Velero.
OCP	OpenShift Container Platform	Red Hat's enterprise Kubernetes platform.
ODF	OpenShift Data Foundation	Red Hat's software-defined storage solution for OpenShift.
OLM	Operator Lifecycle Manager	Component of the Operator Framework that manages the lifecycle of operators in a Kubernetes cluster.
OpenCL	Open Computing Language	Open standard for parallel programming across heterogeneous platforms including CPUs, GPUs, and other processors.
OpenShift	Red Hat OpenShift	Enterprise Kubernetes platform developed by Red Hat. Provides additional features for enterprise container deployments. Supports IBM Storage Scale CNSA.
OS	Operating System	System software that manages computer hardware and software resources and provides common services for computer programs.
OVN	Open Virtual Network	Software-defined networking solution for Kubernetes and OpenShift.
PBS	Portable Batch System	Workload management system for HPC clusters. BCM supports PBS integration for job scheduling.
PCIe	Peripheral Component Interconnect Express	High-speed serial computer expansion bus standard.
PDU	Power Distribution Unit	Device that distributes electric power to multiple devices in a data center. BCM integrates with PDUs for advanced power management.
PGDATA	PostgreSQL Data Directory	Directory where PostgreSQL stores all database files.
Postgres	PostgreSQL	Open-source relational database management system emphasizing extensibility and SQL compliance. Also known as PostgreSQL.

Acronym	Full Name	Definition
PostgreSQL	PostgreSQL	Open-source relational database management system emphasizing extensibility and SQL compliance. Commonly referred to as Postgres.
POSIX	Portable Operating System Interface	Family of standards for maintaining compatibility between operating systems. IBM Storage Scale provides POSIX-compliant file system interface.
PV	Persistent Volume	Kubernetes resource representing a piece of storage in the cluster.
PVC	Persistent Volume Claim	Kubernetes resource representing a request for storage by a user.
PXE	Preboot Execution Environment	Industry standard for booting computers over a network interface, independent of local storage. Used by BCM for node provisioning.
RAID	Redundant Array of Independent Disks	Data storage virtualization technology that combines multiple physical disk drive components.
RBAC	Role-Based Access Control	Method of regulating access to computer or network resources based on the roles of individual users.
RDMA	Remote Direct Memory Access	Technology that allows direct memory access from one computer to another without involving the operating system, enabling high-throughput, low-latency networking.
REST	Representational State Transfer	Architectural style for designing networked applications.
RHCOS	Red Hat Enterprise Linux CoreOS	Immutable operating system designed for running containerized workloads in OpenShift.
RGW	RADOS Gateway	Ceph object storage interface that provides RESTful gateway to Ceph storage clusters with S3 and Swift compatible APIs.
RHEL	Red Hat Enterprise Linux	Commercial Linux distribution developed by Red Hat. Supported platform for IBM Storage Scale and BCM deployments.
RoCE	RDMA over Converged Ethernet	Network protocol that allows RDMA over Ethernet networks, providing high-performance data transfer with low latency.

Acronym	Full Name	Definition
RPO	Recovery Point Objective	Maximum acceptable amount of data loss measured in time. Defines how much data can be lost before business operations are significantly impacted.
RTO	Recovery Time Objective	Maximum acceptable time to restore systems and data after a disaster or disruption. Defines how quickly systems must be recovered.
RWO	ReadWriteOnce	Kubernetes volume access mode where the volume can be mounted as read-write by a single node.
RWX	ReadWriteMany	Kubernetes volume access mode where the volume can be mounted as read-write by many nodes.
S3	Simple Storage Service	Object storage protocol originally developed by Amazon Web Services. IBM Storage Scale supports S3 protocol for object storage access.
SAN	Storage Area Network	Dedicated high-speed network that provides access to consolidated block-level storage.
SAS	Serial Attached SCSI	Point-to-point serial protocol for connecting storage devices.
SCC	SecurityContextConstraints	OpenShift security mechanism that defines what privileges a pod or service account may request.
SCSI	Small Computer System Interface	Set of standards for physically connecting and transferring data between computers and peripheral devices.
Slurm	Simple Linux Utility for Resource Management	Open-source workload manager for Linux clusters. BCM integrates with Slurm for job scheduling and resource allocation.
SMB	Server Message Block	Network file sharing protocol primarily used by Windows systems. IBM Storage Scale provides SMB protocol support for Windows client access.
SPOF	Single Point Of Failure	Component whose failure would cause the entire system to fail.
SRE	Site Reliability Engineer	Professional responsible for the reliability and performance of production systems.
SSH	Secure Shell	Cryptographic network protocol for secure remote login and command execution over unsecured networks.

Acronym	Full Name	Definition
SSS	Storage Scale System	IBM's integrated storage appliance solution based on Storage Scale.
SVC	SAN Volume Controller	IBM's enterprise storage virtualization system that provides a single point of control for managing multiple storage systems and enabling advanced data services.
TFTP	Trivial File Transfer Protocol	Simple file transfer protocol used for transferring files during network boot processes. BCM uses TFTP for initial boot file delivery.
TTFT	Time To First Token	Metric measuring the time from when a request is sent to when the first token of the response is received in LLM inference workloads.
Velero	Velero	Open-source tool for backing up and restoring Kubernetes cluster resources and persistent volumes. Supports disaster recovery and cluster migration scenarios.
UNIX	UNIX	Family of multitasking, multiuser computer operating systems.
vLLM	vLLM	Fast and easy-to-use library for LLM inference and serving. Developed at UC Berkeley, it provides high-throughput LLM inference with efficient memory management.
vLLM Production Stack	vLLM Production Stack	Open-source project that provides a complete inference system using vLLM with LMCache, prefix-aware routing, observability features, and autoscaling capabilities for Kubernetes deployments.
VM	Virtual Machine	Emulation of a computer system that provides the functionality of a physical computer. IBM Storage Scale can be deployed on VMs.
WAL	Write-Ahead Log	Method of providing atomicity and durability in database systems.
YAML	YAML Ain't Markup Language	Human-readable data serialization language commonly used for configuration files.

Additional resources

This document complements but does not replace IBM product documentation. For detailed API references, complete configuration options, and the latest updates, consult the following additional resources:

IBM Redbooks Publications:

Implementation Guide for IBM Storage Scale System 6000 IBM Storage Scale: Working With Abstract Data

IBM Documentation:

- IBM Storage Scale Container Native documentation
- IBM Storage Scale documentation

Other related Documentation:

- Red Hat OpenShift documentation
- Kubernetes documentation

Notices

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

The following terms are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide:

IBM®	IBM FlashSystem®	Kubernetes®
IBM DB2®	IBM Redbooks®	OpenShift®
IBM FileNet®	IBM Z®	Red Hat®

Red Hat, OpenShift, and RHEL are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Kubernetes is a registered trademark of The Linux Foundation in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

AI Attribution

This work was primarily human-created. AI was used to make stylistic edits, such as changes to structure, wording, and clarity. AI was used to make content edits, such as changes to scope, information, and ideas. AI was prompted for its contributions, or AI assistance was enabled. AI-generated content was reviewed and approved. The following model(s) or application(s) were used: IBM Bob, Microsoft Copilot, Google Gemini

Version History

Version 1.0 - Initial Publication (June 2026)

Document Number: MD260016

Title: IBM Storage Scale Container Native: Implementation Guide for Kubernetes and Red Hat OpenShift

Authors:

- Phillip Gerrard
- Marcelo Capile
- Abhishek Jain
- Maarten Kreuger
- Alok Kushvah

- Jorge Padilla
- Tan Long Siau
- Robert Simon
- Ankita Thange

Initial Release Section Content:

- IBM Storage Scale Container Native: Implementation Guide for Kubernetes and Red Hat OpenShift
- IBM Storage Scale Container Native Architecture and Platform Considerations
- Deploying IBM Storage Scale Container Native on Kubernetes
- Deploying IBM Storage Scale Container Native on Red Hat OpenShift
- Troubleshooting and Debugging IBM Storage Scale Container Native Deployment
- Application Use Cases
- Accelerating GenAI with IBM Storage Scale Container Native
- High Availability, Data Sharing, Disaster Recovery

Publication Date: June 2026

Summary: Initial publication providing comprehensive guidance for implementing IBM Storage Scale Container Native in Kubernetes and Red Hat OpenShift environments. Covers architectural foundations, platform-specific deployment procedures for both Kubernetes and OpenShift, troubleshooting methodologies, application use cases including FileNet and PostgreSQL, GenAI acceleration with vLLM Production Stack, and disaster recovery strategies.