NetApp

Use verified architectures

BeeGFS on NetApp with E-Series Storage

NetApp
January 27, 2026

Table of Contents

Use verified architectures

Overview and requirements

Solution overview

The BeeGFS on NetApp solution combines the BeeGFS parallel file system with NetApp EF600 storage systems for a reliable, scalable, and cost-effective infrastructure that keeps pace with demanding workloads.

NVA program

The BeeGFS on NetApp solution is part of the NetApp Verified Architecture (NVA) program, which provides customers with reference configurations and sizing guidance for specific workloads and use cases. NVA solutions are thoroughly tested and designed to minimize deployment risks and to accelerate time to market.

Design Overview

The BeeGFS on NetApp solution is designed as a scalable building block architecture, configurable for a variety of demanding workloads. Whether dealing with many small files, managing substantial large file operations, or a hybrid workload, the file system can be customized to meet these needs. High availability is built into the design with the use of a two-tier hardware structure that allows independent failover at multiple hardware layers and ensures consistent performance, even during partial system degradations. The BeeGFS file system enables a high-performance and scalable environment across different Linux distributions, and presents clients with a single easily accessible storage namespace. Learn more in the architecture overview.

Use cases

The following use cases apply to the BeeGFS on NetApp solution:

- NVIDIA DGX SuperPOD systems featuring DGX's with A100, H100, H200, and B200 GPU's.
- Artificial Intelligence (AI) including machine learning (ML), deep learning (DL), large-scale natural language processing (NLP), and natural language understanding (NLU). For more information, see BeeGFS for AI: Fact versus fiction.
- High-performance computing (HPC) including applications accelerated by MPI (message passing interface) and other distributed computing techniques. For more information, see Why BeeGFS goes beyond HPC.
- Application workloads characterized by:
  - Reading or writing to files larger than 1GB
  - Reading or writing to the same file by multiple clients (10s, 100s, and 1000s)
- Multi-terabyte or multi-petabyte datasets.
- Environments that need a single storage namespace optimizable for a mix of large and small files.

Benefits

The key benefits of using BeeGFS on NetApp include:

- Availability of verified hardware designs providing full integration of hardware and software components to ensure predictable performance and reliability.

- Deployment and management using Ansible for simplicity and consistency at scale.
- Monitoring and observability provided using the E-Series Performance Analyzer and BeeGFS plugin. For more information, see Introducing a Framework to Monitor NetApp E-Series Solutions.
- High availability featuring a shared-disk architecture that provides data durability and availability.
- Support for modern workload management and orchestration using containers and Kubernetes. For more information, see Kubernetes meet BeeGFS: A tale of future-proof investment.

Architecture overview

The BeeGFS on NetApp solution includes architectural design considerations used to determine the specific equipment, cabling, and configurations required to support validated workloads.

Building block architecture

The BeeGFS file system can be deployed and scaled in different ways depending on the storage requirements. For example, use cases primarily featuring numerous small files will benefit from extra metadata performance and capacity, whereas use cases featuring fewer large files might favor more storage capacity and performance for actual file contents. These multiple considerations impact different dimensions of the parallel file system deployment, which adds complexity to designing and deploying the file system.

To address these challenges, NetApp has designed a standard building block architecture that is used to scale out each of these dimensions. Typically, BeeGFS building blocks are deployed in one of three configuration profiles:

- A single base building block, including BeeGFS management, metadata, and storage services
- A BeeGFS metadata plus storage building block
- A BeeGFS storage only building block

The only hardware change between these three options is the use of smaller drives for BeeGFS metadata. Otherwise, all configuration changes are applied through software. And with Ansible as the deployment engine, setting up the desired profile for a particular building block makes configuration tasks straightforward.

For further details, see Verified hardware design.

File system services

The BeeGFS file system includes the following main services:

- **Management service.** Registers and monitors all other services.
- **Storage service.** Stores the distributed user file contents known as data chunk files.
- **Metadata service.** Keeps track of the file system layout, directory, file attributes, and so on.
- **Client service.** Mounts the file system to access the stored data.

The following figure shows BeeGFS solution components and relationships used with NetApp E-Series systems.
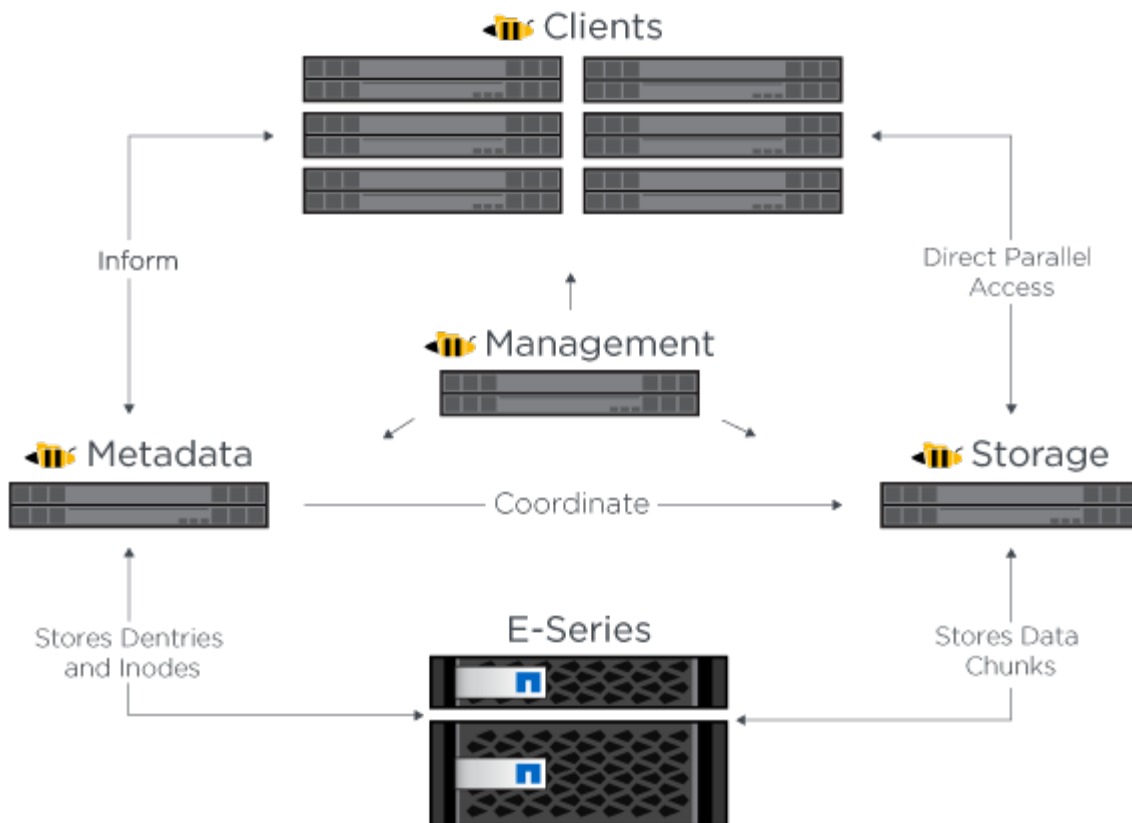
As a parallel file system, BeeGFS stripes its files over multiple server nodes to maximize read/write performance and scalability. The server nodes work together to deliver a single file system that can be simultaneously mounted and accessed by other server nodes, commonly known as *clients*. These clients can see and consume the distributed file system similarly to a local file system such as NTFS, XFS, or ext4.

The four main services run on a wide range of supported Linux distributions and communicate via any TCP/IP or RDMA-capable network, including InfiniBand (IB), Omni-Path (OPA), and RDMA over Converged Ethernet (RoCE). The BeeGFS server services (management, storage, and metadata) are user space daemons, while the client is a native kernel module (patchless). All components can be installed or updated without rebooting, and you can run any combination of services on the same node.

HA architecture

BeeGFS on NetApp expands the functionality of the BeeGFS enterprise edition by creating a fully integrated solution with NetApp hardware that enables a shared-disk high availability (HA) architecture.

> (i) While the BeeGFS community edition can be used free of charge, the enterprise edition requires purchasing a professional support subscription contract from a partner like NetApp. The enterprise edition allows use of several additional features including resiliency, quota enforcement, and storage pools.

The following figure compares the shared-nothing and shared-disk HA architectures.

Shared-Nothing Architecture vs. Shared-Disk Architecture
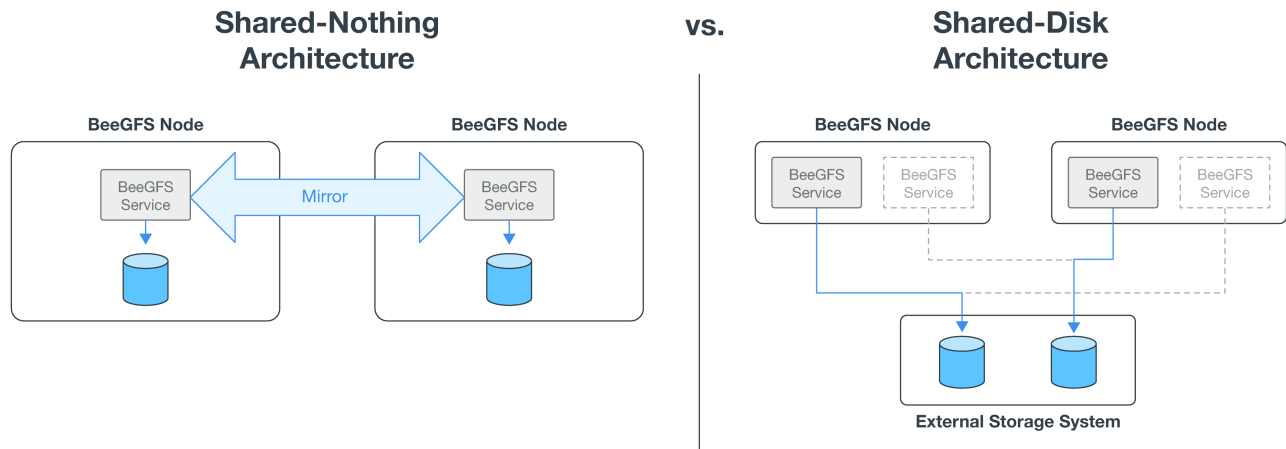
For more information, see Announcing High Availability for BeeGFS Supported by NetApp.

Verified nodes

The BeeGFS on NetApp solution has verified the nodes listed below.

Node	Hardware	Details
Block	NetApp EF600 Storage System	A high-performance, all-NVMe 2U storage array designed for demanding workloads.
File	Lenovo ThinkSystem SR665 V3 Server	A two-socket 2U server featuring PCIe 5.0, dual AMD EPYC 9124 processors. For more information about the Lenovo SR665 V3, see Lenovo's website.
	Lenovo ThinkSystem SR665 Server	A two-socket 2U server featuring PCIe 4.0, dual AMD EPYC 7003 processors. For more information about the Lenovo SR665, see Lenovo's website.

Verified hardware design

The solution's building blocks (shown in the following figure) uses the verified file node servers for the BeeGFS file layer and two EF600 storage systems as the block layer.
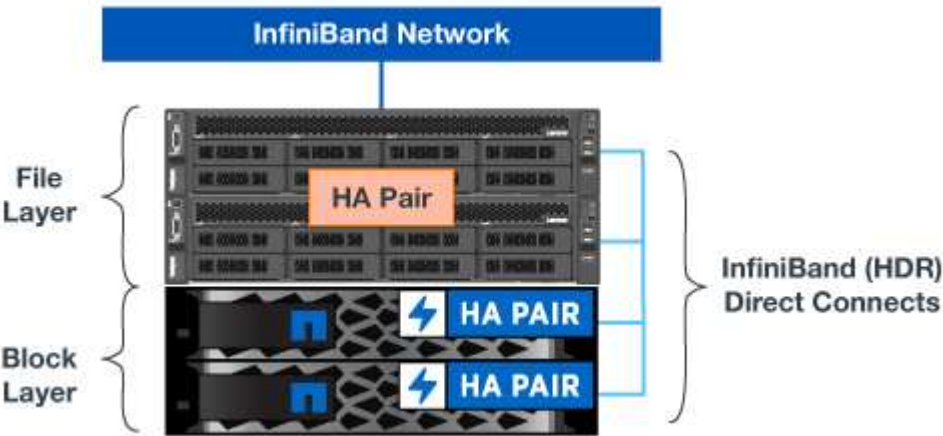
The BeeGFS on NetApp solution runs across all building blocks in the deployment. The first building block

deployed must run BeeGFS management, metadata, and storage services (known as the base building block). All subsequent building blocks can be configured through software to extend metadata and storage services, or to provide storage services exclusively. This modular approach enables scaling the file system to the needs of a workload while using the same underlying hardware platforms and building block design.

Up to five building blocks can be deployed to form a standalone Linux HA cluster. This optimizes resource management with Pacemaker and maintains efficient synchronization with Corosync. One or more of these standalone BeeGFS HA clusters are combined to create a BeeGFS file system that is accessible to clients as a single storage namespace. On the hardware side, a single 42U rack can accommodate up to five building blocks, along with two 1U InfiniBand switches for the storage/data network. See the graphic below for a visual representation.

> (i) A minimum of two building blocks are required to establish quorum in the failover cluster. A two-node cluster has limitations that might prevent a successful failover from occurring. You can configure a two-node cluster by incorporating a third device as a tiebreaker; however, this documentation does not describe that design.
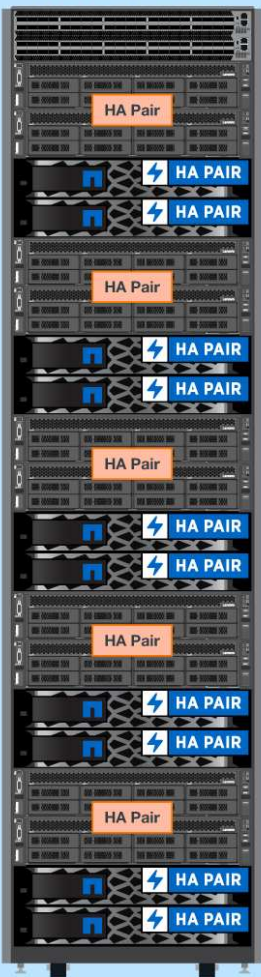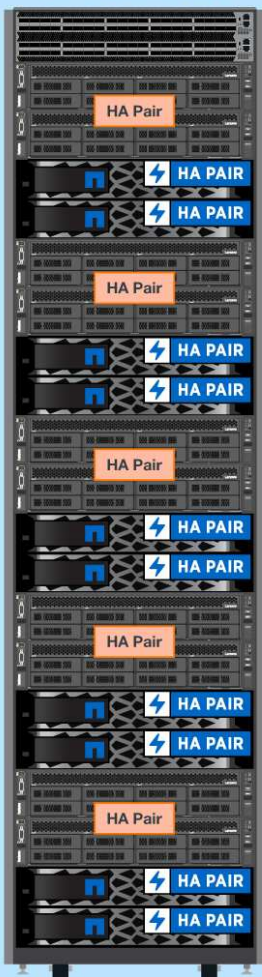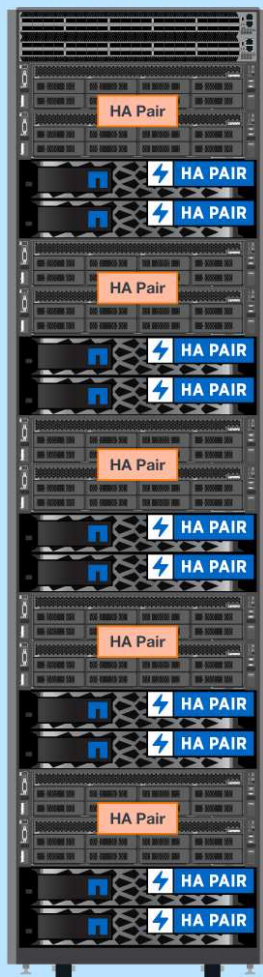
Ansible

BeeGFS on NetApp is delivered and deployed using Ansible automation, which is hosted on GitHub and Ansible Galaxy (the BeeGFS collection is available from Ansible Galaxy and NetApp's E-Series GitHub). Although Ansible is primarily tested with the hardware used to assemble the BeeGFS building blocks, you can configure it to run on virtually any x86-based server using a supported Linux distribution.

For more information, see Deploying BeeGFS with E-Series Storage.

Technical requirements

To implement the BeeGFS on NetApp solution, ensure your environment meets the technology requirements outlined in this document.

Hardware requirements

Before you begin, ensure that your hardware meets the following specifications for a single second-generation building block design of the BeeGFS on NetApp solution. The exact components for a particular deployment may vary based on customer requirements.

Quantity	Hardware component	Requirements
2	BeeGFS file nodes	Each file node should meet or exceed the specifications of the recommended file nodes to achieve expected performance. **Recommended file node options:** • **Lenovo ThinkSystem SR665 V3** ◦ **Processors:** 2x AMD EPYC 9124 16C 3.0 GHz (configured as two NUMA zones). ◦ **Memory:** 256GB (16x 16GB TruDDR5 4800MHz RDIMM-A) ◦ **PCIe Expansion:** Four PCIe Gen5 x16 slots (two per NUMA zone) ◦ **Miscellaneous:** ▪ Two drives in RAID 1 for OS (1TB 7.2K SATA or better) ▪ 1GbE port for in-band OS management ▪ 1GbE BMC with Redfish API for out-of-band server management ▪ Dual hot swap power supplies and performance fans
2	E-Series block nodes (EF600 array)	**Memory:** 256GB (128GB per controller). **Adapter:** 2-port 200Gb/HDR (NVMe/IB). **Drives:** Configured to match desired metadata and storage capacity.
8	InfiniBand host card adapters (for file nodes).	Host card adapters may vary based on the file node's server model. Recommendations for verified file nodes include: • **Lenovo ThinkSystem SR665 V3 Server:** ◦ MCX755106AS-HEAT ConnectX-7, NDR200, QSFP112, 2-port, PCIe Gen5 x16, InfiniBand Adapter
1	Storage network switch	The storage network switch must be capable of 200Gb/s InfiniBand speeds. Recommended switch models include: • **NVIDIA QM9700 Quantum 2 NDR InfiniBand switch** • **NVIDIA MQM8700 Quantum HDR InfiniBand switch**

Cabling requirements

Direct connections from block nodes to file nodes.

Quantity	Part Number	Length
8	MCP1650-H001E30 (NVIDIA Passive Copper cable, QSFP56, 200Gb/s)	1m

Connections from file nodes to the storage network switch.
Select the appropriate cable option from the following table according to your InfiniBand storage switch.
The recommended cable length is 2m; however, this may vary based on the customer's environment.

Switch model	Cable Type	Quantity	Part Number
NVIDIA QM9700	Active Fiber (including transceivers)	2	MMA4Z00-NS (multimode, IB/ETH, 800Gb/s 2x400Gb/s Twin-port OSFP)
		4	MFP7E20-Nxxx (multimode, 4-channel-to-two 2-channel splitter fiber cable)
		8	MMA1Z00-NS400 (multimode, IB/ETH, 400Gb/s Single-port QSFP-112)
	Passive Copper	2	MCP7Y40-N002 (NVIDIA passive copper splitter cable, InfiniBand 800Gb/s to 4x 200Gb/s, OSFP to 4x QSFP112)
NVIDIA MQM8700	Active Fiber	8	MFS1S00-H003E (NVIDIA Active Fiber cable, InfiniBand 200Gb/s, QSFP56)
	Passive Copper	8	MCP1650-H002E26 (NVIDIA Passive Copper cable, InfiniBand 200Gb/s, QSFP56)

Software and firmware requirements

To ensure predictable performance and reliability, releases of the BeeGFS on NetApp solution are tested with specific versions of software and firmware components. These versions are required for implementation of the solution.

File node requirements

Software	Version
Red Hat Enterprise Linux (RHEL)	RHEL 9.4 Server Physical with High Availability (2 socket). **Note:** File nodes require a valid Red Hat Enterprise Linux Server subscription and the Red Hat Enterprise Linux High Availability Add-On.
Linux Kernel	5.14.0-427.42.1.el9_4.x86_64
HCA Firmware	**ConnectX-7 HCA Firmware** FW: 28.45.1200 PXE: 3.7.0500 UEFI: 14.38.0016 **ConnectX-6 HCA Firmware** FW: 20.43.2566 PXE: 3.7.0500 UEFI: 14.37.0013

EF600 block node requirements

Software	Version
SANtricity OS	11.90R3
NVSRAM	N6000-890834-D02.dlp
Drive Firmware	Latest available for the drive models in use. See the E-Series disk firmware site.

Software deployment requirements

The following table lists the software requirements deployed automatically as part of the Ansible-based BeeGFS deployment.

Software	Version
BeeGFS	7.4.6
Corosync	3.1.8-1
Pacemaker	2.1.7-5.2
PCS	0.11.7-2
Fence Agents (redfish/apc)	4.10.0-62
InfiniBand / RDMA Drivers	MLNX_OFED_LINUX-23.10-3.2.2.1-LTS

Ansible control node requirements

The BeeGFS on NetApp solution is deployed and managed from an Ansible control node. For more information, see the Ansible documentation.

The software requirements listed in the following tables are specific to the version of the NetApp BeeGFS Ansible collection listed below.

Software	Version
Ansible	10.x
Ansible-core	>= 2.13.0
Python	3.10
Additional Python packages	Cryptography-43.0.0, netaddr-1.3.0, ipaddr-2.2.0
NetApp E-Series BeeGFS Ansible Collection	3.2.0

Review solution design

Design overview

Specific equipment, cabling, and configurations are required to support the BeeGFS on NetApp solution, which combines the BeeGFS parallel file system with the NetApp EF600

storage systems.

Learn more:

- Hardware configuration
- Software configuration
- Design verification
- Sizing guidelines
- Performance tuning

Derivative architectures with variations in design and performance:

- High Capacity Building Block

Hardware configuration

The hardware configuration for BeeGFS on NetApp includes file nodes and network cabling.

File node configuration

File nodes have two CPU sockets configured as separate NUMA zones, which include local access to an equal number of PCIe slots and memory.

InfiniBand adapters must be populated in the appropriate PCI risers or slots, so the workload is balanced over the available PCIe lanes and memory channels. You balance the workload by fully isolating work for individual BeeGFS services to a particular NUMA node. The goal is to achieve similar performance from each file node as if it were two independent single socket servers.

The following figure shows the file node NUMA configuration.
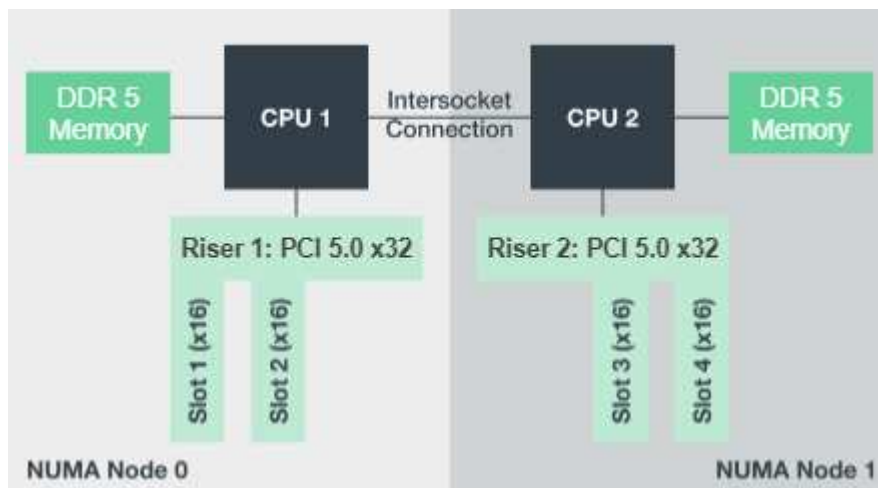
The BeeGFS processes are pinned to a particular NUMA zone to ensure that the interfaces used are in the same zone. This configuration avoids the need for remote access over the inter-socket connection. The inter-socket connection is sometimes known as the QPI or GMI2 link; even in modern processor architectures, they can be a bottleneck when using high-speed networking like HDR InfiniBand.

Network cabling configuration

Within a building block, each file node is connected to two block nodes using a total of four redundant InfiniBand connections. In addition, each file node has four redundant connections to the InfiniBand storage network.

In the following figure, notice that:

- All file node ports outlined in green are used to connect to the storage fabric; all other file node ports are the direct connects to the block nodes.
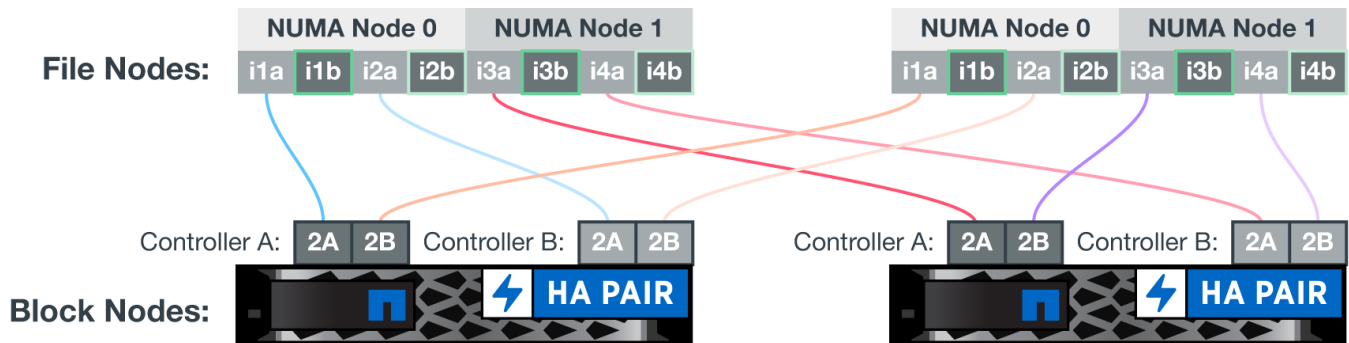- Two InfiniBand ports in a specific NUMA zone connect to the A and B controllers of the same block node.
- Ports in NUMA node 0 always connect to the first block node.
- Ports in NUMA node 1 connect to the second block node.

> ⓘ When using splitter cables to connect the storage switch to file nodes, one cable should branch out and connect to the ports outlined in light green. Another cable should branch out and connect to the ports outlined in dark green.
> Additionally, for storage networks with redundant switches, ports outlined in light green should connect to one switch, while ports in dark green should connect to another switch.

The cabling configuration depicted in the figure allows each BeeGFS service to:

- Run in the same NUMA zone regardless of which file node is running the BeeGFS service.
- Have secondary optimal paths to the front-end storage network and to the back-end block nodes regardless of where a failure occurs.
- Minimize performance effects if a file node or controller in a block node requires maintenance.

Cabling to leverage bandwidth

To leverage the full PCIe bidirectional bandwidth, make sure one port on each InfiniBand adapter connects to the storage fabric, and the other port connects to a block node.

The following figure shows the cabling design used to leverage the full PCIe bidirectional bandwidth.
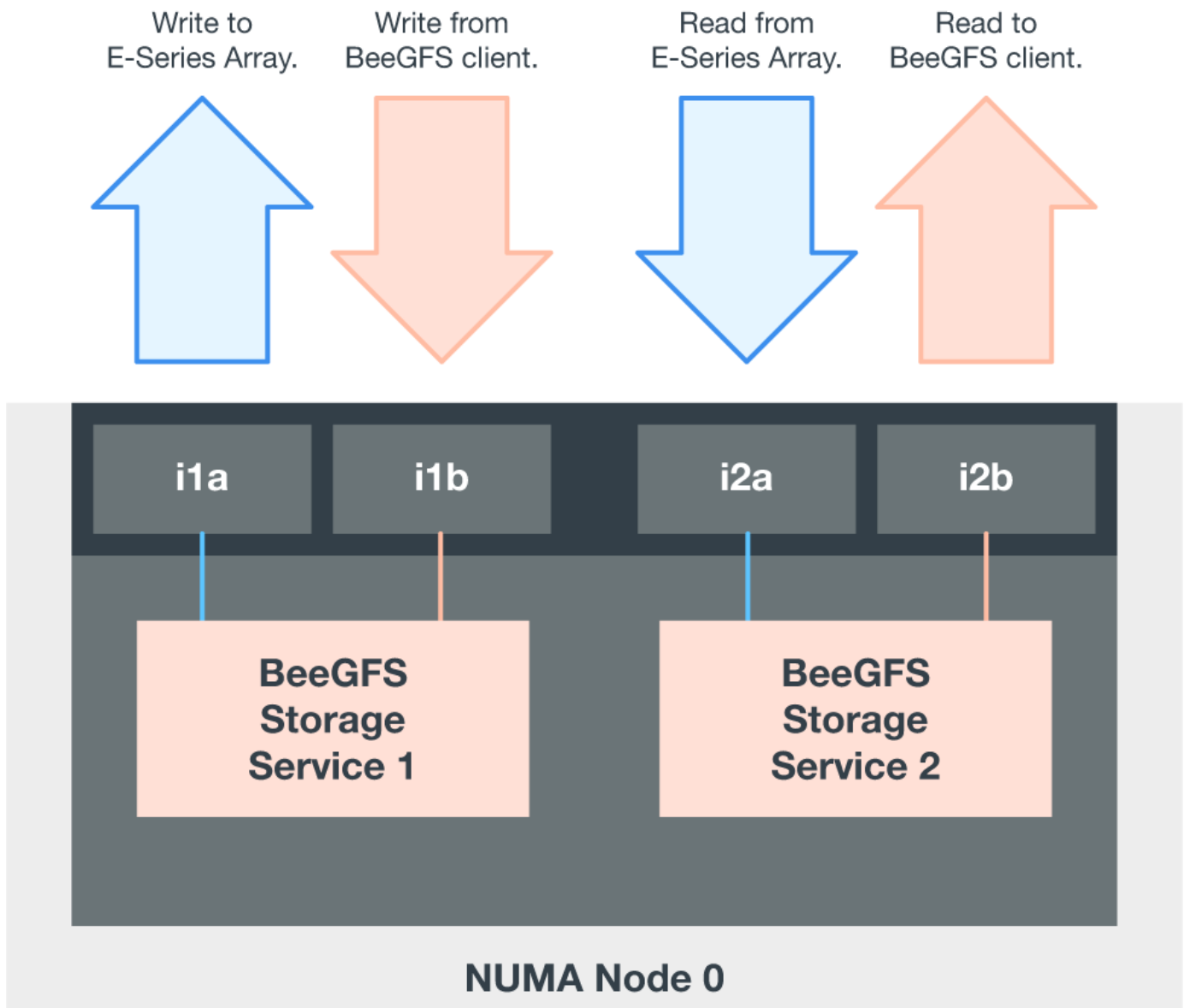
For each BeeGFS service, use the same adapter to connect the preferred port used for client traffic with the path to the block nodes controller that is the primary owner of that services volumes. For more information, see Software configuration.

Software configuration

The software configuration for BeeGFS on NetApp includes BeeGFS network components, EF600 block nodes, BeeGFS file nodes, resource groups, and BeeGFS services.

BeeGFS network configuration

The BeeGFS network configuration consists of the following components.

- **Floating IPs**
  Floating IPs are a kind of virtual IP address that can be dynamically routed to any server in the same network. Multiple servers can own the same Floating IP address, but it can only be active on one server at any given time.

Each BeeGFS server service has its own IP address that can move between file nodes depending on the run location of the BeeGFS server service. This floating IP configuration allows each service to fail over independently to the other file node. The client simply needs to know the IP address for a particular BeeGFS service; it does not need to know which file node is currently running that service.

- **BeeGFS server multi-homing configuration**
  To increase the density of the solution, each file node has multiple storage interfaces with IPs configured in the same IP subnet.

  Additional configuration is required to make sure that this configuration works as expected with the Linux networking stack, because by default, requests to one interface can be responded to on a different interface if their IPs are in the same subnet. In addition to other drawbacks, this default behavior makes it impossible to properly establish or maintain RDMA connections.

  The Ansible-based deployment handles tightening of the reverse path (RP) and address resolution protocol (ARP) behavior, along with ensuring when floating IPs are started and stopped; corresponding IP routes and rules are dynamically created to allow the multihomed network configuration to work properly.

- **BeeGFS client multi-rail configuration**
  Multi-rail refers to the ability of an application to use multiple independent network connections, or "rails", to increase performance.

  BeeGFS implements multi-rail support to allow the use of multiple IB interfaces in a single IPoIB subnet. This capability enables features such as dynamic load balancing across RDMA NICs, optimizing the use of network resources. It also integrates with NVIDIA GPUDirect Storage (GDS), which offers increased system bandwidth and decreases latency and utilization on the client's CPU.

  This documentation provides instructions for single IPoIB subnet configurations. Dual IPoIB subnet configurations are supported but do not provide the same advantages as single subnet configurations.

  The following figure shows the balancing of traffic across multiple BeeGFS client interfaces.
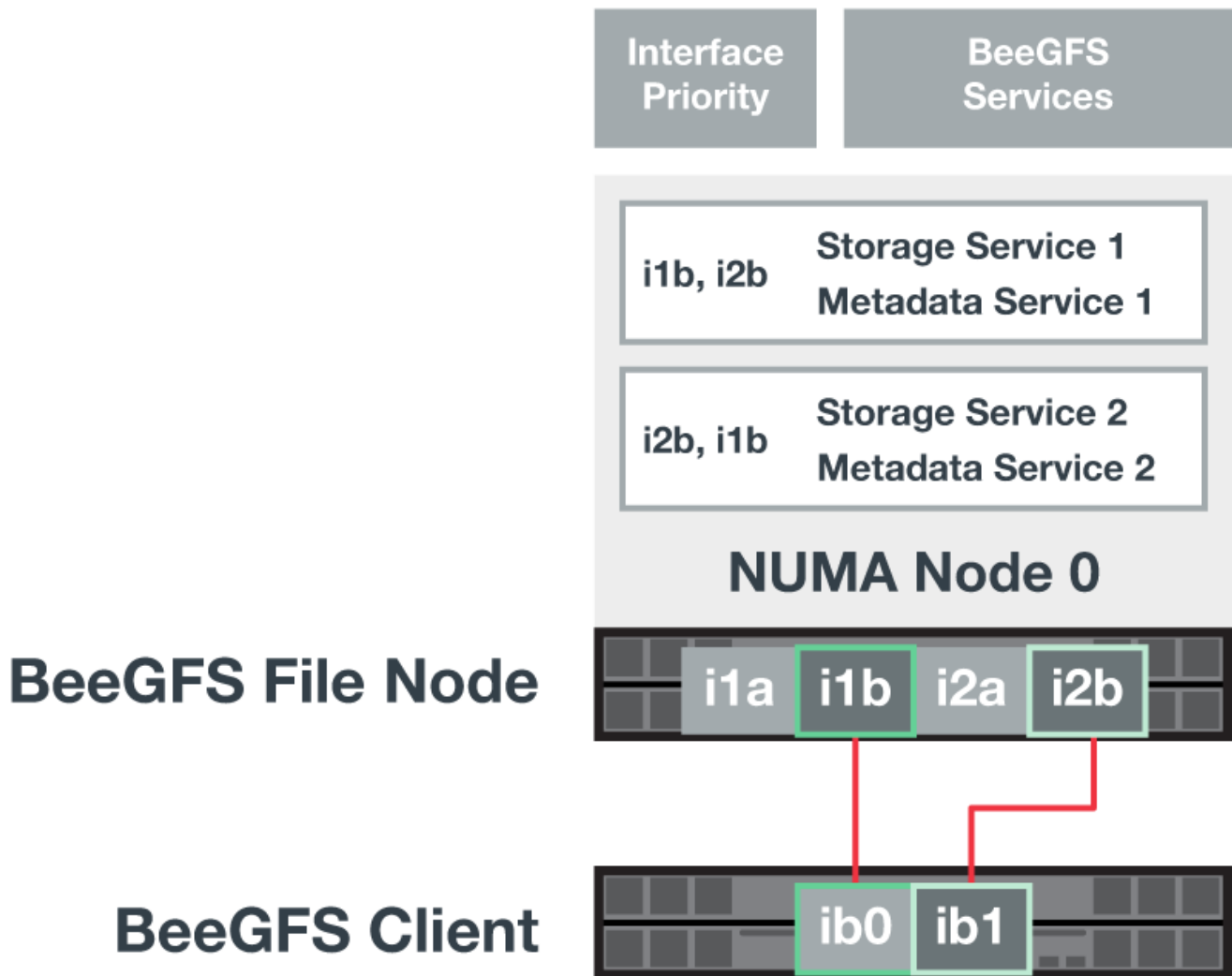
Because each file in BeeGFS is typically striped across multiple storage services, the multi-rail configuration allows the client to achieve more throughput than is possible with a single InfiniBand port. For example, the following code sample shows a common file-striping configuration that allows the client to balance traffic across both interfaces:

+

```
root@beegfs01:/mnt/beegfs# beegfs-ctl --getentryinfo myfile
Entry type: file
EntryID: 11D-624759A9-65
Metadata node: meta_01_tgt_0101 [ID: 101]
Stripe pattern details:
+ Type: RAID0
+ Chunksize: 1M
+ Number of storage targets: desired: 4; actual: 4
+ Storage targets:
  + 101 @ stor_01_tgt_0101 [ID: 101]
  + 102 @ stor_01_tgt_0101 [ID: 101]
  + 201 @ stor_02_tgt_0201 [ID: 201]
  + 202 @ stor_02_tgt_0201 [ID: 201]
```

EF600 block node configuration

Block nodes are comprised of two active/active RAID controllers with shared access to the same set of drives. Typically, each controller owns half the volumes configured on the system, but can take over for the other controller as needed.

Multipathing software on the file nodes determines the active and optimized path to each volume and automatically moves to the alternate path in the event of a cable, adapter, or controller failure.

The following diagram shows the controller layout in EF600 block nodes.

To facilitate the shared-disk HA solution, volumes are mapped to both file nodes so that they can take over for each other as needed. The following diagram shows an example of how the BeeGFS service and preferred volume ownership is configured for maximum performance. The interface to the left of each BeeGFS service indicates the preferred interface that the clients and other services use to contact it.
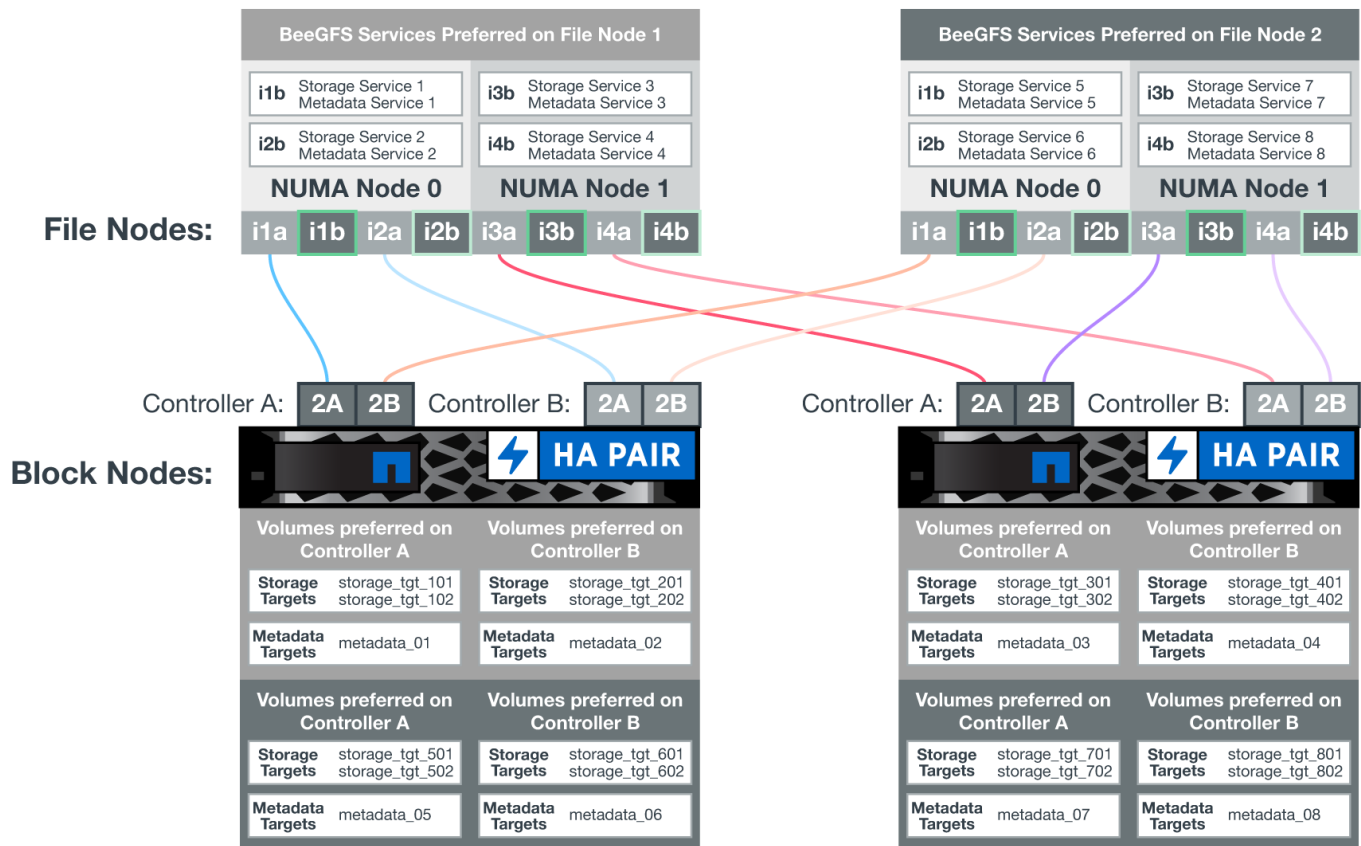
In the previous example, clients and server services prefer to communicate with storage service 1 using interface i1b. Storage service 1 uses interface i1a as the preferred path to communicate with its volumes (storage_tgt_101, 102) on controller A of the first block node. This configuration makes use of the full bidirectional PCIe bandwidth available to the InfiniBand adapter and achieves better performance from a dual-port HDR InfiniBand adapter than would otherwise be possible with PCIe 4.0.

BeeGFS file node configuration

The BeeGFS file nodes are configured into a High-Availability (HA) cluster to facilitate failover of BeeGFS services between multiple file nodes.

The HA cluster design is based on two widely used Linux HA projects: Corosync for cluster membership and Pacemaker for cluster resource management. For more information, see Red Hat training for high-availability add-ons.

NetApp authored and extended several open cluster framework (OCF) resource agents to allow the cluster to intelligently start and monitor the BeeGFS resources.

BeeGFS HA clusters

Typically, when you start a BeeGFS service (with or without HA), a few resources must be in place:

- IP addresses where the service is reachable, typically configured by Network Manager.
- Underlying file systems used as the targets for BeeGFS to store data.

  These are typically defined in `/etc/fstab` and mounted by Systemd.

- A Systemd service responsible for starting BeeGFS processes when the other resources are ready.

Without additional software, these resources start only on a single file node. Therefore, if the file node goes offline, a portion of the BeeGFS file system is inaccessible.

Because multiple nodes can start each BeeGFS service, Pacemaker must make sure each service and dependent resources are only running on one node at a time. For example, if two nodes try to start the same BeeGFS service, there is a risk of data corruption if they both try to write to the same files on the underlying target. To avoid this scenario, Pacemaker relies on Corosync to reliably keep the state of the overall cluster in sync across all nodes and establish quorum.

If a failure occurs in the cluster, Pacemaker reacts and restarts BeeGFS resources on another node. In some scenarios, Pacemaker might not be able to communicate with the original faulty node to confirm the resources are stopped. To verify that the node is down before restarting BeeGFS resources elsewhere, Pacemaker fences off the faulty node, ideally by removing power.

Many open-source fencing agents are available that enable Pacemaker to fence a node with a power distribution unit (PDU) or by using the server baseboard management controller (BMC) with APIs such as Redfish.

When BeeGFS is running in an HA cluster, all BeeGFS services and underlying resources are managed by Pacemaker in resource groups. Each BeeGFS service and the resources it depends on, are configured into a resource group, which ensures resources are started and stopped in the correct order and collocated on the same node.

For each BeeGFS resource group, Pacemaker runs a custom BeeGFS monitoring resource that is responsible for detecting failure conditions and intelligently triggering failovers when a BeeGFS service is no longer accessible on a particular node.

The following figure shows the Pacemaker-controlled BeeGFS services and dependencies.
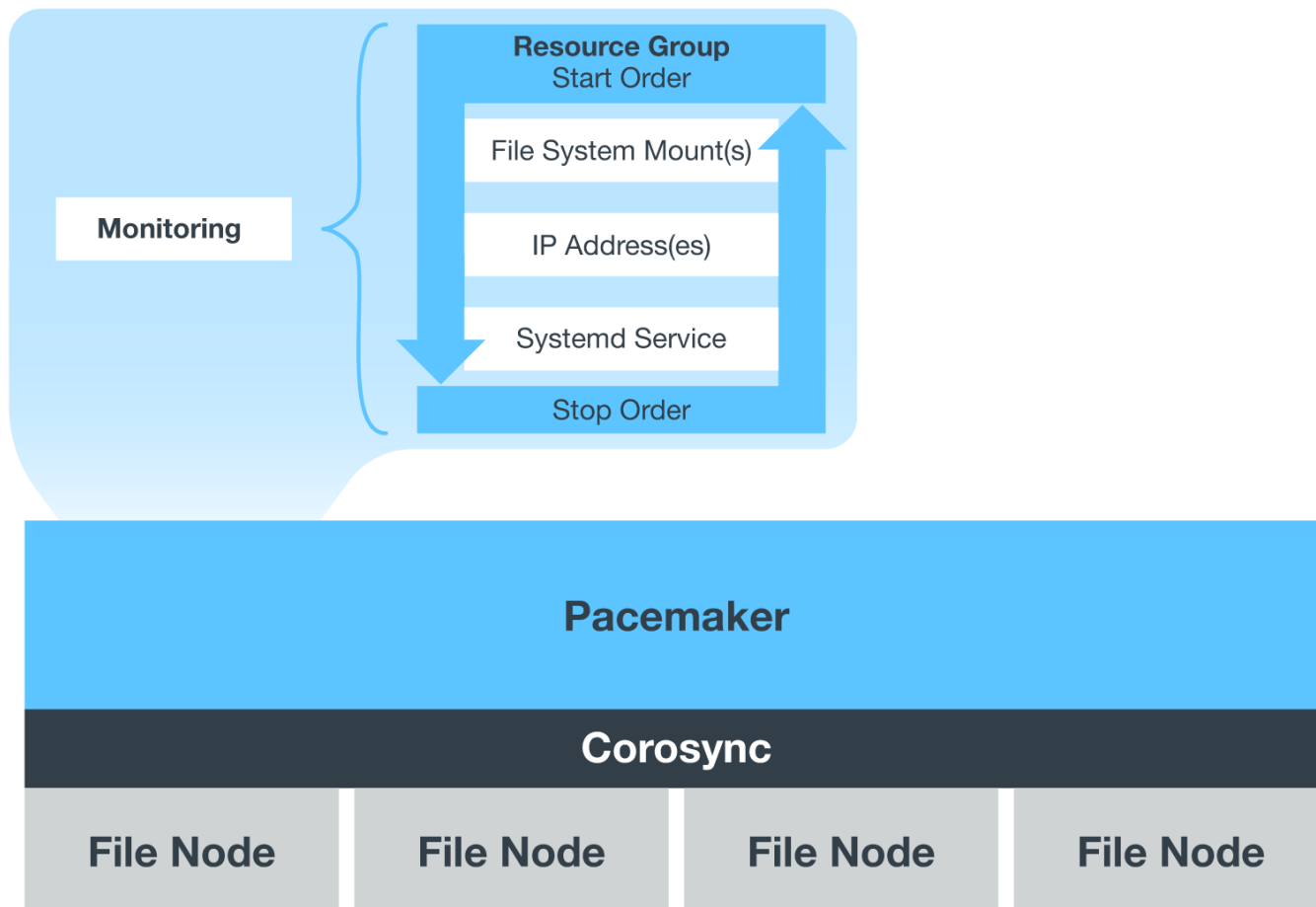
ⓘ So that multiple BeeGFS services of the same type are started on the same node, Pacemaker is configured to start BeeGFS services using the Multi Mode configuration method. For more information, see the BeeGFS documentation on Multi Mode.

Because BeeGFS services must be able to start on multiple nodes, the configuration file for each service (normally located at `/etc/beegfs`) is stored on one of the E-Series volumes used as the BeeGFS target for that service. This makes the configuration along with the data for a particular BeeGFS service accessible to all nodes that might need to run the service.

```
# tree stor_01_tgt_0101/ -L 2
stor_01_tgt_0101/
├── data
│   ├── benchmark
│   ├── buddymir
│   ├── chunks
│   ├── format.conf
│   ├── lock.pid
│   ├── nodeID
│   ├── nodeNumID
│   ├── originalNodeID
│   ├── targetID
│   └── targetNumID
└── storage_config
    ├── beegfs-storage.conf
    ├── connInterfacesFile.conf
    └── connNetFilterFile.conf
```

Design verification

The second-generation design for the BeeGFS on NetApp solution was verified using three building block configuration profiles.

The configuration profiles include the following:

- A single base building block, including BeeGFS management, metadata, and storage services.
- A BeeGFS metadata plus a storage building block.
- A BeeGFS storage-only building block.

The building blocks were attached to two NVIDIA Quantum InfiniBand (MQM8700) switches. Ten BeeGFS clients were also attached to the InfiniBand switches and used to run synthetic benchmark utilities.

The following figure shows the BeeGFS configuration used to validate the BeeGFS on NetApp solution.

InfiniBand Storage Network
(BeeGFS Client/Server Traffic)

BeeGFS Management, Metadata, and Storage Building Block

BeeGFS Metadata, and Storage Building Block

BeeGFS Storage Only Building Block

BeeGFS CPU Clients
(used for validation)

BeeGFS GPU Client
(used for validation)
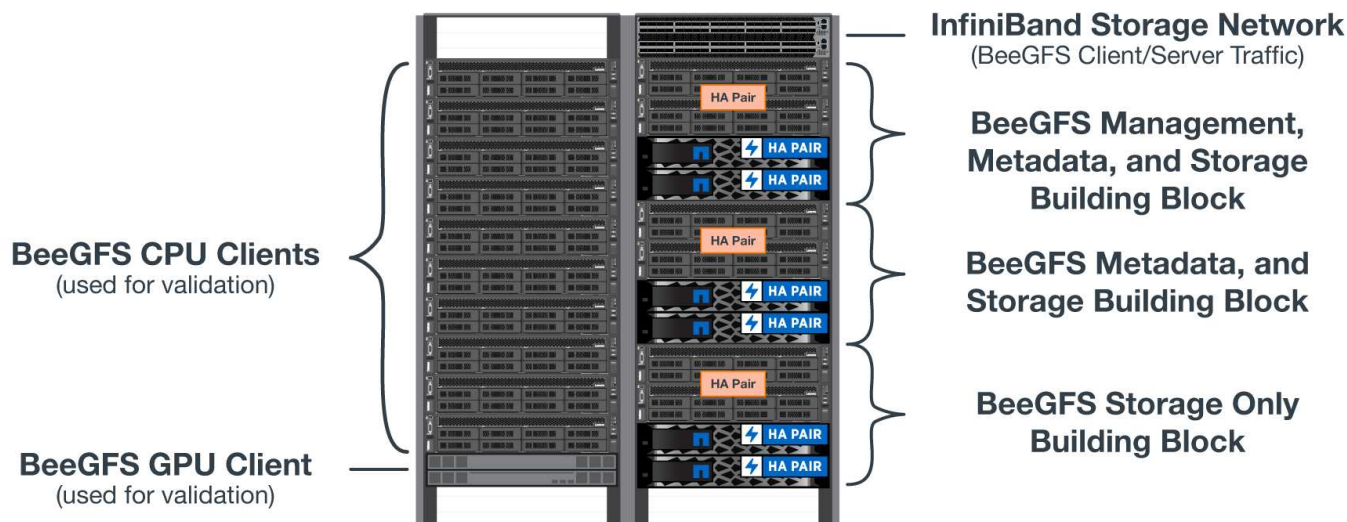
BeeGFS file striping

A benefit of parallel file systems is the ability to stripe individual files across multiple storage targets, which could represent volumes on the same or different underlying storage systems.

In BeeGFS, you can configure striping on a per-directory and per-file basis to control the number of targets used for each file and to control the chunksize (or block size) used for each file stripe. This configuration allows the file system to support different types of workloads and I/O profiles without the need for reconfiguring or restarting services. You can apply stripe settings using the `beegfs-ctl` command line tool or with applications that use the striping API. For more information, see the BeeGFS documentation for Striping and Striping API.

To achieve the best performance, stripe patterns were adjusted throughout testing, and the parameters used for each test are noted.

IOR bandwidth tests: Multiple clients

The IOR bandwidth tests used OpenMPI to run parallel jobs of the synthetic I/O generator tool IOR (available from HPC GitHub) across all 10 client nodes to one or more BeeGFS building blocks. Unless otherwise noted:

- All tests used direct I/O with a 1MiB transfer size.
- BeeGFS file striping was set to a 1MB chunksize and one target per file.

The following parameters were used for IOR with the segment count adjusted to keep the aggregate file size to 5TiB for one building block and 40TiB for three building blocks.

```
mpirun --allow-run-as-root --mca btl tcp -np 48 -map-by node -hostfile
10xnodes ior -b 1024k --posix.odirect -e -t 1024k -s 54613 -z -C -F -E -k
```

One BeeGFS base (management, metadata, and storage) building block

The following figure shows the IOR test results with a single BeeGFS base (management, metadata, and storage) building block.

70.00 GiB/s
60.00 GiB/s
50.00 GiB/s
40.00 GiB/s
30.00 GiB/s
20.00 GiB/s
10.00 GiB/s
0.00 GiB/s

	16 processes	48 processes	160 processes
	1 MiB		
	Mgt+Meta+Storage		
■ READ - Direct-IO - sequential	17.55 GiB/s	40.75 GiB/s	63.61 GiB/s
■ WRITE - Direct-IO - sequential	15.10 GiB/s	18.55 GiB/s	21.41 GiB/s

BeeGFS metadata + storage building block

The following figure shows the IOR test results with a single BeeGFS metadata + storage building block.
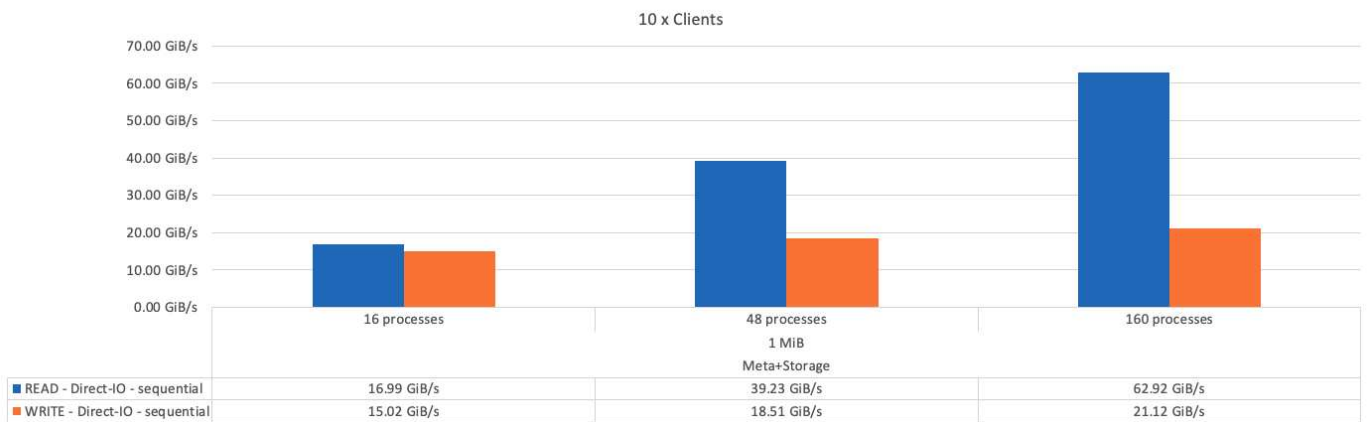
10 x Clients

70.00 GiB/s
60.00 GiB/s
50.00 GiB/s
40.00 GiB/s
30.00 GiB/s
20.00 GiB/s
10.00 GiB/s
0.00 GiB/s

	16 processes	48 processes	160 processes
	1 MiB		
	Meta+Storage		
■ READ - Direct-IO - sequential	16.99 GiB/s	39.23 GiB/s	62.92 GiB/s
■ WRITE - Direct-IO - sequential	15.02 GiB/s	18.51 GiB/s	21.12 GiB/s

BeeGFS storage-only building block

The following figure shows the IOR test results with a single BeeGFS storage-only building block.

10 x Clients

70.00 GiB/s
60.00 GiB/s
50.00 GiB/s
40.00 GiB/s
30.00 GiB/s
20.00 GiB/s
10.00 GiB/s
0.00 GiB/s

	16 processes	48 processes	160 processes
	1 MiB		
	Storage Only		
■ READ - Direct-IO - sequential	17.32 GiB/s	41.35 GiB/s	65.74 GiB/s
■ WRITE - Direct-IO - sequential	15.02 GiB/s	18.59 GiB/s	21.07 GiB/s

Three BeeGFS building blocks

The following figure shows the IOR test results with three BeeGFS building blocks.

10 x Clients

	48 processes	160 processes	480 processes	960 processes
■ READ - Direct-IO - sequential	50.93 GiB/s	128.79 GiB/s	191.47 GiB/s	211.00 GiB/s
■ WRITE - Direct-IO - sequential	45.04 GiB/s	57.12 GiB/s	63.79 GiB/s	65.92 GiB/s

1 MiB
3x Building Blocks 7.2.6
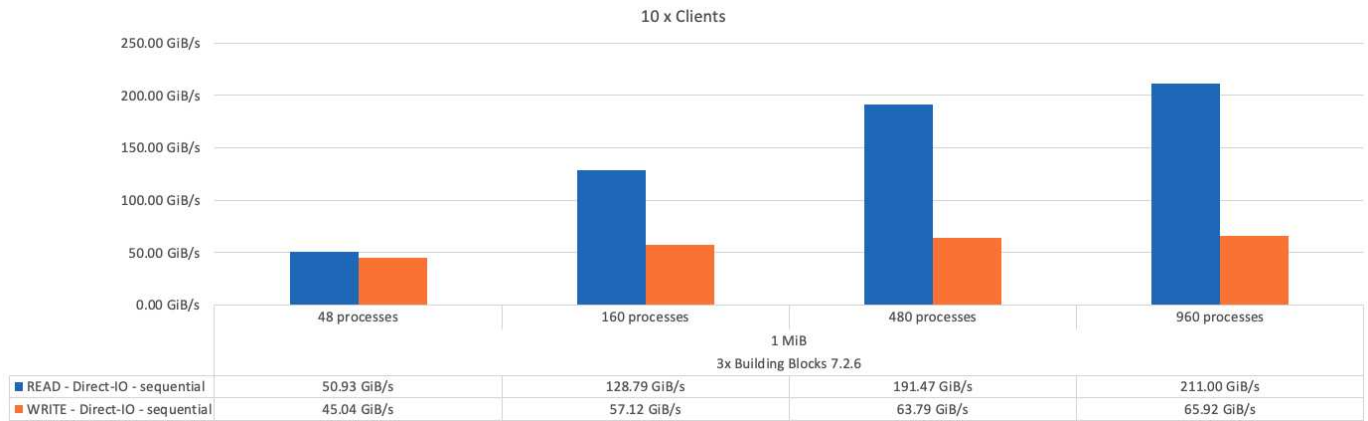
As expected, the performance difference between the base building block and the subsequent metadata + storage building block is negligible. Comparing the metadata + storage building block and a storage-only building block shows a slight increase in read performance due to the additional drives used as storage targets. However, there is no significant difference in write performance. To achieve higher performance, you can add multiple building blocks together to scale performance in a linear fashion.
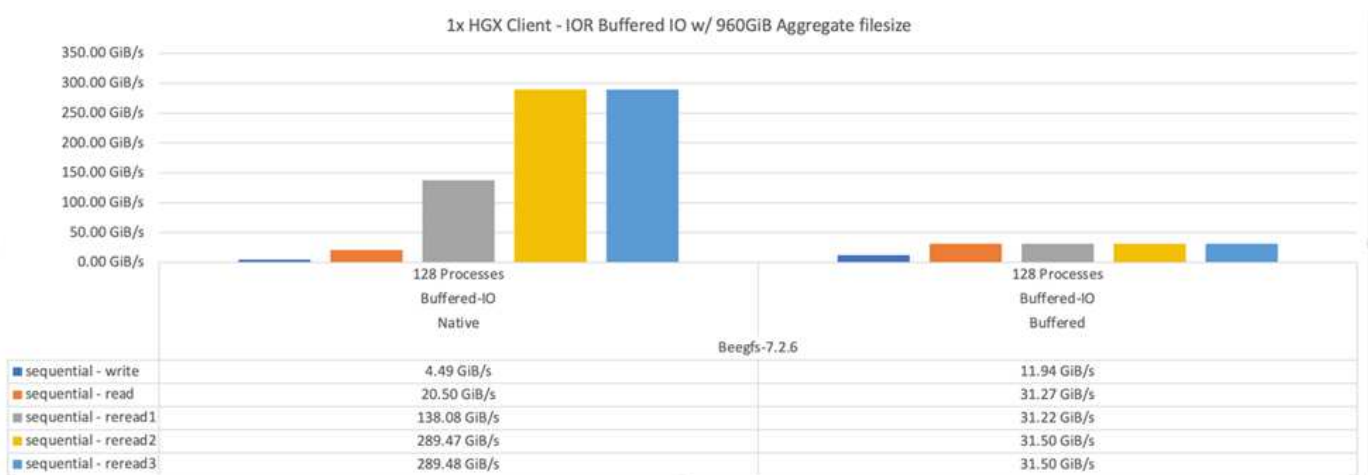
IOR bandwidth tests: Single client

The IOR bandwidth test used OpenMPI to run multiple IOR processes using a single high-performance GPU server to explore the performance achievable to a single client.

This test also compares the reread behavior and performance of BeeGFS when the client is configured to use the Linux kernel page-cache (`tuneFileCacheType = native`) versus the default `buffered` setting.

The native caching mode uses the Linux kernel page-cache on the client, allowing reread operations to come from local memory instead of being retransmitted over the network.

The following diagram shows the IOR test results with three BeeGFS building blocks and a single client.

1x HGX Client - IOR Buffered IO w/ 960GiB Aggregate filesize

	128 Processes Buffered-IO Native	128 Processes Buffered-IO Buffered
■ sequential - write	4.49 GiB/s	11.94 GiB/s
■ sequential - read	20.50 GiB/s	31.27 GiB/s
■ sequential - reread1	138.08 GiB/s	31.22 GiB/s
■ sequential - reread2	289.47 GiB/s	31.50 GiB/s
■ sequential - reread3	289.48 GiB/s	31.50 GiB/s

Beegfs-7.2.6

ⓘ   BeeGFS striping for these tests was set to a 1MB chunksize with eight targets per file.

Although write and initial read performance is higher using the default buffered mode, for workloads that reread the same data multiple times, a significant performance boost is seen with the native caching mode. This improved reread performance is important for workloads like deep learning that reread the same dataset multiple times across many epochs.

Metadata performance test

The Metadata performance tests used the MDTest tool (included as part of IOR) to measure the metadata performance of BeeGFS. The tests utilized OpenMPI to run parallel jobs across all ten client nodes.
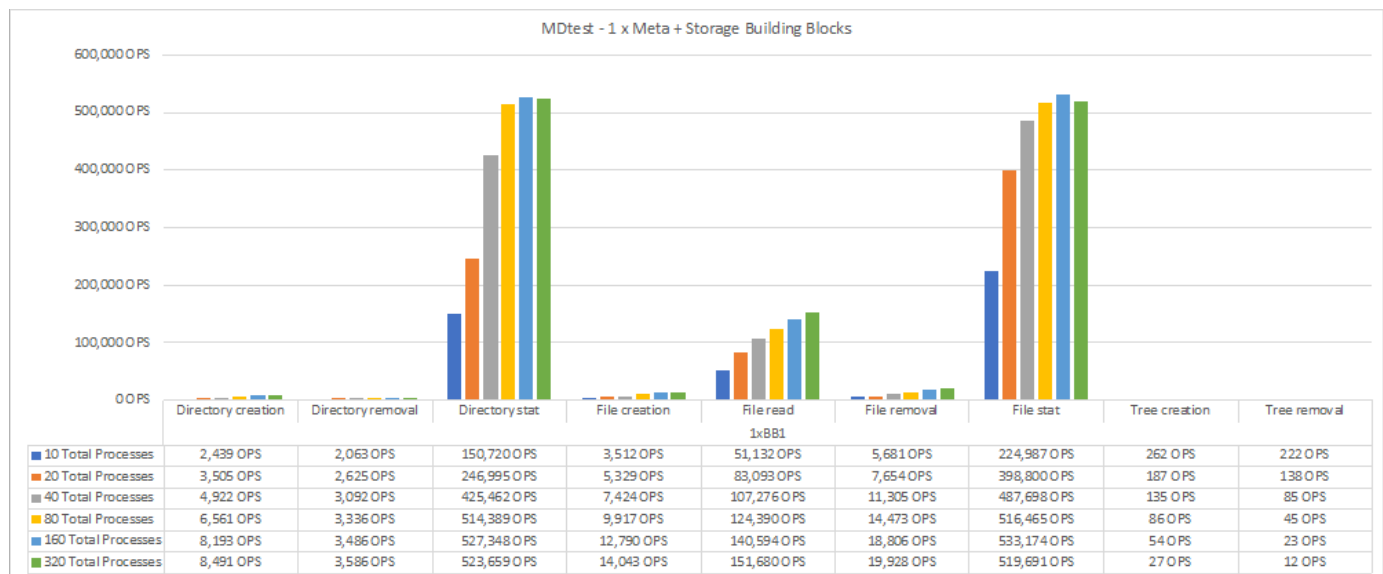
The following parameters were used to run the benchmark test with the total number of processes scaled from 10 to 320 in step of 2x and with a file size of 4k.

```
mpirun -h 10xnodes –map-by node np $processes mdtest -e 4k -w 4k -i 3 -I
16 -z 3 -b 8 -u
```

Metadata performance was measured first with one then two metadata + storage building blocks to show how performance scales by adding additional building blocks.
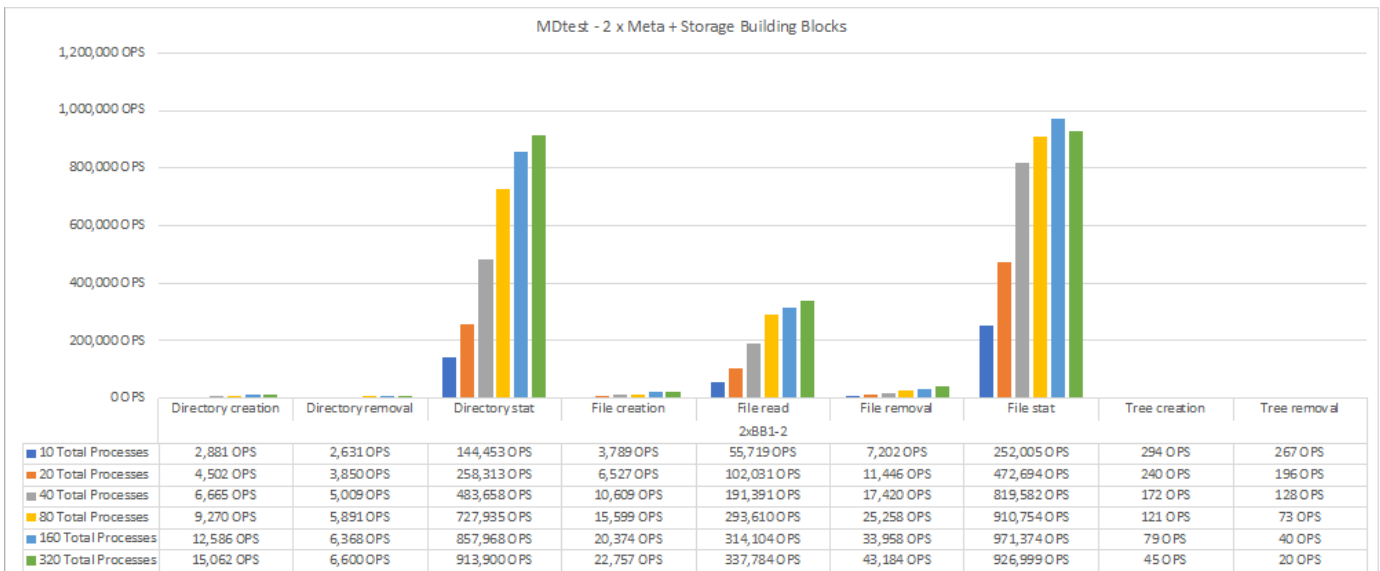
One BeeGFS metadata + storage building block

The following diagram shows the MDTest results with one BeeGFS metadata + storage building blocks.

MDtest - 1 x Meta + Storage Building Blocks

	Directory creation	Directory removal	Directory stat	File creation	File read	File removal	File stat	Tree creation	Tree removal
10 Total Processes	2,439 OPS	2,063 OPS	150,720 OPS	3,512 OPS	51,132 OPS	5,681 OPS	224,987 OPS	262 OPS	222 OPS
20 Total Processes	3,505 OPS	2,625 OPS	246,995 OPS	5,329 OPS	83,093 OPS	7,654 OPS	398,800 OPS	187 OPS	138 OPS
40 Total Processes	4,922 OPS	3,092 OPS	425,462 OPS	7,424 OPS	107,276 OPS	11,305 OPS	487,698 OPS	135 OPS	85 OPS
80 Total Processes	6,561 OPS	3,336 OPS	514,389 OPS	9,917 OPS	124,390 OPS	14,473 OPS	516,465 OPS	86 OPS	45 OPS
160 Total Processes	8,193 OPS	3,486 OPS	527,348 OPS	12,790 OPS	140,594 OPS	18,806 OPS	533,174 OPS	54 OPS	23 OPS
320 Total Processes	8,491 OPS	3,586 OPS	523,659 OPS	14,043 OPS	151,680 OPS	19,928 OPS	519,691 OPS	27 OPS	12 OPS

Two BeeGFS metadata + storage building blocks

The following diagram shows the MDTest results with two BeeGFS metadata + storage building blocks.

MDtest - 2 x Meta + Storage Building Blocks

	Directory creation	Directory removal	Directory stat	File creation	File read 2xBB1-2	File removal	File stat	Tree creation	Tree removal
10 Total Processes	2,881 OPS	2,631 OPS	144,453 OPS	3,789 OPS	55,719 OPS	7,202 OPS	252,005 OPS	294 OPS	267 OPS
20 Total Processes	4,502 OPS	3,850 OPS	258,313 OPS	6,527 OPS	102,031 OPS	11,446 OPS	472,694 OPS	240 OPS	196 OPS
40 Total Processes	6,665 OPS	5,009 OPS	483,658 OPS	10,609 OPS	191,391 OPS	17,420 OPS	819,582 OPS	172 OPS	128 OPS
80 Total Processes	9,270 OPS	5,891 OPS	727,935 OPS	15,599 OPS	293,610 OPS	25,258 OPS	910,754 OPS	121 OPS	73 OPS
160 Total Processes	12,586 OPS	6,368 OPS	857,968 OPS	20,374 OPS	314,104 OPS	33,958 OPS	971,374 OPS	79 OPS	40 OPS
320 Total Processes	15,062 OPS	6,600 OPS	913,900 OPS	22,757 OPS	337,784 OPS	43,184 OPS	926,999 OPS	45 OPS	20 OPS

Functional validation

As part of validating this architecture, NetApp executed several functional tests including the following:

- Failing a single client InfiniBand port by disabling the switch port.
- Failing a single server InfiniBand port by disabling the switch port.
- Triggering an immediate server power off using the BMC.
- Gracefully placing a node in standby and failing over service to another node.
- Gracefully placing a node back online and failing back services to the original node.
- Powering off one of the InfiniBand switches using the PDU. All tests were performed while stress testing was in progress with the `sysSessionChecksEnabled: false` parameter set on the BeeGFS clients. No errors or disruption to I/O was observed.

> (i) There is a known issue (see the Changelog) when BeeGFS client/server RDMA connections are disrupted unexpectedly, either through loss of the primary interface (as defined in `connInterfacesFile`) or a BeeGFS server failing; active client I/O can hang for up to ten minutes before resuming. This issue does not occur when BeeGFS nodes are gracefully placed in and out of standby for planned maintenance or if TCP is in use.

NVIDIA DGX SuperPOD and BasePOD validation

NetApp validated a storage solution for NVIDIAs DGX A100 SuperPOD using a similar BeeGFS file system consisting of three building blocks with the metadata plus storage configuration profile applied. The qualification effort involved testing the solution described by this NVA with twenty DGX A100 GPU servers running a variety of storage, machine learning, and deep learning benchmarks. Building on the validation established with NVIDIA's DGX A100 SuperPOD, the BeeGFS on NetApp solution has been approved for DGX SuperPOD H100, H200, and B200 systems. This extension is based on meeting the previously established benchmarks and system requirements as validated with the NVIDIA DGX A100.

For more information, see NVIDIA DGX SuperPOD with NetApp and NVIDIA DGX BasePOD.

Sizing guidelines

The BeeGFS solution includes recommendations for performance and capacity sizing that were based on verification tests.

The objective with a building-block architecture is to create a solution that is simple to size by adding multiple building blocks to meet the requirements for a particular BeeGFS system. Using the guidelines below, you can estimate the quantity and types of BeeGFS building blocks that are needed to meet the requirements of your environment.

Keep in mind that these estimates are best-case performance. Synthetic benchmarking applications are written and utilized to optimize the use of underlying file systems in ways that real-world applications might not.

Performance sizing

The following table provides recommended performance sizing.

Configuration profile	1MiB reads	1MiB writes
Metadata + storage	62GiBps	21GiBps
Storage only	64GiBps	21GiBps

Metadata capacity sizing estimates are based on the "rule of thumb" that 500GB of capacity is sufficient for roughly 150 million files in BeeGFS. (For more information, see the BeeGFS documentation for System Requirements.)

The use of features like access control lists and the number of directories and files per directory also affect how quickly metadata space is consumed. Storage capacity estimates do account for usable drive capacity along with RAID 6 and XFS overhead.

Capacity sizing for metadata + storage building blocks

The following table provides recommended capacity sizing for metadata plus storage building blocks.

Drive size (2+2 RAID 1) metadata volume groups	Metadata capacity (number of files)	Drive size (8+2 RAID 6) storage volume groups	Storage capacity (file content)
1.92TB	1,938,577,200	1.92TB	51.77TB
3.84TB	3,880,388,400	3.84TB	103.55TB
7.68TB	8,125,278,000	7.68TB	216.74TB
15.3TB	17,269,854,000	15.3TB	460.60TB

> ⓘ When sizing metadata plus storage building blocks, you can reduce costs by using smaller drives for metadata volume groups versus storage volume groups.

Capacity sizing for storage-only building blocks

The following table provides rule-of-thumb capacity sizing for storage-only building blocks.

Drive size (10+2 RAID 6) storage volume groups	Storage capacity (file content)
1.92TB	59.89TB
3.84TB	119.80TB
7.68TB	251.89TB
15.3TB	538.55TB

> ⓘ The performance and capacity overhead of including the management service in the base (first) building block are minimal, unless global file locking is enabled.

Performance tuning

The BeeGFS solution includes recommendations for performance tuning that were based on verification tests.

Although BeeGFS provides reasonable performance out of the box, NetApp has developed a set of recommended tuning parameters to maximize performance. These parameters take into account the capabilities of the underlying E-Series block nodes and any special requirements needed to run BeeGFS in a shared-disk HA architecture.

Performance tuning for file nodes

The available tuning parameters that you can configure include the following:

1. **System settings in the UEFI/BIOS of file nodes.**
   To maximize performance, we recommend configuring the system settings on the server model you use as your file nodes. You configure the system settings when you set up your file nodes by using either the system setup (UEFI/BIOS) or the Redfish APIs provided by the baseboard management controller (BMC).

   The system settings vary depending on the server model you use as your file node. The settings must be manually configured based on the server model in use. To learn how to configure the system settings for the validated Lenovo SR665 V3 file nodes, see Tune file node system settings for performance.

2. **Default settings for required configuration parameters.**
   The required configuration parameters affect how BeeGFS services are configured and how E-Series volumes (block devices) are formatted and mounted by Pacemaker. These required configuration parameters include the following:

   - BeeGFS Service configuration parameters

     You can override the default settings for the configuration parameters as needed. For the parameters that you can adjust for your specific workloads or use cases, see the BeeGFS service configuration parameters.

   - Volume formatting and mounting parameters are set to recommended defaults, and should only be adjusted for advanced use cases. The default values will do the following:

     - Optimize initial volume formatting based on the target type (such as management, metadata, or storage), along with the RAID configuration and segment size of the underlying volume.

     - Adjust how Pacemaker mounts each volume to ensure that changes are immediately flushed to E-series block nodes. This prevents data loss when file nodes fail with active writes in progress.

For the parameters that you can adjust for your specific workloads or use cases, see the volume formatting and mounting configuration parameters.

3. **System settings in the Linux OS installed on the file nodes.**
   You can override the default Linux OS system settings when you create the Ansible inventory in step 4 of Create the Ansible inventory.

   The default settings were used to validate the BeeGFS on NetApp solution, but you can change them to adjust for your specific workloads or use cases. Some examples of the Linux OS system settings that you can change include the following:

   ◦ I/O queues on E-Series block devices.

     You can configure I/O queues on the E-Series block devices used as BeeGFS targets to:

     ▪ Adjust the scheduling algorithm based on the device type (NVMe, HDD, and so on).
     ▪ Increase the number of outstanding requests.
     ▪ Adjust request sizes.
     ▪ Optimize read ahead behavior.

   ◦ Virtual memory settings.

     You can adjust virtual memory settings for optimal sustained streaming performance.

   ◦ CPU settings.

     You can adjust the CPU frequency governor and other CPU configurations for maximum performance.

   ◦ Read request size.

     You can increase the maximum read request size for NVIDIA HCAs.

Performance tuning for block nodes

Based on the configuration profiles applied to a particular BeeGFS building block, the volume groups configured on the block nodes change slightly. For example, with a 24-drive EF600 block node:

- For the single base building block, including BeeGFS management, metadata, and storage services:
  ◦ 1x 2+2 RAID 10 volume group for BeeGFS management and metadata services
  ◦ 2x 8+2 RAID 6 volume groups for BeeGFS storage services
- For a BeeGFS metadata + storage building block:
  ◦ 1x 2+2 RAID 10 volume group for BeeGFS metadata services
  ◦ 2x 8+2 RAID 6 volume groups for BeeGFS storage services
- For BeeGFS storage only building block:
  ◦ 2x 10+2 RAID 6 volume groups for BeeGFS storage services

> As BeeGFS needs significantly less storage space for management and metadata versus storage, one option is to use smaller drives for the RAID 10 volume groups. Smaller drives should be populated in the outermost drive slots. For more information, see the deployment instructions.

These are all configured by the Ansible-based deployment, along with several other settings generally recommended to optimize performance/behavior including:

- Adjusting the global cache block size to 32KiB and adjusting demand-based cache flushing to 80%.
- Disabling autoload balancing (ensuring controller volume assignments stay as intended).
- Enabling read caching and disabling read-ahead caching.
- Enabling write caching with mirroring and requiring battery backup, so that caches persist through failure of a block node controller.
- Specifying the order that drives are assigned to volume groups, balancing I/O across available drive channels.

High capacity building block

The standard BeeGFS solution design is built with high performance workloads in mind. Customers looking for a high capacity use cases should observe the variations in design and performance characteristics outlined here.

Hardware and software configuration

Hardware and software configuration for the high capacity building block is standard except that the EF600 controllers should be replaced with a EF300 controllers with an option to attach between 1 and 7 IOM expansion trays with 60 drives each for each storage array, totaling 2 to 14 expansions trays per building block.

Customers deploying a high capacity building block design are likely to use only the base building block style configuration consisting of BeeGFS management, metadata, and storage services for each node. For cost efficiency, high capacity storage nodes should provision metadata volumes on the NVMe drives in the EF300 controller enclosure and should provision storage volumes to the NL-SAS drives in the expansion trays.

[high capacity rack diagram]

Sizing guidelines

These sizing guidelines assume high capacity building blocks are configured with one 2+2 NVMe SSD volume group for metadata in the base EF300 enclosure and 6x 8+2 NL-SAS volume groups per IOM expansion tray for storage.

Drive size (capacity HDDs)	Capacity per BB (1 Tray)	Capacity per BB (2 Trays)	Capacity per BB (3 Trays)	Capacity per BB (4 Trays)
4TB	439TB	878 TB	1317 TB	1756 TB
8 TB	878 TB	1756 TB	2634 TB	3512 TB
10 TB	1097 TB	2195 TB	3292 TB	4390 TB
12 TB	1317 TB	2634 TB	3951 TB	5268 TB
16 TB	1756 TB	3512 TB	5268 TB	7024 TB
18 TB	1975 TB	3951 TB	5927 TB	7902 TB

Deploy solution

Deployment overview

BeeGFS on NetApp can be deployed to validated file and block nodes using Ansible with NetApp's BeeGFS building block design.

Ansible collections and roles

The BeeGFS on NetApp solution is deployed using Ansible, a popular IT automation engine that automates application deployments. Ansible uses a series of files collectively known as the inventory, which models the BeeGFS file system you want to deploy.

Ansible allows companies like NetApp to expand on built-in functionality using collections available on Ansible Galaxy (see NetApp E-Series BeeGFS collection). Collections include modules that perform specific functions or tasks (such as creating an E-Series volume) and roles that can call multiple modules and other roles. This automated approach reduces the time needed to deploy the BeeGFS file system and the underlying HA cluster. Additionally, it simplifies maintaining and expanding the cluster and BeeGFS filesystem.

For additional details, see Learn about the Ansible inventory.

> ⓘ   Because numerous steps are involved in deploying the BeeGFS on NetApp solution, NetApp does not support manually deploying the solution.

Configuration profiles for BeeGFS building blocks

The deployment procedures cover the following configuration profiles:

- One base building block that includes management, metadata, and storage services.
- A second building block that includes metadata and storage services.
- A third building block that includes only storage services.

These profiles demonstrate the full range of recommended configuration profiles for the NetApp BeeGFS building blocks. For each deployment, the number of metadata and storage building blocks or storage services-only building blocks may vary depending on capacity and performance requirements.

Overview of deployment steps

Deployment involves the following high-level tasks:

Hardware deployment

1. Physically assemble each building block.

2. Rack and cable hardware.
   For detailed procedures, see Deploy hardware.

Software deployment

1. Set up file and block nodes.

   ◦ Configure BMC IPs on file nodes

   ◦ Install a supported operating system and configure management networking on file nodes

- Configure management IPs on block nodes

2. Set up an Ansible control node.

3. Tune system settings for performance.

4. Create the Ansible inventory.

5. Define Ansible inventory for BeeGFS building blocks.

6. Deploy BeeGFS using Ansible.

7. Configure BeeGFS clients.

> ⓘ The deploy procedures includes several examples where text needs to be copied to a file. Pay close attention to any inline comments denoted by "#" or "//" characters for anything that should or can be modified for a specific deployment. For example:
>
> ```
> `beegfs_ha_ntp_server_pools:  # THIS IS AN EXAMPLE OF A COMMENT!
>   - "pool 0.pool.ntp.org iburst maxsources 3"
>   - "pool 1.pool.ntp.org iburst maxsources 3"`
> ```

Derivative architectures with variations in deployment recommendations:

- High Capacity Building Block

Learn about the Ansible inventory

Before beginning a deployment, familiarize yourself with how Ansible is configured and used to deploy the BeeGFS on NetApp solution.

The Ansible inventory is a directory structure listing the file and block nodes for the BeeGFS file system to be deployed across. It includes hosts, groups, and variables describing the desired BeeGFS file system. The Ansible inventory needs to be stored on the Ansible control node, which is any machine with access to the file and block nodes used to run the Ansible playbook. Sample inventories can be downloaded from the NetApp E-Series BeeGFS GitHub.

Ansible modules and roles

To apply the configuration described by the Ansible inventory, use the various Ansible modules and roles provided in the NetApp E-Series Ansible collection (available from the NetApp E-Series BeeGFS GitHub) that deploy the end-to-end solution.

Each role in the NetApp E-Series Ansible collection is a complete end-to-end deployment of the BeeGFS on NetApp solution. The roles use the NetApp E-Series SANtricity, Host, and BeeGFS collections that allow you to configure the BeeGFS file system with HA (High Availability). You can then provision and map storage, and ensure the cluster storage is ready for use.

While in-depth documentation is provided with the roles, the deployment procedures describe how to use the role to deploy a NetApp Verified Architecture using the second generation BeeGFS building block design.

> ⓘ Although the deployment steps attempt to provide enough detail so that prior experience with Ansible is not a prerequisite, you should have some familiarity with Ansible and related terminology.

Inventory layout for a BeeGFS HA cluster

Define a BeeGFS HA cluster using the Ansible inventory structure.

Anyone with previous Ansible experience should be aware that the BeeGFS HA role implements a custom method for discovering which variables (or facts) apply to each host. This design simplifies structuring the Ansible inventory to describe resources that can run on multiple servers.

An Ansible inventory typically consists of the files in `host_vars` and `group_vars`, along with an `inventory.yml` file that assigns hosts to specific groups (and potentially groups to other groups).

> ⓘ Don't create any files with the content in this subsection, which is intended as an example only.

Although this configuration is predetermined based on the configuration profile, you should have a general understanding of how everything is laid out as an Ansible inventory, as follows:

```
# BeeGFS HA (High Availability) cluster inventory.
all:
  children:
    # Ansible group representing all block nodes:
    eseries_storage_systems:
      hosts:
        netapp01:
        netapp02:
    # Ansible group representing all file nodes:
    ha_cluster:
      children:
        meta_01:  # Group representing a metadata service with ID 01.
          hosts:
            beegfs_01:  # This service is preferred on the first file
 node.
            beegfs_02:  # And can failover to the second file node.
        meta_02:  # Group representing a metadata service with ID 02.
          hosts:
            beegfs_02:  # This service is preferred on the second file
 node.
            beegfs_01: # And can failover to the first file node.
```

For each service, an additional file is created under `group_vars` describing its configuration:

```
# meta_01 - BeeGFS HA Metadata Resource Group
beegfs_ha_beegfs_meta_conf_resource_group_options:
  connMetaPortTCP: 8015
  connMetaPortUDP: 8015
  tuneBindToNumaZone: 0
floating_ips:
  - i1b: <IP>/<SUBNET_MASK>
  - i2b: <IP>/<SUBNET_MASK>
# Type of BeeGFS service the HA resource group will manage.
beegfs_service: metadata # Choices: management, metadata, storage.
# What block node should be used to create a volume for this service:
beegfs_targets:
  netapp01:
    eseries_storage_pool_configuration:
      - name: beegfs_m1_m2_m5_m6
        raid_level: raid1
        criteria_drive_count: 4
        common_volume_configuration:
          segment_size_kb: 128
        volumes:
          - size: 21.25
            owning_controller: A
```

This layout allows the BeeGFS service, network, and storage configuration for each resource to be defined in a single place. Behind the scenes, the BeeGFS role aggregates the necessary configuration for each file and block node based on this inventory structure.

> (i) The BeeGFS numerical and string node ID for each service is automatically configured based on the group name. Thus, in addition to the general Ansible requirement for group names to be unique, groups representing a BeeGFS service must end in a number that is unique for the type of BeeGFS service the group represents. For example, meta_01 and stor_01 are allowed, but metadata_01 and meta_01 are not.

Review best practices

Follow the best practice guidelines when deploying the BeeGFS on NetApp solution.

Standard conventions

When physically assembling and creating the Ansible inventory file, follow these standard conventions (for more information, see Create the Ansible inventory).

- File node host names are sequentially numbered (h01-hN) with lower numbers at the top of the rack and higher numbers at the bottom.

  For example, the naming convention `[location][row][rack]hN` looks like: `beegfs_01`.

- Each block node is comprised of two storage controllers, each with their own host name.

A storage array name is used to refer to the whole block storage system as part of an Ansible inventory. The storage array names should be sequentially numbered (a01 - aN), and the host names for individual controllers are derived from that naming convention.

For example, a block node named `ictad22a01` typically can have host names configured for each controller like `ictad22a01-a` and `ictad22a01-b`, but be referred to in an Ansible inventory as `netapp_01`.

- File and block nodes within the same building block share the same numbering scheme and are adjacent to each other in the rack with both file nodes on top and both block nodes directly underneath them.

  For example, in the first building block, file nodes h01 and h02 are both directly connected to block nodes a01 and a02. From top to bottom, the host names are h01, h02, a01, and a02.

- Building blocks are installed in sequential order based on their host names, so that lower numbered host names are at the top of the rack and higher numbered host names are at the bottom.

  The intent is to minimize the length of cable running to the top of rack switches, and to define a standard deployment practice to simplify troubleshooting. For datacenters where this is not allowed due to concerns around rack stability, the inverse is certainly allowed, populating the rack from the bottom up.

InfiniBand storage network configuration

Half the InfiniBand ports on each file node are used to connect directly to block nodes. The other half are connected to the InfiniBand switches and are used for BeeGFS client-server connectivity. When determining the size of the IPoIB subnets that are used for BeeGFS clients and servers, you must consider the anticipated growth of your compute/GPU cluster and BeeGFS file system. If you must deviate from the recommended IP ranges, keep in mind that each direct connect in a single building block has a unique subnet and there is no overlap with subnets used for client-server connectivity.

Direct connects

File and block nodes within each building block always use the IPs in the following table for their direct connects.

> (i) This addressing scheme adheres to the following rule: The third octet is always odd or even, which depends on whether the file node is odd or even.

File node	IB port	IP address	Block node	IB port	Physical IP	Virtual IP
Odd (h1)	i1a	192.168.1.10	Odd (c1)	2a	192.168.1.100	192.168.1.101
Odd (h1)	i2a	192.168.3.10	Odd (c1)	2a	192.168.3.100	192.168.3.101
Odd (h1)	i3a	192.168.5.10	Even (c2)	2a	192.168.5.100	192.168.5.101
Odd (h1)	i4a	192.168.7.10	Even (c2)	2a	192.168.7.100	192.168.7.101
Even (h2)	i1a	192.168.2.10	Odd (c1)	2b	192.168.2.100	192.168.2.101
Even (h2)	i2a	192.168.4.10	Odd (c1)	2b	192.168.4.100	192.168.4.101
Even (h2)	i3a	192.168.6.10	Even (c2)	2b	192.168.6.100	192.168.6.101
Even (h2)	i4a	192.168.8.10	Even (c2)	2b	192.168.8.100	192.168.8.101

BeeGFS client-server IPoIB addressing schemes

Each file node runs multiple BeeGFS server services (management, metadata, or storage). To allow each service to fail over independently to the other file node, each is configured with unique IP addresses that can float between both nodes (sometimes referred to as a logical interface or LIF).

While not mandatory, this deployment presumes the following IPoIB subnet ranges are in use for these connections and defines a standard addressing scheme that applies the following rules:

- The second octet is always odd or even, based on whether the file node InfiniBand port is odd or even.
- BeeGFS cluster IPs are always `xxx. 127.100.yyy` or `xxx.128.100.yyy`.

> (i) In addition to the interface used for in-band OS management, additional interfaces can be used by Corosync for cluster heart beating and synchronization. This ensures that the loss of a single interface does not bring down the entire cluster.

- The BeeGFS Management service is always at `xxx.yyy.101.0` or `xxx.yyy.102.0`.
- BeeGFS Metadata services are always at `xxx.yyy.101.zzz` or `xxx.yyy.102.zzz`.
- BeeGFS Storage services are always at `xxx.yyy.103.zzz` or `xxx.yyy.104.zzz`.
- Addresses in the range `100.xxx.1.1` through `100.xxx.99.255` are reserved for clients.

IPoIB single subnet addressing scheme

This deployment guide will utilize a single subnet schema given the advantages listed in the software architecture.

Subnet: 100.127.0.0/16
The following table provides the range for a single subnet: 100.127.0.0/16.

Purpose	InfiniBand port	IP address or range
BeeGFS Cluster IP	i1b or i4b	100.127.100.1 - 100.127.100.255
BeeGFS Management	i1b	100.127.101.0
	i2b	100.127.102.0
BeeGFS Metadata	i1b or i3b	100.127.101.1 - 100.127.101.255
	i2b or i4b	100.127.102.1 - 100.127.102.255
BeeGFS Storage	i1b or i3b	100.127.103.1 - 100.127.103.255
	i2b or i4b	100.127.104.1 - 100.127.104.255
BeeGFS Clients	(varies by client)	100.127.1.1 - 100.127.99.255

IPoIB two subnet addressing scheme

A two subnet addressing scheme is no longer recommended, but can still be implemented. Refer to the tables below for a recommended two subnet scheme.

Subnet A: 100.127.0.0/16
The following table provides the range for Subnet A: 100.127.0.0/16.

Purpose	InfiniBand port	IP address or range
BeeGFS Cluster IP	i1b	100.127.100.1 - 100.127.100.255
BeeGFS Management	i1b	100.127.101.0
BeeGFS Metadata	i1b or i3b	100.127.101.1 - 100.127.101.255
BeeGFS Storage	i1b or i3b	100.127.103.1 - 100.127.103.255
BeeGFS Clients	(varies by client)	100.127.1.1 - 100.127.99.255

Subnet B: 100.128.0.0/16

The following table provides the range for Subnet B: 100.128.0.0/16.

Purpose	InfiniBand port	IP address or range
BeeGFS Cluster IP	i4b	100.128.100.1 - 100.128.100.255
BeeGFS Management	i2b	100.128.102.0
BeeGFS Metadata	i2b or i4b	100.128.102.1 - 100.128.102.255
BeeGFS Storage	i2b or i4b	100.128.104.1 - 100.128.104.255
BeeGFS Clients	(varies by client)	100.128.1.1 - 100.128.99.255

> ⓘ Not all IPs in the above ranges are used in this NetApp Verified Architecture. They demonstrate how IP addresses can be pre-allocated to allow easy file system expansion using a consistent IP addressing scheme. In this scheme, BeeGFS file nodes and service IDs correspond with the fourth octet of a well-known range of IPs. The file system could certainly scale beyond 255 nodes or services if needed.

Deploy hardware

Each building block consists of two validated x86 file nodes directly connected to two block nodes using HDR (200Gb) InfiniBand cables.

> ⓘ A minimum of two building blocks are required to establish quorum in the failover cluster. A two-node cluster has limitations that might prevent a successful failover from occurring. You can configure a two-node cluster by incorporating a third device as a tiebreaker; however, this documentation does not describe that design.

The following steps are identical for each building block in the cluster, regardless of whether it is used to run both BeeGFS metadata and storage services, or just storage services, unless it is otherwise noted.

Steps

1. Set up each BeeGFS file node with four Host Channel Adapters (HCAs) using the models specified in the Technical requirements. Insert the HCAs into the PCIe slots of your file node according to the specifications below:

   ◦ **Lenovo ThinkSystem SR665 V3 Server:** Use PCIe slots 1, 2, 4, and 5.

   ◦ **Lenovo ThinkSystem SR665 Server:** Use PCIe slots 2, 3, 5, and 6.

2. Configure each BeeGFS block node with a dual-port 200Gb Host Interface Card (HIC) and install the HIC in each of its two storage controllers.

Rack the building blocks so the two BeeGFS file nodes are above the BeeGFS block nodes. The following figure shows the correct hardware configuration for the BeeGFS building block using Lenovo ThinkSystem SR665 V3 servers as the file nodes (rear view).
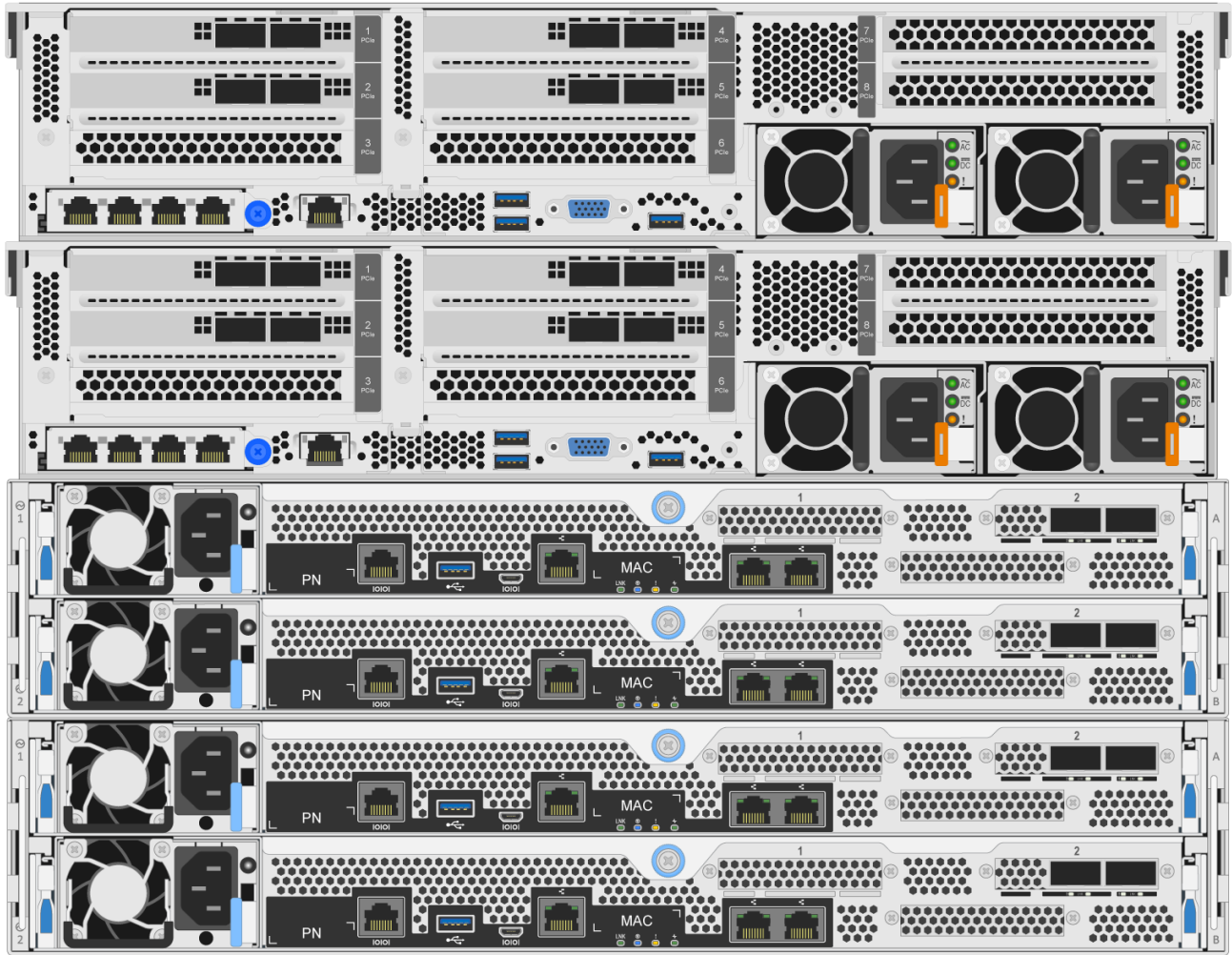

> ⓘ The power supply configuration for production use cases should typically use redundant PSUs.

3. If needed, install the drives in each of the BeeGFS block nodes.

   a. If the building block will be used to run BeeGFS metadata and storage services and smaller drives are used for metadata volumes, verify that they are populated in the outermost drive slots, as shown in the figure below.

   b. For all building block configurations, if a drive enclosure is not fully populated, make sure that an equal number of drives are populated in slots 0–11 and 12–23 for optimal performance.

RAID 6 (8+2) Data

RAID 6 (8+2) Data

RAID 1 (2+2) Metadata
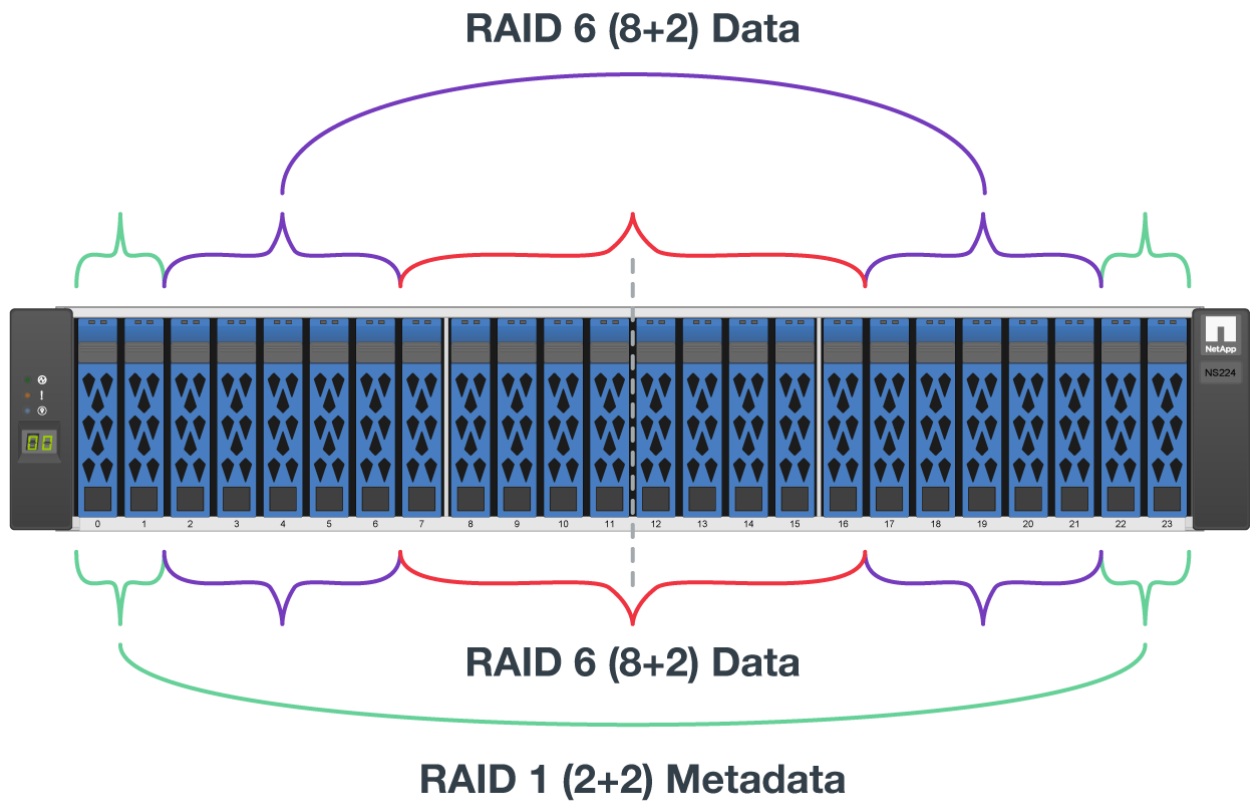
4. Connect the block and file nodes using the 1m InfiniBand HDR 200Gb direct attach copper cables, so that they match the topology shown in the following figure.
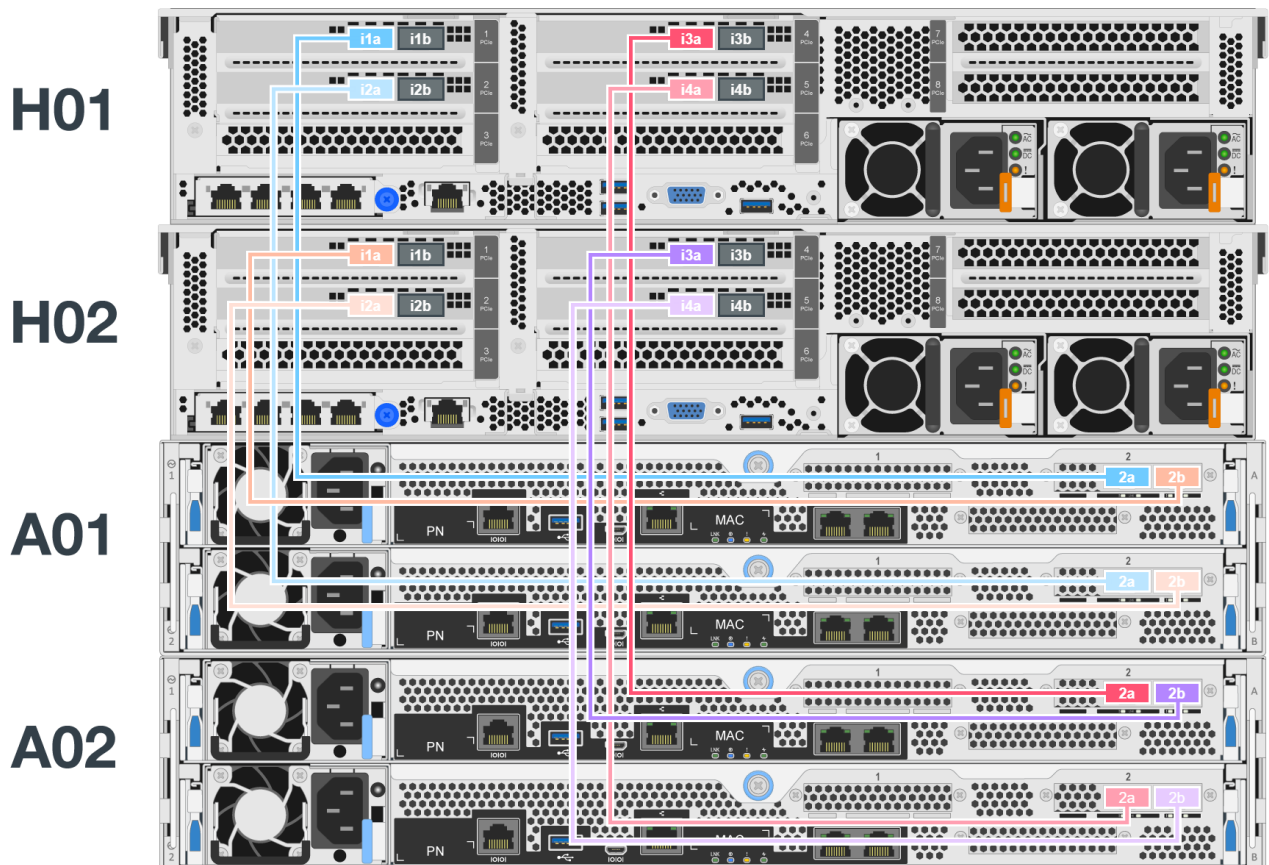
> (i) The nodes across multiple building blocks are never directly connected. Each building block should be treated as a standalone unit and all communication between building blocks occurs through network switches.

5. Connect the remaining InfiniBand ports on the file node to the storage network's InfiniBand switch using the 2m InfiniBand cables specific to your InfiniBand storage switch.

   When using splitter cables to connect the storage switch to file nodes, one cable should branch out from the switch and connect to the ports outlined in light green. Another splitter cable should branch out from the switch and connect to the ports outlined in dark green.

   Additionally, for storage networks with redundant switches, ports outlined in light green should connect to one switch, while ports in dark green should connect to another switch.
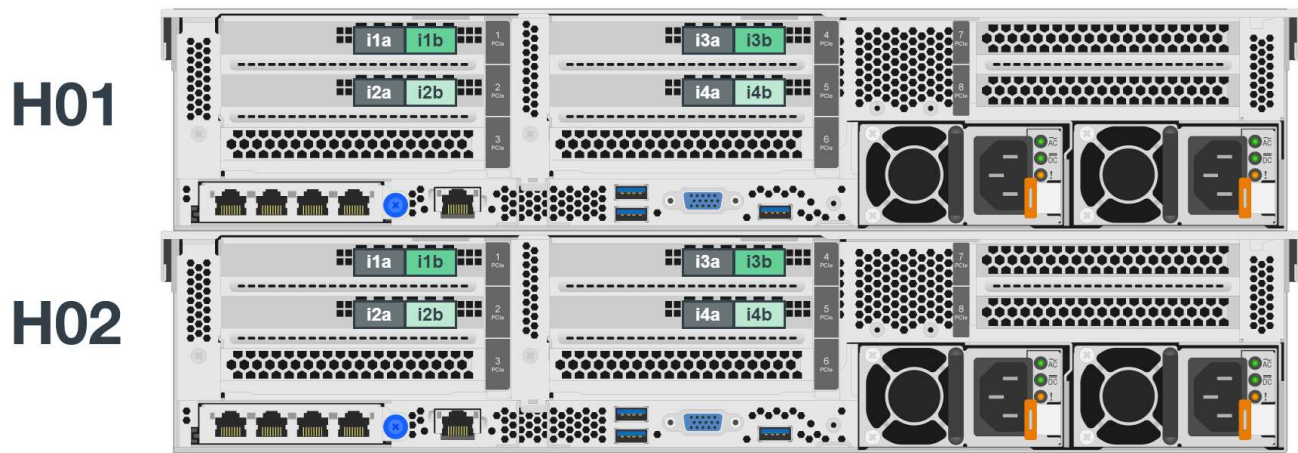
6. As needed, assemble additional building blocks following the same cabling guidelines.

> (i) The total number of building blocks that can be deployed in a single rack depends on the available power and cooling at each site.

Deploy software

Set up file nodes and block nodes

While most software configuration tasks are automated using the NetApp-provided Ansible collections, you must configure networking on the baseboard management controller (BMC) of each server and configure the management port on each controller.

Set up file nodes

1. Configure networking on the baseboard management controller (BMC) of each server.

   To learn how to configure networking for the validated Lenovo SR665 V3 file nodes, see the Lenovo ThinkSystem Documentation.

	A baseboard management controller (BMC), sometimes referred to as a service processor, is the generic name for the out-of-band management capability built into various server platforms that can provide remote access even if the operating system is not installed or accessible. Vendors typically market this functionality with their own branding. For example, on the Lenovo SR665, the BMC is referred to as the *Lenovo XClarity Controller (XCC)*.

2. Configure the system settings for maximum performance.

   You configure the system settings using the UEFI setup (formerly known as the BIOS) or by using the Redfish APIs provided by many BMCs. The system settings vary based on the server model used as a file node.

   To learn how to configure the system settings for the validated Lenovo SR665 V3 file nodes, see Tune system settings for performance.

3. Install Red Hat Enterprise Linux (RHEL) 9.4 and configure the host name and network port used to manage the operating system including SSH connectivity from the Ansible control node.

   Do not configure IPs on any of the InfiniBand ports at this time.

	While not strictly required, subsequent sections presume that host names are sequentially numbered (such as h1-hN) and refer to tasks that should be completed on odd versus even numbered hosts.

4. Use Red Hat Subscription Manager to register and subscribe the system to allow installation of the required packages from the official Red Hat repositories and to limit updates to the supported version of Red Hat: `subscription-manager release --set=9.4`. For instructions, see How to register and subscribe a RHEL system and How to limit updates.

5. Enable the Red Hat repository containing the packages required for high availability.

   ```
   subscription-manager repo-override --repo=rhel-9-for-x86_64
   -highavailability-rpms --add=enabled:1
   ```

6. Update all HCA firmware to the version recommended in Technology requirements using the Update file node adapter firmware guide.

Set up block nodes

Set up the EF600 block nodes by configuring the management port on each controller.

1. Configure the management port on each EF600 controller.

   For instructions on configuring ports, go to the E-Series Documentation Center.

2. Optionally, set the storage array name for each system.

   Setting a name can make it easier to refer to each system in subsequent sections. For instructions on setting the array name, go to the E-Series Documentation Center.

> ℹ️ While not strictly required, subsequent topics presume storage array names are sequentially numbered (such as c1 - cN) and refer to the steps that should be completed on odd versus even numbered systems.

Tune file node system settings for performance

To maximize performance, we recommend configuring the system settings on the server model you use as your file nodes.

The system settings vary depending on the server model you use as your file node. This topic describes how to configure the system settings for the validated Lenovo ThinkSystem SR665 server file nodes.

Use the UEFI interface to tune system settings

The Lenovo SR665 V3 server's system firmware contains numerous tuning parameters that can be set through the UEFI interface. These tuning parameters can affect all aspects of how the server functions and how well the server performs.

Under **UEFI Setup > System Settings**, adjust the following system settings:

Operating Mode menu

System Setting	Change to
Operating Mode	Custom
cTDP	Manual
cTDP Manual	350
Package Power Limit	Manual
Efficiency Mode	Disable
Global-Cstate-Control	Disable
SOC P-states	P0
DF C-States	Disable
P-State	Disable
Memory Power Down Enable	Disable
NUMA Nodes per Socket	NPS1

Devices and I/O ports menu

System Setting	Change to
IOMMU	Disable

Power menu

System Setting	Change to
PCIe Power Brake	Disable

Processors menu

System Setting	Change to
Global C-state Control	Disable
DF C-States	Disable
SMT Mode	Disable
CPPC	Disable

Use Redfish API to tune system settings

In addition to using UEFI Setup, you can use the Redfish API to change system settings.

Example

```
curl --request PATCH \
  --url https://<BMC_IP_ADDRESS>/redfish/v1/Systems/1/Bios/Pending \
  --user <BMC_USER>:<BMC- PASSWORD> \
  --header 'Content-Type: application/json' \
  --data '{
"Attributes": {
"OperatingModes_ChooseOperatingMode": "CustomMode",
"Processors_cTDP": "Manual",
"Processors_PackagePowerLimit": "Manual",
"Power_EfficiencyMode": "Disable",
"Processors_GlobalC_stateControl": "Disable",
"Processors_SOCP_states": "P0",
"Processors_DFC_States": "Disable",
"Processors_P_State": "Disable",
"Memory_MemoryPowerDownEnable": "Disable",
"DevicesandIOPorts_IOMMU": "Disable",
"Power_PCIePowerBrake": "Disable",
"Processors_GlobalC_stateControl": "Disable",
"Processors_DFC_States": "Disable",
"Processors_SMTMode": "Disable",
"Processors_CPPC": "Disable",
"Memory_NUMANodesperSocket":"NPS1"
}
}
'
```

For detailed information on the Redfish schema, see the DMTF website.

Set up an Ansible control node

To set up an Ansible control node, you must designate a virtual or physical machine with network access to all file and block nodes deployed for the BeeGFS on NetApp solution.

Review the Technical requirements for a list of recommended package versions. The following steps were tested on Ubuntu 22.04. For steps specific to your preferred Linux distribution, see the Ansible documentation.

1. From your Ansible control node, install the following Python and Python Virtual Environment packages.

   ```
   sudo apt-get install python3 python3-pip python3-setuptools python3.10-
   venv
   ```

2. Create a Python virtual environment.

```
python3 -m venv ~/pyenv
```

3. Activate the virtual environment.

```
source ~/pyenv/bin/activate
```

4. Install the required Python packages within the activated virtual environment.

```
pip install ansible netaddr cryptography passlib
```

5. Install the BeeGFS collection using Ansible Galaxy.

```
ansible-galaxy collection install netapp_eseries.beegfs
```

6. Verify the installed versions of Ansible, Python, and the BeeGFS collection match the Technical requirements.

```
ansible --version
ansible-galaxy collection list netapp_eseries.beegfs
```

7. Set up passwordless SSH to allow Ansible to access the remote BeeGFS file nodes from the Ansible control node.

    a. On the Ansible control node, if needed, generate a pair of public keys.

```
ssh-keygen
```

    b. Set up passwordless SSH to each of the file nodes.

```
ssh-copy-id <ip_or_hostname>
```

> (!) Do **not** set up passwordless SSH to the block nodes. This is neither supported nor required.

Create the Ansible inventory

To define the configuration for file and block nodes, you create an Ansible inventory that represents the BeeGFS file system you want to deploy. The inventory includes hosts, groups, and variables describing the desired BeeGFS file system.

Step 1: Define configuration for all building blocks

Define the configuration that applies to all building blocks, regardless of which configuration profile you may apply to them individually.

Before you begin

- Choose a subnet addressing scheme for your deployment. Due to the benefits listed in the software architecture, it is recommended to use a single subnet addressing scheme.

Steps

1. On your Ansible control node, identify a directory that you want to use to store the Ansible inventory and playbook files.

   Unless otherwise noted, all files and directories created in this step and following steps are created relative to this directory.

2. Create the following subdirectories:

   `host_vars`

   `group_vars`

   `packages`

3. Create a subdirectory for cluster passwords and secure the file by encrypting it with Ansible Vault (see Encrypting content with Ansible Vault):

   a. Create the subdirectory `group_vars/all`.

   b. In the `group_vars/all` directory, create a passwords file labeled `passwords.yml`.

   c. Populate the `passwords.yml file` with the following, replacing all username and password parameters according to your configuration:

```
# Credentials for storage system's admin password
eseries_password: <PASSWORD>

# Credentials for BeeGFS file nodes
ssh_ha_user: <USERNAME>
ssh_ha_become_pass: <PASSWORD>

# Credentials for HA cluster
ha_cluster_username: <USERNAME>
ha_cluster_password: <PASSWORD>
ha_cluster_password_sha512_salt: randomSalt

# Credentials for fencing agents
# OPTION 1: If using APC Power Distribution Units (PDUs) for fencing:
# Credentials for APC PDUs.
apc_username: <USERNAME>
apc_password: <PASSWORD>

# OPTION 2: If using the Redfish APIs provided by the Lenovo XCC (and
other BMCs) for fencing:
# Credentials for XCC/BMC of BeeGFS file nodes
bmc_username: <USERNAME>
bmc_password: <PASSWORD>
```

d. Run `ansible-vault encrypt passwords.yml` and set a vault password when prompted.

Step 2: Define configuration for individual file and block nodes

Define the configuration that applies to individual file nodes and individual building block nodes.

1. Under `host_vars/`, create a file for each BeeGFS file node named `<HOSTNAME>.yml` with the following content, paying special attention to the notes regarding content to populate for BeeGFS cluster IPs and host names ending in odd versus even numbers.

   Initially, the file node interface names do match what is listed here (such as ib0 or ibs1f0). These custom names are configured in Step 4: Define configuration that should apply to all file nodes.

```
ansible_host: "<MANAGEMENT_IP>"
eseries_ipoib_interfaces:  # Used to configure BeeGFS cluster IP
addresses.
  - name: i1b
    address: 100.127.100. <NUMBER_FROM_HOSTNAME>/16
  - name: i4b
    address: 100.127.100. <NUMBER_FROM_HOSTNAME>/16
beegfs_ha_cluster_node_ips:
  - <MANAGEMENT_IP>
  - <i1b_BEEGFS_CLUSTER_IP>
  - <i4b_BEEGFS_CLUSTER_IP>
# NVMe over InfiniBand storage communication protocol information
# For odd numbered file nodes (i.e., h01, h03, ..):
eseries_nvme_ib_interfaces:
  - name: i1a
    address: 192.168.1.10/24
    configure: true
  - name: i2a
    address: 192.168.3.10/24
    configure: true
  - name: i3a
    address: 192.168.5.10/24
    configure: true
  - name: i4a
    address: 192.168.7.10/24
    configure: true
# For even numbered file nodes (i.e., h02, h04, ..):
# NVMe over InfiniBand storage communication protocol information
eseries_nvme_ib_interfaces:
  - name: i1a
    address: 192.168.2.10/24
    configure: true
  - name: i2a
    address: 192.168.4.10/24
    configure: true
  - name: i3a
    address: 192.168.6.10/24
    configure: true
  - name: i4a
    address: 192.168.8.10/24
    configure: true
```

> ℹ️ If you have already deployed the BeeGFS cluster, you must stop the cluster before adding or changing statically configured IP addresses, including cluster IPs and IPs used for NVMe/IB. This is required so these changes take effect properly and do not disrupt cluster operations.

2. Under `host_vars/`, create a file for each BeeGFS block node named `<HOSTNAME>.yml` and populate it with the following content.

   Pay special attention to the notes regarding content to populate for storage array names ending in odd versus even numbers.

   For each block node, create one file and specify the `<MANAGEMENT_IP>` for one of the two controllers (usually A).

```
eseries_system_name: <STORAGE_ARRAY_NAME>
eseries_system_api_url: https://<MANAGEMENT_IP>:8443/devmgr/v2/
eseries_initiator_protocol: nvme_ib
# For odd numbered block nodes (i.e., a01, a03, ..):
eseries_controller_nvme_ib_port:
  controller_a:
    - 192.168.1.101
    - 192.168.2.101
    - 192.168.1.100
    - 192.168.2.100
  controller_b:
    - 192.168.3.101
    - 192.168.4.101
    - 192.168.3.100
    - 192.168.4.100
# For even numbered block nodes (i.e., a02, a04, ..):
eseries_controller_nvme_ib_port:
  controller_a:
    - 192.168.5.101
    - 192.168.6.101
    - 192.168.5.100
    - 192.168.6.100
  controller_b:
    - 192.168.7.101
    - 192.168.8.101
    - 192.168.7.100
    - 192.168.8.100
```

Step 3: Define configuration that should apply to all file and block nodes

You can define configuration common to a group of hosts under `group_vars` in a file name that corresponds with the group. This prevents repeating a shared configuration in multiple places.

About this task

Hosts can be in more than one group, and at runtime, Ansible chooses what variables apply to a particular host based on its variable precedence rules. (For more information on these rules, see the Ansible documentation for Using variables.)

Host-to-group assignments are defined in the actual Ansible inventory file, which is created towards the end of this procedure.

Step

In Ansible, any configuration you want to apply to all hosts can be defined in a group called `All`. Create the file `group_vars/all.yml` with the following content:

```
ansible_python_interpreter: /usr/bin/python3
beegfs_ha_ntp_server_pools:  # Modify the NTP server addressess if
desired.
  - "pool 0.pool.ntp.org iburst maxsources 3"
  - "pool 1.pool.ntp.org iburst maxsources 3"
```

Step 4: Define configuration that should apply to all file nodes

The shared configuration for file nodes is defined in a group called `ha_cluster`. The steps in this section build out the configuration that should be included in the `group_vars/ha_cluster.yml` file.

Steps

1. At the top of the file, define the defaults, including the password to use as the `sudo` user on the file nodes.

```
### ha_cluster Ansible group inventory file.
# Place all default/common variables for BeeGFS HA cluster resources
below.
### Cluster node defaults
ansible_ssh_user: {{ ssh_ha_user }}
ansible_become_password: {{ ssh_ha_become_pass }}
eseries_ipoib_default_hook_templates:
  - 99-multihoming.j2   # This is required for single subnet
deployments, where static IPs containing multiple IB ports are in the
same IPoIB subnet. i.e: cluster IPs, multirail, single subnet, etc.
# If the following options are specified, then Ansible will
automatically reboot nodes when necessary for changes to take effect:
eseries_common_allow_host_reboot: true
eseries_common_reboot_test_command: "! systemctl status
eseries_nvme_ib.service || systemctl --state=exited | grep
eseries_nvme_ib.service"
eseries_ib_opensm_options:
  virt_enabled: "2"
  virt_max_ports_in_process: "0"
```

> ℹ️ If the `ansible_ssh_user` is already `root`, then you can optionally omit the `ansible_become_password` and specify the `--ask-become-pass` option when running the playbook.

2. Optionally, configure a name for the high-availability (HA) cluster and specify a user for intra-cluster communication.

   If you are modifying the private IP addressing scheme, you must also update the default `beegfs_ha_mgmtd_floating_ip`. This must match what you configure later for the BeeGFS Management resource group.

   Specify one or more emails that should receive alerts for cluster events using `beegfs_ha_alert_email_list`.

```
### Cluster information
beegfs_ha_firewall_configure: True
eseries_beegfs_ha_disable_selinux: True
eseries_selinux_state: disabled
# The following variables should be adjusted depending on the desired
configuration:
beegfs_ha_cluster_name: hacluster                    # BeeGFS HA cluster
name.
beegfs_ha_cluster_username: "{{ ha_cluster_username }}" # Parameter for
BeeGFS HA cluster username in the passwords file.
beegfs_ha_cluster_password: "{{ ha_cluster_password }}" # Parameter for
BeeGFS HA cluster username's password in the passwords file.
beegfs_ha_cluster_password_sha512_salt: "{{
ha_cluster_password_sha512_salt }}" # Parameter for BeeGFS HA cluster
username's password salt in the passwords file.
beegfs_ha_mgmtd_floating_ip: 100.127.101.0          # BeeGFS management
service IP address.
# Email Alerts Configuration
beegfs_ha_enable_alerts: True
beegfs_ha_alert_email_list: ["email@example.com"]  # E-mail recipient
list for notifications when BeeGFS HA resources change or fail.  Often a
distribution list for the team responsible for managing the cluster.
beegfs_ha_alert_conf_ha_group_options:
      mydomain: "example.com"
# The mydomain parameter specifies the local internet domain name. This
is optional when the cluster nodes have fully qualified hostnames (i.e.
host.example.com).
# Adjusting the following parameters is optional:
beegfs_ha_alert_timestamp_format: "%Y-%m-%d %H:%M:%S.%N" #%H:%M:%S.%N
beegfs_ha_alert_verbosity: 3
#  1) high-level node activity
#  3) high-level node activity + fencing action information + resources
(filter on X-monitor)
#  5) high-level node activity + fencing action information + resources
```

ⓘ   While seemingly redundant, `beegfs_ha_mgmtd_floating_ip` is important when you scale the BeeGFS file system beyond a single HA cluster. Subsequent HA clusters are deployed without an additional BeeGFS management service and point at the management service provided by the first cluster.

3. Configure a fencing agent. (For more details, see Configure fencing in a Red Hat High Availability cluster.) The following output shows examples for configuring common fencing agents. Choose one of these options.

   For this step, be aware that:

- By default, fencing is enabled, but you need to configure a fencing *agent*.

- The `<HOSTNAME>` specified in the `pcmk_host_map` or `pcmk_host_list` must correspond with the hostname in the Ansible inventory.

- Running the BeeGFS cluster without fencing is not supported, particularly in production. This is largely to ensure when BeeGFS services, including any resource dependencies like block devices, fail over due to an issue, there is no risk of concurrent access by multiple nodes that result in file system corruption or other undesirable or unexpected behavior. If fencing must be disabled, refer to the general notes in the BeeGFS HA role's getting started guide and set `beegfs_ha_cluster_crm_config_options["stonith-enabled"]` to false in `ha_cluster.yml`.

- There are multiple node-level fencing devices available, and the BeeGFS HA role can configure any fencing agent available in the Red Hat HA package repository. When possible, use a fencing agent that works through the uninterruptible power supply (UPS) or rack power distribution unit (rPDU), because some fencing agents such as the baseboard management controller (BMC) or other lights-out devices that are built into the server might not respond to the fence request under certain failure scenarios.

```
### Fencing configuration:
# OPTION 1: To enable fencing using APC Power Distribution Units
(PDUs):
beegfs_ha_fencing_agents:
 fence_apc:
    - ipaddr: <PDU_IP_ADDRESS>
      login: "{{ apc_username }}" # Parameter for APC PDU username in
the passwords file.
      passwd: "{{ apc_password }}" # Parameter for APC PDU password in
the passwords file.
      pcmk_host_map:
"<HOSTNAME>:<PDU_PORT>,<PDU_PORT>;<HOSTNAME>:<PDU_PORT>,<PDU_PORT>"
# OPTION 2: To enable fencing using the Redfish APIs provided by the
Lenovo XCC (and other BMCs):
redfish: &redfish
  username: "{{ bmc_username }}" # Parameter for XCC/BMC username in
the passwords file.
  password: "{{ bmc_password }}" # Parameter for XCC/BMC password in
the passwords file.
    ssl_insecure: 1 # If a valid SSL certificate is not available
specify "1".
beegfs_ha_fencing_agents:
  fence_redfish:
    - pcmk_host_list: <HOSTNAME>
      ip: <BMC_IP>
      <<: *redfish
    - pcmk_host_list: <HOSTNAME>
      ip: <BMC_IP>
      <<: *redfish
# For details on configuring other fencing agents see
https://access.redhat.com/documentation/en-
us/red_hat_enterprise_linux/9/html/configuring_and_managing_high_avai
lability_clusters/assembly_configuring-fencing-configuring-and-
managing-high-availability-clusters.
```

4. Enable recommended performance tuning in the Linux OS.

   While many users find the default settings for the performance parameters generally work well, you can optionally change the default settings for a particular workload. As such, these recommendations are included in the BeeGFS role, but are not enabled by default to ensure users are aware of the tuning applied to their file system.

   To enable performance tuning, specify:

```
### Performance Configuration:
beegfs_ha_enable_performance_tuning: True
```

5. (Optional) You can adjust the performance tuning parameters in the Linux OS as needed.

   For a comprehensive list of the available tuning parameters that you can adjust, see the Performance Tuning Defaults section of the BeeGFS HA role in E-Series BeeGFS GitHub site. The default values can be overridden for all nodes in the cluster in this file or the `host_vars` file for an individual node.

6. To allow full 200Gb/HDR connectivity between block and file nodes, use the Open Subnet Manager (OpenSM) package from the NVIDIA Open Fabrics Enterprise Distribution (MLNX_OFED). The MLNX_OFED version in listed the file node requirements comes bundled with the recommended OpenSM packages. Although deployment using Ansible is supported, you must first install the MLNX_OFED driver on all file nodes.

   a. Populate the following parameters in `group_vars/ha_cluster.yml` (adjust packages as needed):

   ```
   ### OpenSM package and configuration information
   eseries_ib_opensm_options:
     virt_enabled: "2"
     virt_max_ports_in_process: "0"
   ```

7. Configure the `udev` rule to ensure consistent mapping of logical InfiniBand port identifiers to underlying PCIe devices.

   The `udev` rule must be unique to the PCIe topology of each server platform used as a BeeGFS file node.

   Use the following values for verified file nodes:

```
### Ensure Consistent Logical IB Port Numbering
# OPTION 1: Lenovo SR665 V3 PCIe address-to-logical IB port mapping:
eseries_ipoib_udev_rules:
  "0000:01:00.0": i1a
  "0000:01:00.1": i1b
  "0000:41:00.0": i2a
  "0000:41:00.1": i2b
  "0000:81:00.0": i3a
  "0000:81:00.1": i3b
  "0000:a1:00.0": i4a
  "0000:a1:00.1": i4b

# OPTION 2: Lenovo SR665 PCIe address-to-logical IB port mapping:
eseries_ipoib_udev_rules:
  "0000:41:00.0": i1a
  "0000:41:00.1": i1b
  "0000:01:00.0": i2a
  "0000:01:00.1": i2b
  "0000:a1:00.0": i3a
  "0000:a1:00.1": i3b
  "0000:81:00.0": i4a
  "0000:81:00.1": i4b
```

8. (Optional) Update the metadata target selection algorithm.

```
beegfs_ha_beegfs_meta_conf_ha_group_options:
  tuneTargetChooser: randomrobin
```

> (i) In verification testing, `randomrobin` was typically used to ensure that test files were evenly distributed across all BeeGFS storage targets during performance benchmarking (for more information on benchmarking, see the BeeGFS site for Benchmarking a BeeGFS System). With real world use, this might cause lower numbered targets to fill up faster than higher numbered targets. Omitting `randomrobin` and just using the default `randomized` value has been shown to provide good performance while still utilizing all available targets.

Step 5: Define the configuration for the common block node

The shared configuration for block nodes is defined in a group called `eseries_storage_systems`. The steps in this section build out the configuration that should be included in the `group_vars/eseries_storage_systems.yml` file.

Steps

1. Set the Ansible connection to local, provide the system password, and specify if SSL certificates should be verified. (Normally, Ansible uses SSH to connect to managed hosts, but in the case of the NetApp E-Series storage systems used as block nodes, the modules use the REST API for communication.) At the top of

the file, add the following:

```
### eseries_storage_systems Ansible group inventory file.
# Place all default/common variables for NetApp E-Series Storage Systems
here:
ansible_connection: local
eseries_system_password: {{ eseries_password }} # Parameter for E-Series
storage array password in the passwords file.
eseries_validate_certs: false
```

2. To ensure optimal performance, install the versions listed for block nodes in Technical requirements.

   Download the corresponding files from the NetApp Support site. You can either upgrade them manually or include them in the `packages/` directory of the Ansible control node, and then populate the following parameters in `eseries_storage_systems.yml` to upgrade using Ansible:

```
# Firmware, NVSRAM, and Drive Firmware (modify the filenames as needed):
eseries_firmware_firmware: "packages/RCB_11.80GA_6000_64cc0ee3.dlp"
eseries_firmware_nvsram: "packages/N6000-880834-D08.dlp"
```

3. Download and install the latest drive firmware available for the drives installed in your block nodes from the NetApp Support site. You can either upgrade them manually or include them in the `packages/` directory of the Ansible control node, and then populate the following parameters in `eseries_storage_systems.yml` to upgrade using Ansible:

```
eseries_drive_firmware_firmware_list:
  - "packages/<FILENAME>.dlp"
eseries_drive_firmware_upgrade_drives_online: true
```

> (i) Setting `eseries_drive_firmware_upgrade_drives_online` to `false` will speed up the upgrade, but should not be done until after BeeGFS is deployed. This is because that setting requires stopping all I/O to the drives before the upgrade to avoid application errors. Although performing an online drive firmware upgrade before configuring volumes is still quick, we recommend you always set this value to `true` to avoid issues later.

4. To optimize performance, make the following changes to the global configuration:

```
# Global Configuration Defaults
eseries_system_cache_block_size: 32768
eseries_system_cache_flush_threshold: 80
eseries_system_default_host_type: linux dm-mp
eseries_system_autoload_balance: disabled
eseries_system_host_connectivity_reporting: disabled
eseries_system_controller_shelf_id: 99 # Required.
```

5. To ensure optimal volume provisioning and behavior, specify the following parameters:

```
# Storage Provisioning Defaults
eseries_volume_size_unit: pct
eseries_volume_read_cache_enable: true
eseries_volume_read_ahead_enable: false
eseries_volume_write_cache_enable: true
eseries_volume_write_cache_mirror_enable: true
eseries_volume_cache_without_batteries: false
eseries_storage_pool_usable_drives:
"99:0,99:23,99:1,99:22,99:2,99:21,99:3,99:20,99:4,99:19,99:5,99:18,99:6,
99:17,99:7,99:16,99:8,99:15,99:9,99:14,99:10,99:13,99:11,99:12"
```

> ⓘ The value specified for `eseries_storage_pool_usable_drives` is specific to NetApp EF600 block nodes and controls the order in which drives are assigned to new volume groups. This ordering ensures that the I/O to each group is evenly distributed across backend drive channels.

Define Ansible inventory for BeeGFS building blocks

After defining the general Ansible inventory structure, define the configuration for each building block in the BeeGFS file system.

These deployment instructions demonstrate how to deploy a file system that consists of a base building block including management, metadata, and storage services; a second building block with metadata and storage services; and a third storage-only building block.

These steps are intended to show the full range of typical configuration profiles that you can use to configure NetApp BeeGFS building blocks to meet the requirements of the overall BeeGFS file system.

> ⓘ In this and subsequent sections, adjust as needed to build the inventory representing the BeeGFS file system that you want to deploy. In particular, use Ansible host names that represent each block or file node and the desired IP addressing scheme for the storage network to ensure it can scale to the number of BeeGFS file nodes and clients.

Step 1: Create the Ansible inventory file

Steps

1. Create a new `inventory.yml` file, and then insert the following parameters, replacing the hosts under `eseries_storage_systems` as needed to represent the block nodes in your deployment. The names should correspond with the name used for `host_vars/<FILENAME>.yml`.

```
# BeeGFS HA (High Availability) cluster inventory.
all:
  children:
    # Ansible group representing all block nodes:
    eseries_storage_systems:
      hosts:
        netapp_01:
        netapp_02:
        netapp_03:
        netapp_04:
        netapp_05:
        netapp_06:
    # Ansible group representing all file nodes:
    ha_cluster:
      children:
```

In the subsequent sections, you will create additional Ansible groups under `ha_cluster` that represent the BeeGFS services you want to run in the cluster.

Step 2: Configure the inventory for a management, metadata, and storage building block

The first building block in the cluster or base building block must include the BeeGFS management service along with metadata and storage services:

Steps

1. In `inventory.yml`, populate the following parameters under `ha_cluster: children`:

```
    # beegfs_01/beegfs_02 HA Pair (mgmt/meta/storage building block):
    mgmt:
      hosts:
        beegfs_01:
        beegfs_02:
    meta_01:
      hosts:
        beegfs_01:
        beegfs_02:
    stor_01:
      hosts:
        beegfs_01:
        beegfs_02:
    meta_02:
      hosts:
        beegfs_01:
        beegfs_02:
    stor_02:
```

```
      hosts:
        beegfs_01:
        beegfs_02:
    meta_03:
      hosts:
        beegfs_01:
        beegfs_02:
    stor_03:
      hosts:
        beegfs_01:
        beegfs_02:
    meta_04:
      hosts:
        beegfs_01:
        beegfs_02:
    stor_04:
      hosts:
        beegfs_01:
        beegfs_02:
    meta_05:
      hosts:
        beegfs_02:
        beegfs_01:
    stor_05:
      hosts:
        beegfs_02:
        beegfs_01:
    meta_06:
      hosts:
        beegfs_02:
        beegfs_01:
    stor_06:
      hosts:
        beegfs_02:
        beegfs_01:
    meta_07:
      hosts:
        beegfs_02:
        beegfs_01:
    stor_07:
      hosts:
        beegfs_02:
        beegfs_01:
    meta_08:
      hosts:
        beegfs_02:
```

```
          beegfs_01:
        stor_08:
          hosts:
            beegfs_02:
            beegfs_01:
```

2. Create the file `group_vars/mgmt.yml` and include the following:

```
# mgmt - BeeGFS HA Management Resource Group
# OPTIONAL: Override default BeeGFS management configuration:
# beegfs_ha_beegfs_mgmtd_conf_resource_group_options:
#   <beegfs-mgmt.conf:key>:<beegfs-mgmt.conf:value>
floating_ips:
  - i1b: 100.127.101.0/16
  - i2b: 100.127.102.0/16
beegfs_service: management
beegfs_targets:
  netapp_01:
    eseries_storage_pool_configuration:
      - name: beegfs_m1_m2_m5_m6
        raid_level: raid1
        criteria_drive_count: 4
        common_volume_configuration:
          segment_size_kb:  128
        volumes:
          - size: 1
            owning_controller: A
```

3. Under `group_vars/`, create files for resource groups `meta_01` through `meta_08` using the following template, and then fill in the placeholder values for each service referencing the table below:

```
# meta_0X - BeeGFS HA Metadata Resource Group
beegfs_ha_beegfs_meta_conf_resource_group_options:
  connMetaPortTCP: <PORT>
  connMetaPortUDP: <PORT>
  tuneBindToNumaZone: <NUMA ZONE>
floating_ips:
  - <PREFERRED PORT:IP/SUBNET> # Example: i1b:192.168.120.1/16
  - <SECONDARY PORT:IP/SUBNET>
beegfs_service: metadata
beegfs_targets:
  <BLOCK NODE>:
    eseries_storage_pool_configuration:
      - name: <STORAGE POOL>
        raid_level: raid1
        criteria_drive_count: 4
        common_volume_configuration:
          segment_size_kb:  128
        volumes:
          - size: 21.25 # SEE NOTE BELOW!
            owning_controller: <OWNING CONTROLLER>
```

(i) The volume size is specified as a percentage of the overall storage pool (also referred to as a volume group). NetApp highly recommends that you leave some free capacity in each pool to allow room for SSD overprovisioning (for more information, see Introduction to NetApp EF600 array). The storage pool, `beegfs_m1_m2_m5_m6`, also allocates 1% of the pool's capacity for the management service. Thus, for metadata volumes in the storage pool, `beegfs_m1_m2_m5_m6`, when 1.92TB or 3.84TB drives are used, set this value to `21.25`; for 7.65TB drives, set this value to `22.25`; and for 15.3TB drives, set this value to `23.75`.

File name	Port	Floating IPs	NUMA zone	Block node	Storage pool	Owning controller
meta_01.yml	8015	i1b:100.127.1 01.1/16 i2b:100.127.1 02.1/16	0	netapp_01	beegfs_m1_ m2_m5_m6	A
meta_02.yml	8025	i2b:100.127.1 02.2/16 i1b:100.127.1 01.2/16	0	netapp_01	beegfs_m1_ m2_m5_m6	B
meta_03.yml	8035	i3b:100.127.1 01.3/16 i4b:100.127.1 02.3/16	1	netapp_02	beegfs_m3_ m4_m7_m8	A

File name	Port	Floating IPs	NUMA zone	Block node	Storage pool	Owning controller
meta_04.yml	8045	i4b:100.127.1 02.4/16 i3b:100.127.1 01.4/16	1	netapp_02	beegfs_m3_ m4_m7_m8	B
meta_05.yml	8055	i1b:100.127.1 01.5/16 i2b:100.127.1 02.5/16	0	netapp_01	beegfs_m1_ m2_m5_m6	A
meta_06.yml	8065	i2b:100.127.1 02.6/16 i1b:100.127.1 01.6/16	0	netapp_01	beegfs_m1_ m2_m5_m6	B
meta_07.yml	8075	i3b:100.127.1 01.7/16 i4b:100.127.1 02.7/16	1	netapp_02	beegfs_m3_ m4_m7_m8	A
meta_08.yml	8085	i4b:100.127.1 02.8/16 i3b:100.127.1 01.8/16	1	netapp_02	beegfs_m3_ m4_m7_m8	B

4. Under `group_vars/`, create files for resource groups `stor_01` through `stor_08` using the following template, and then fill in the placeholder values for each service referencing the example:

```
# stor_0X - BeeGFS HA Storage Resource
Groupbeegfs_ha_beegfs_storage_conf_resource_group_options:
  connStoragePortTCP: <PORT>
  connStoragePortUDP: <PORT>
  tuneBindToNumaZone: <NUMA ZONE>
floating_ips:
  - <PREFERRED PORT:IP/SUBNET>
  - <SECONDARY PORT:IP/SUBNET>
beegfs_service: storage
beegfs_targets:
  <BLOCK NODE>:
    eseries_storage_pool_configuration:
      - name: <STORAGE POOL>
        raid_level: raid6
        criteria_drive_count: 10
        common_volume_configuration:
          segment_size_kb: 512          volumes:
          - size: 21.50 # See note below!          owning_controller:
<OWNING CONTROLLER>
          - size: 21.50          owning_controller: <OWNING
CONTROLLER>
```

ⓘ   For the correct size to use, see Recommended storage pool overprovisioning percentages.

File name	Port	Floating IPs	NUMA zone	Block node	Storage pool	Owning controller
stor_01.yml	8013	i1b:100.127.1 03.1/16 i2b:100.127.1 04.1/16	0	netapp_01	beegfs_s1_s2	A
stor_02.yml	8023	i2b:100.127.1 04.2/16 i1b:100.127.1 03.2/16	0	netapp_01	beegfs_s1_s2	B
stor_03.yml	8033	i3b:100.127.1 03.3/16 i4b:100.127.1 04.3/16	1	netapp_02	beegfs_s3_s4	A
stor_04.yml	8043	i4b:100.127.1 04.4/16 i3b:100.127.1 03.4/16	1	netapp_02	beegfs_s3_s4	B

File name	Port	Floating IPs	NUMA zone	Block node	Storage pool	Owning controller
stor_05.yml	8053	i1b:100.127.103.5/16 i2b:100.127.104.5/16	0	netapp_01	beegfs_s5_s6	A
stor_06.yml	8063	i2b:100.127.104.6/16 i1b:100.127.103.6/16	0	netapp_01	beegfs_s5_s6	B
stor_07.yml	8073	i3b:100.127.103.7/16 i4b:100.127.104.7/16	1	netapp_02	beegfs_s7_s8	A
stor_08.yml	8083	i4b:100.127.104.8/16 i3b:100.127.103.8/16	1	netapp_02	beegfs_s7_s8	B

Step 3: Configure the inventory for a Metadata + storage building block

These steps describe how to set up an Ansible inventory for a BeeGFS metadata + storage building block.

Steps

1. In `inventory.yml`, populate the following parameters under the existing configuration:

```
meta_09:
  hosts:
    beegfs_03:
    beegfs_04:
stor_09:
  hosts:
    beegfs_03:
    beegfs_04:
meta_10:
  hosts:
    beegfs_03:
    beegfs_04:
stor_10:
  hosts:
    beegfs_03:
    beegfs_04:
meta_11:
  hosts:
    beegfs_03:
    beegfs_04:
```

```
        stor_11:
          hosts:
            beegfs_03:
            beegfs_04:
        meta_12:
          hosts:
            beegfs_03:
            beegfs_04:
        stor_12:
          hosts:
            beegfs_03:
            beegfs_04:
        meta_13:
          hosts:
            beegfs_04:
            beegfs_03:
        stor_13:
          hosts:
            beegfs_04:
            beegfs_03:
        meta_14:
          hosts:
            beegfs_04:
            beegfs_03:
        stor_14:
          hosts:
            beegfs_04:
            beegfs_03:
        meta_15:
          hosts:
            beegfs_04:
            beegfs_03:
        stor_15:
          hosts:
            beegfs_04:
            beegfs_03:
        meta_16:
          hosts:
            beegfs_04:
            beegfs_03:
        stor_16:
          hosts:
            beegfs_04:
            beegfs_03:
```

2. Under `group_vars/`, create files for resource groups `meta_09` through `meta_16` using the following

template, and then fill in the placeholder values for each service referencing the example:

```
# meta_0X - BeeGFS HA Metadata Resource Group
beegfs_ha_beegfs_meta_conf_resource_group_options:
  connMetaPortTCP: <PORT>
  connMetaPortUDP: <PORT>
  tuneBindToNumaZone: <NUMA ZONE>
floating_ips:
  - <PREFERRED PORT:IP/SUBNET>
  - <SECONDARY PORT:IP/SUBNET>
beegfs_service: metadata
beegfs_targets:
  <BLOCK NODE>:
    eseries_storage_pool_configuration:
      - name: <STORAGE POOL>
        raid_level: raid1
        criteria_drive_count: 4
        common_volume_configuration:
          segment_size_kb: 128
        volumes:
          - size: 21.5 # SEE NOTE BELOW!
            owning_controller: <OWNING CONTROLLER>
```

ⓘ For the correct size to use, see Recommended storage pool overprovisioning percentages.

File name	Port	Floating IPs	NUMA zone	Block node	Storage pool	Owning controller
meta_09.yml	8015	i1b:100.127.1 01.9/16 i2b:100.127.1 02.9/16	0	netapp_03	beegfs_m9_ m10_m13_m 14	A
meta_10.yml	8025	i2b:100.127.1 02.10/16 i1b:100.127.1 01.10/16	0	netapp_03	beegfs_m9_ m10_m13_m 14	B
meta_11.yml	8035	i3b:100.127.1 01.11/16 i4b:100.127.1 02.11/16	1	netapp_04	beegfs_m11_ m12_m15_m 16	A
meta_12.yml	8045	i4b:100.127.1 02.12/16 i3b:100.127.1 01.12/16	1	netapp_04	beegfs_m11_ m12_m15_m 16	B

File name	Port	Floating IPs	NUMA zone	Block node	Storage pool	Owning controller
meta_13.yml	8055	i1b:100.127.1 01.13/16 i2b:100.127.1 02.13/16	0	netapp_03	beegfs_m9_ m10_m13_m 14	A
meta_14.yml	8065	i2b:100.127.1 02.14/16 i1b:100.127.1 01.14/16	0	netapp_03	beegfs_m9_ m10_m13_m 14	B
meta_15.yml	8075	i3b:100.127.1 01.15/16 i4b:100.127.1 02.15/16	1	netapp_04	beegfs_m11_ m12_m15_m 16	A
meta_16.yml	8085	i4b:100.127.1 02.16/16 i3b:100.127.1 01.16/16	1	netapp_04	beegfs_m11_ m12_m15_m 16	B

3. Under `group_vars/`, create files for resource groups `stor_09` through `stor_16` using the following template, and then fill in the placeholder values for each service referencing the example:

```
# stor_0X - BeeGFS HA Storage Resource Group
beegfs_ha_beegfs_storage_conf_resource_group_options:
  connStoragePortTCP: <PORT>
  connStoragePortUDP: <PORT>
  tuneBindToNumaZone: <NUMA ZONE>
floating_ips:
  - <PREFERRED PORT:IP/SUBNET>
  - <SECONDARY PORT:IP/SUBNET>
beegfs_service: storage
beegfs_targets:
  <BLOCK NODE>:
    eseries_storage_pool_configuration:
      - name: <STORAGE POOL>
        raid_level: raid6
        criteria_drive_count: 10
        common_volume_configuration:
          segment_size_kb: 512        volumes:
          - size: 21.50 # See note below!
            owning_controller: <OWNING CONTROLLER>
          - size: 21.50            owning_controller: <OWNING
CONTROLLER>
```

ⓘ For the correct size to use, see Recommended storage pool overprovisioning percentages..

File name	Port	Floating IPs	NUMA zone	Block node	Storage pool	Owning controller
stor_09.yml	8013	i1b:100.127.1 03.9/16 i2b:100.127.1 04.9/16	0	netapp_03	beegfs_s9_s1 0	A
stor_10.yml	8023	i2b:100.127.1 04.10/16 i1b:100.127.1 03.10/16	0	netapp_03	beegfs_s9_s1 0	B
stor_11.yml	8033	i3b:100.127.1 03.11/16 i4b:100.127.1 04.11/16	1	netapp_04	beegfs_s11_s 12	A
stor_12.yml	8043	i4b:100.127.1 04.12/16 i3b:100.127.1 03.12/16	1	netapp_04	beegfs_s11_s 12	B
stor_13.yml	8053	i1b:100.127.1 03.13/16 i2b:100.127.1 04.13/16	0	netapp_03	beegfs_s13_s 14	A
stor_14.yml	8063	i2b:100.127.1 04.14/16 i1b:100.127.1 03.14/16	0	netapp_03	beegfs_s13_s 14	B
stor_15.yml	8073	i3b:100.127.1 03.15/16 i4b:100.127.1 04.15/16	1	netapp_04	beegfs_s15_s 16	A
stor_16.yml	8083	i4b:100.127.1 04.16/16 i3b:100.127.1 03.16/16	1	netapp_04	beegfs_s15_s 16	B

Step 4: Configure the inventory for a storage-only building block

These steps describe how to set up an Ansible inventory for a BeeGFS storage-only building block. The major difference between setting up the configuration for a metadata + storage versus a storage-only building block is the omission of all metadata resource groups and changing `criteria_drive_count` from 10 to 12 for each storage pool.

Steps

1. In `inventory.yml`, populate the following parameters under the existing configuration:

```
        # beegfs_05/beegfs_06 HA Pair (storage only building block):
        stor_17:
          hosts:
            beegfs_05:
            beegfs_06:
        stor_18:
          hosts:
            beegfs_05:
            beegfs_06:
        stor_19:
          hosts:
            beegfs_05:
            beegfs_06:
        stor_20:
          hosts:
            beegfs_05:
            beegfs_06:
        stor_21:
          hosts:
            beegfs_06:
            beegfs_05:
        stor_22:
          hosts:
            beegfs_06:
            beegfs_05:
        stor_23:
          hosts:
            beegfs_06:
            beegfs_05:
        stor_24:
          hosts:
            beegfs_06:
            beegfs_05:
```

2. Under `group_vars/`, create files for resource groups `stor_17` through `stor_24` using the following template, and then fill in the placeholder values for each service referencing the example:

```
# stor_0X - BeeGFS HA Storage Resource Group
beegfs_ha_beegfs_storage_conf_resource_group_options:
  connStoragePortTCP: <PORT>
  connStoragePortUDP: <PORT>
  tuneBindToNumaZone: <NUMA ZONE>
floating_ips:
  - <PREFERRED PORT:IP/SUBNET>
  - <SECONDARY PORT:IP/SUBNET>
beegfs_service: storage
beegfs_targets:
  <BLOCK NODE>:
    eseries_storage_pool_configuration:
      - name: <STORAGE POOL>
        raid_level: raid6
        criteria_drive_count: 12
        common_volume_configuration:
          segment_size_kb: 512
        volumes:
          - size: 21.50 # See note below!
            owning_controller: <OWNING CONTROLLER>
          - size: 21.50
            owning_controller: <OWNING CONTROLLER>
```

ⓘ | For the correct size to use, see Recommended storage pool overprovisioning percentages.

File name	Port	Floating IPs	NUMA zone	Block node	Storage pool	Owning controller
stor_17.yml	8013	i1b:100.127.103.17/16 i2b:100.127.104.17/16	0	netapp_05	beegfs_s17_s18	A
stor_18.yml	8023	i2b:100.127.104.18/16 i1b:100.127.103.18/16	0	netapp_05	beegfs_s17_s18	B
stor_19.yml	8033	i3b:100.127.103.19/16 i4b:100.127.104.19/16	1	netapp_06	beegfs_s19_s20	A
stor_20.yml	8043	i4b:100.127.104.20/16 i3b:100.127.103.20/16	1	netapp_06	beegfs_s19_s20	B

File name	Port	Floating IPs	NUMA zone	Block node	Storage pool	Owning controller
stor_21.yml	8053	i1b:100.127.1 03.21/16 i2b:100.127.1 04.21/16	0	netapp_05	beegfs_s21_s 22	A
stor_22.yml	8063	i2b:100.127.1 04.22/16 i1b:100.127.1 03.22/16	0	netapp_05	beegfs_s21_s 22	B
stor_23.yml	8073	i3b:100.127.1 03.23/16 i4b:100.127.1 04.23/16	1	netapp_06	beegfs_s23_s 24	A
stor_24.yml	8083	i4b:100.127.1 04.24/16 i3b:100.127.1 03.24/16	1	netapp_06	beegfs_s23_s 24	B

Deploy BeeGFS

Deploying and managing the configuration involves running one or more playbooks that contain the tasks Ansible needs to execute and bring the overall system to the desired state.

While all tasks can be included in a single playbook, for complex systems, this quickly becomes unwieldy to manage. Ansible allows you to create and distribute roles as a way of packaging reusable playbooks and related content (for example: default variables, tasks, and handlers). For more information, see the Ansible documentation for Roles.

Roles are often distributed as part of an Ansible collection containing related roles and modules. Thus, these playbooks primarily just import several roles distributed in the various NetApp E-Series Ansible collections.

> (i) Currently, at least two building blocks (four file nodes) are required to deploy BeeGFS, unless a separate quorum device is configured as a tiebreaker to mitigate any issues when establishing quorum with a two-node cluster.

Steps

1. Create a new `playbook.yml` file and include the following:

```
# BeeGFS HA (High Availability) cluster playbook.
- hosts: eseries_storage_systems
  gather_facts: false
  collections:
    - netapp_eseries.santricity
  tasks:
    - name: Configure NetApp E-Series block nodes.
```

```
        import_role:
          name: nar_santricity_management
- hosts: all
  any_errors_fatal: true
  gather_facts: false
  collections:
    - netapp_eseries.beegfs
  pre_tasks:
    - name: Ensure a supported version of Python is available on all
file nodes.
      block:
        - name: Check if python is installed.
          failed_when: false
          changed_when: false
          raw: python --version
          register: python_version
        - name: Check if python3 is installed.
          raw: python3 --version
          failed_when: false
          changed_when: false
          register: python3_version
          when: 'python_version["rc"] != 0 or (python_version["stdout"]
| regex_replace("Python ", "")) is not version("3.0", ">=")'
        - name: Install python3 if needed.
          raw: |
            id=$(grep "^ID=" /etc/*release* | cut -d= -f 2 | tr -d '"')
            case $id in
              ubuntu) sudo apt install python3 ;;
              rhel|centos) sudo yum -y install python3 ;;
              sles) sudo zypper install python3 ;;
            esac
          args:
            executable: /bin/bash
          register: python3_install
          when: python_version['rc'] != 0 and python3_version['rc'] != 0
          become: true
        - name: Create a symbolic link to python from python3.
          raw: ln -s /usr/bin/python3 /usr/bin/python
          become: true
          when: python_version['rc'] != 0
      when: inventory_hostname not in
groups[beegfs_ha_ansible_storage_group]
    - name: Verify any provided tags are supported.
      fail:
        msg: "{{ item }} tag is not a supported BeeGFS HA tag. Rerun
your playbook command with --list-tags to see all valid playbook tags."
```

```
        when: 'item not in ["all", "storage", "beegfs_ha",
"beegfs_ha_package", "beegfs_ha_configure",
"beegfs_ha_configure_resource", "beegfs_ha_performance_tuning",
"beegfs_ha_backup", "beegfs_ha_client"]'
        loop: "{{ ansible_run_tags }}"
    tasks:
      - name: Verify before proceeding.
        pause:
          prompt: "Are you ready to proceed with running the BeeGFS HA
role? Depending on the size of the deployment and network performance
between the Ansible control node and BeeGFS file and block nodes this
can take awhile (10+ minutes) to complete."
      - name: Verify the BeeGFS HA cluster is properly deployed.
        ansible.builtin.import_role:
          name: netapp_eseries.beegfs.beegfs_ha_7_4
```

> (i) This playbook runs a few `pre_tasks` that verify Python 3 is installed on the file nodes and check that the Ansible tags provided are supported.

2. Use the `ansible-playbook` command with the inventory and playbook files when you're ready to deploy BeeGFS.

   The deployment will run all `pre_tasks`, and then prompt for user confirmation before proceeding with the actual BeeGFS deployment.

   Run the following command, adjusting the number of forks as needed (see the note below):

   ```
   ansible-playbook -i inventory.yml playbook.yml --forks 20
   ```

   > (i) Especially for larger deployments, overriding the default number of forks (5) using the `forks` parameter is recommended to increase the number of hosts that Ansible configures in parallel. (For more information, see Controlling playbook execution.) The maximum value setting depends on the processing power available on the Ansible control node. The above example of 20 was run on a virtual Ansible control node with 4 CPUs (Intel® Xeon® Gold 6146 CPU @ 3.20GHz).

   Depending on the size of the deployment and network performance between the Ansible control node and BeeGFS file and block nodes, deployment time might vary.

Configure BeeGFS clients

You must install and configure the BeeGFS client on any hosts that need access to the BeeGFS file system, such as compute or GPU nodes. For this task, you can use Ansible and the BeeGFS collection.

Steps

1. If needed, set up passwordless SSH from the Ansible control node to each of the hosts you want to configure as BeeGFS clients:

```
ssh-copy-id <user>@<HOSTNAME_OR_IP>
```

2. Under `host_vars/`, create a file for each BeeGFS client named `<HOSTNAME>.yml` with the following content, filling in the placeholder text with the correct information for your environment:

```
# BeeGFS Client
ansible_host: <MANAGEMENT_IP>
# OPTIONAL: If you want to use the NetApp E-Series Host Collection's
IPoIB role to configure InfiniBand interfaces for clients to connect to
BeeGFS file systems:
eseries_ipoib_interfaces:
  - name: <INTERFACE>
    address: <IP>/<SUBNET_MASK> # Example: 100.127.1.1/16
  - name: <INTERFACE>
    address: <IP>/<SUBNET_MASK>
```

> (i) If deploying with a two subnet addressing scheme, two InfiniBand interfaces must be configured on each client, one in each of the two storage IPoIB subnets. If using the example subnets and recommended ranges for each BeeGFS service listed here, clients should have one interface configured in the range of `100.127.1.0` to `100.127.99.255` and the other in `100.128.1.0` to `100.128.99.255`.

3. Create a new file `client_inventory.yml`, and then populate the following parameters at the top:

```
# BeeGFS client inventory.
all:
  vars:
    ansible_ssh_user: <USER> # This is the user Ansible should use to
connect to each client.
    ansible_become_password: <PASSWORD> # This is the password Ansible
will use for privilege escalation, and requires the ansible_ssh_user be
root, or have sudo privileges.
The defaults set by the BeeGFS HA role are based on the testing
performed as part of this NetApp Verified Architecture and differ from
the typical BeeGFS client defaults.
```

> (i) Do not store passwords in plain text. Instead, use the Ansible Vault (see the Ansible documentation for Encrypting content with Ansible Vault) or use the `--ask-become-pass` option when running the playbook.

4. In the `client_inventory.yml` file, list all hosts that should be configured as BeeGFS clients under the `beegfs_clients` group, and then specify any additional configuration required to build the BeeGFS client kernel module.

```
  children:
    # Ansible group representing all BeeGFS clients:
    beegfs_clients:
      hosts:
        beegfs_01:
        beegfs_02:
        beegfs_03:
        beegfs_04:
        beegfs_05:
        beegfs_06:
        beegfs_07:
        beegfs_08:
        beegfs_09:
        beegfs_10:
      vars:
        # OPTION 1: If you're using the NVIDIA OFED drivers and they are
already installed:
        eseries_ib_skip: True # Skip installing inbox drivers when using
the IPoIB role.
        beegfs_client_ofed_enable: True
        beegfs_client_ofed_include_path:
"/usr/src/ofa_kernel/default/include"
        # OPTION 2: If you're using inbox IB/RDMA drivers and they are
already installed:
        eseries_ib_skip: True # Skip installing inbox drivers when using
the IPoIB role.
        # OPTION 3: If you want to use inbox IB/RDMA drivers and need
them installed/configured.
        eseries_ib_skip: False # Default value.
        beegfs_client_ofed_enable: False # Default value.
```

> ⓘ When using the NVIDIA OFED drivers, make sure that
> `beegfs_client_ofed_include_path` points to the correct "header include path" for
> your Linux installation. For more information, see the BeeGFS documentation for RDMA
> support.

5. In the `client_inventory.yml` file, list the BeeGFS file systems you want mounted at the bottom of any
   previously defined `vars`.

```
        beegfs_client_mounts:
          - sysMgmtdHost: 100.127.101.0 # Primary IP of the BeeGFS
management service.
            mount_point: /mnt/beegfs    # Path to mount BeeGFS on the
client.
            connInterfaces:
              - <INTERFACE> # Example: ibs4f1
              - <INTERFACE>
            beegfs_client_config:
              # Maximum number of simultaneous connections to the same
node.
              connMaxInternodeNum: 128 # BeeGFS Client Default: 12
              # Allocates the number of buffers for transferring IO.
              connRDMABufNum: 36 # BeeGFS Client Default: 70
              # Size of each allocated RDMA buffer
              connRDMABufSize: 65536 # BeeGFS Client Default: 8192
              # Required when using the BeeGFS client with the shared-
disk HA solution.
              # This does require BeeGFS targets be mounted in the
default "sync" mode.
              # See the documentation included with the BeeGFS client
role for full details.
              sysSessionChecksEnabled: false
```

The `beegfs_client_config` represents the settings that were tested. See the documentation included with the `netapp_eseries.beegfs` collection's `beegfs_client` role for a comprehensive overview of all options. This includes details around mounting multiple BeeGFS file systems or mounting the same BeeGFS file system multiple times.

6. Create a new `client_playbook.yml` file, and then populate the following parameters:

```
# BeeGFS client playbook.
- hosts: beegfs_clients
  any_errors_fatal: true
  gather_facts: true
  collections:
    - netapp_eseries.beegfs
    - netapp_eseries.host
  tasks:
    - name: Ensure IPoIB is configured
      import_role:
        name: ipoib
    - name: Verify the BeeGFS clients are configured.
      import_role:
        name: beegfs_client
```

> (i) Omit importing the `netapp_eseries.host` collection and `ipoib` role if you have already installed the required IB/RDMA drivers and configured IPs on the appropriate IPoIB interfaces.

7. To install and build the client and mount BeeGFS, run the following command:

```
ansible-playbook -i client_inventory.yml client_playbook.yml
```

8. Before you place the BeeGFS file system in production, we **strongly** recommend that you log in to any clients and run `beegfs-fsck --checkfs` to ensure that all nodes are reachable and there are no issues reported.

Scale beyond five building blocks

You can configure Pacemaker and Corosync to scale beyond five building blocks (10 file nodes). However, there are drawbacks to larger clusters, and eventually Pacemaker and Corosync do impose a maximum of 32 nodes.

NetApp has only tested BeeGFS HA clusters for up to 10 nodes; scaling individual clusters beyond this limit is not recommended or supported. However, BeeGFS file systems still need to scale far beyond 10 nodes, and NetApp has accounted for this in the BeeGFS on NetApp solution.

By deploying multiple HA clusters containing a subset of the building blocks in each file system, you can scale the overall BeeGFS file system independently of any recommended or hard limits on the underlying HA clustering mechanisms. In this scenario, do the following:

- Create a new Ansible inventory representing the additional HA cluster(s), and then omit configuring another management service. Instead, point the `beegfs_ha_mgmtd_floating_ip` variable in each additional cluster `ha_cluster.yml` to the IP for the first BeeGFS management service.

- When adding additional HA clusters to the same file system, ensure the following:
  - The BeeGFS node IDs are unique.

- ◦ The file names corresponding with each service under `group_vars` is unique across all clusters.
- ◦ The BeeGFS client and server IP addresses are unique across all clusters.
- ◦ The first HA cluster containing the BeeGFS management service is running before trying to deploy or update additional clusters.
- • Maintain inventories for each HA cluster separately in their own directory tree.

Trying to mix the inventory files for multiple clusters in one directory tree might cause issues with how the BeeGFS HA role aggregates the configuration applied to a particular cluster.

> ⓘ There is no requirement that each HA cluster scale to five building blocks before creating a new one. In many cases, using fewer building blocks per cluster is easier to manage. One approach is to configure the building blocks in each single rack as an HA cluster.

Recommended storage pool overprovisioning percentages

When following the standard four volumes per storage pool configuration for second generation building blocks, refer to the following table.

This table provides recommended percentages to use as the volume size in the `eseries_storage_pool_configuration` for each BeeGFS metadata or storage target:

Drive size	Size
1.92TB	18
3.84TB	21.5
7.68TB	22.5
15.3TB	24

> ⓘ The above guidance does not apply to the storage pool containing the management service, which should reduce the sizes above by .25% to allocate 1% of the storage pool for management data.

To understand how these values were determined, see TR-4800: Appendix A: Understanding SSD endurance and overprovisioning.

High capacity building block

The standard BeeGFS solution deployment guide outlines procedures and recommendations for high performance workload requirements. Customers looking to meet high capacity requirements should observe the variations in deployment and recommendations outlined here.

InfiniBand Storage Network
(BeeGFS Client/Server Traffic)

BeeGFS management and metadata on EF300 NVMe

BeeGFS CPU Clients
(used for validation)

BeeGFS storage on NLSAS IOM expansion trays

BeeGFS GPU Client
(used for validation)

Controllers

For high capacity building blocks EF600 controllers should be replaced with EF300 controllers, each with a Cascade HIC installed for SAS expansion. Each block node will have a minimal number of NVMe SSDs populated in the array enclosure for BeeGFS metadata storage and will be attached to expansion shelves populated with NL-SAS HDDs for BeeGFS storage volumes.

File node to Block node configuration remains the same.

Drive placement

A minimum of 4 NVMe SSD's are required in each block node for BeeGFS metadata storage. These drives should be placed in the outermost slots of the enclosure.

RAID 1 (2+2) Metadata
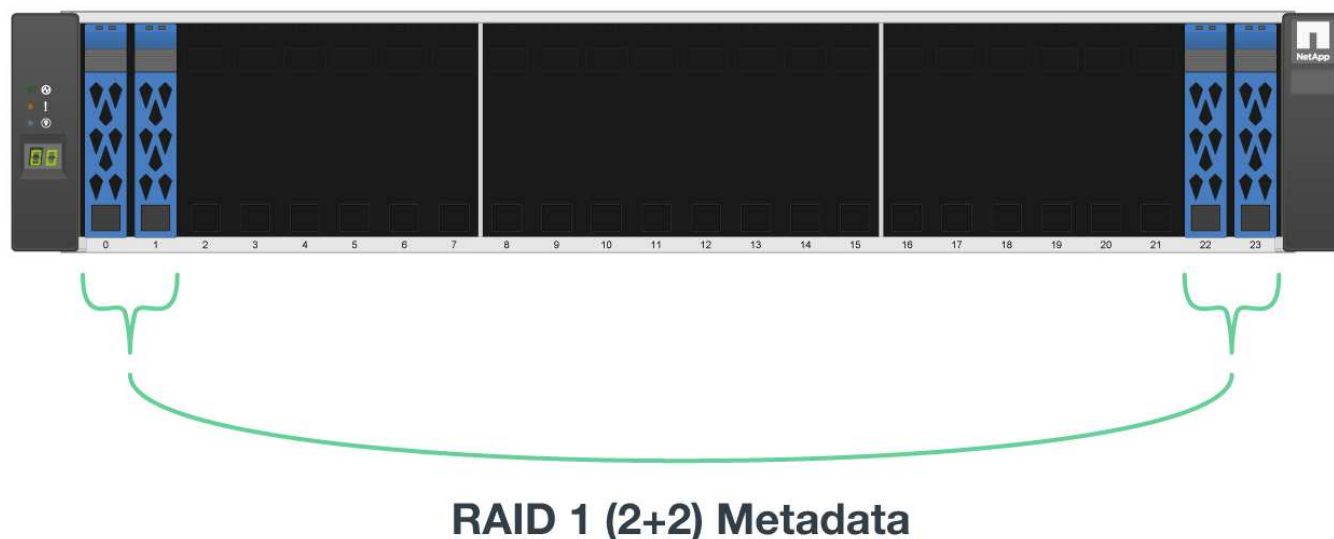
Expansion trays

The high capacity building block can be sized with 1-7, 60 drive expansion trays per storage array.

For instructions to cable each expansion tray, refer to EF300 cabling for drive shelves.