Module ||| Part II

OPENSTACK STORAGE

Block, Object and File Share

Overview

01

Block, Object and File Share

- **Block Storage**
- Definition: OpenStack Block Storage, also known as **Cinder**, provides block-level storage devices to virtual machines. These storage devices can be attached to instances (virtual machines) and are used like physical disks.
- Use Cases: Block storage is suitable for applications that require high-performance storage with low latency, such as databases and applications that require direct access to storage devices.
- Features: Cinder supports features like snapshots, cloning, and volume types, allowing users to create customized storage solutions based on their specific requirements.
- Protocols: Block storage supports iSCSI, Fibre Channel, and other storage protocols.

Block, Object and File Share

- **Object Storage**
- Definition: amounts of unstructured data, such as backups, media files, aOpenStack Object Storage, also known as **Swift**, provides a scalable and redundant object storage system. Objects, which can be files, images, videos, or any other type of data, are stored in a flat namespace and can be accessed via HTTP/HTTPS protocols.
- Use Cases: Object storage is suitable for storing large nd archival data. It is highly scalable and resilient, making it ideal for storing massive amounts of data.
- Features: Swift offers features like data replication, versioning, and support for large files, making it a reliable solution for storing and retrieving data.
- Protocols: Object storage uses HTTP/HTTPS APIs for data access.

01 Block, Object and File Share

- **Shared File Syste Storage**
- Definition: OpenStack Shared File Systems, also known as **Manila**, provides shared file storage resources that can be accessed simultaneously by multiple instances. It enables the creation of shared file systems, such as Network Attached Storage (NAS), that can be mounted by multiple virtual machines.
- Use Cases: Shared file systems are suitable for applications that require shared access to files, such as development environments, content management systems, and collaboration tools.
- Features: Manila supports features like snapshots, access control, and different backends (such as NFS and CephFS), allowing users to choose the appropriate file system for their workloads.
- Protocols: Shared file systems support NFS, CIFS/SMB, and other file-based protocols.

Storage Types – Persistent and Ephemeral Storage

- **In OpenStack, both persistent and ephemeral storage options are available for virtual machines (instances) within a cloud environment.**
- **Persistent Storage:**
- Persistent storage refers to storage that retains data even when the associated virtual machine (instance) is terminated or rebooted. It provides long-term storage capabilities and ensures that data remains intact across instance lifecycles.
- Persistent storage is suitable for applications and data that require durability and permanence. Databases, user files, and application configurations are common use cases for persistent storage.
- In OpenStack, persistent storage is typically provided through Block Storage (Cinder) volumes. These volumes can be attached to instances and persist data even if the instance is deleted or replaced. Cinder volumes can also be detached from one instance and attached to another, allowing for data mobility.

02

Storage Types – Persistent and Ephemeral Storage

- **In OpenStack, both persistent and ephemeral storage options are available for virtual machines (instances) within a cloud environment.**
- **Ephemeral Storage:**
- Ephemeral storage, also known as instance store or temporary storage, is storage that is directly attached to the virtual machine. It is temporary in nature and does not persist data across instance terminations or reboots.
- Ephemeral storage is suitable for temporary data, caching, and applications that do not require data persistence. Instances used for stateless applications or temporary processing tasks often use ephemeral storage.
- Ephemeral storage is usually provided as local disks on the physical host where the virtual machine runs. When an instance is created, it can be configured with a certain amount of local storage, and this storage is directly associated with the instance. However, once the instance is terminated, all data stored in ephemeral storage is lost.

03

Origins of Swift:

Swift was one of the first two OpenStack projects, developed jointly by ***NASA and Rackspace***.

Major changes in data consumption patterns and the concept of ***software-defined storage (SDS)*** fueled the development of object-based storage systems like Swift.

03

Origins of Swift:

Benefits of Adopting Swift:

- Scalability: Swift offers a distributed architecture for high performance and scalability.
- On-demand Provisioning: Storage can be provisioned on demand through a centralized management endpoint.
- Elasticity: Swift allows dynamic adjustment of storage resources, enabling scaling up or down as needed.

Swift Architecture:

- The architecture of OpenStack Swift is designed to provide a robust and reliable storage solution. Here are the key components of Swift's architecture:
- **Storage Nodes:**
- Storage nodes are the fundamental building blocks of Swift. These nodes store data and handle read and write requests. Multiple storage nodes are distributed across different physical servers, ensuring redundancy and fault tolerance.
- Each storage node contains hard drives or other storage media where the actual data is stored. Data is divided into smaller chunks and distributed across these storage nodes for scalability and performance.
- **Proxy Servers:**
- Proxy servers act as intermediaries between clients (applications, users) and the storage nodes. When a client sends a request to Swift, it reaches a proxy server first.
- Proxy servers are responsible for handling authentication, authorization, and routing requests to the appropriate storage nodes. They also cache metadata and help optimize client requests, improving overall system performance.

Swift Architecture:

- **Account, Container, and Object Servers:**
- Swift organizes data into accounts, containers, and objects. Account servers manage user accounts, container servers manage containers (which are similar to directories or folders), and object servers handle individual objects (files).
- Each type of server maintains metadata, such as account and container metadata, allowing for efficient management and retrieval of data.
- **Ring Structure:**
- Swift uses a ring-based architecture to distribute data across storage nodes. The ring is a data structure that maps data partitions to specific storage nodes.
- The ring allows Swift to distribute data evenly, balance the load across nodes, and ensure fault tolerance. If a storage node fails, the ring helps Swift locate replicas of the lost data on other nodes.
- **Replication and Data Integrity:**
- Swift includes built-in replication mechanisms to ensure data durability and fault tolerance. Data is replicated across multiple storage nodes, providing redundancy in case of hardware failures or other issues.
- Swift also employs checksums to verify data integrity, ensuring that stored objects remain intact and uncorrupted.

Swift Architecture:

- **Background Processes and Housekeeping:**
- **Swift runs background daemons (auditors, updaters, replicators, reapers) for housekeeping tasks on large data stores.**
- **Replication services ensure consistency and availability throughout the cluster.**
- **Metadata Usage and Indexing:**
- **Extensive usage of metadata for searching, retrieving, and indexing data in Object Storage Devices (OSDs).**
- **Metadata in OSDs are stored with objects in key-value pairs, even when objects are sliced or chunked for storage efficiency.**

Swift Architecture:

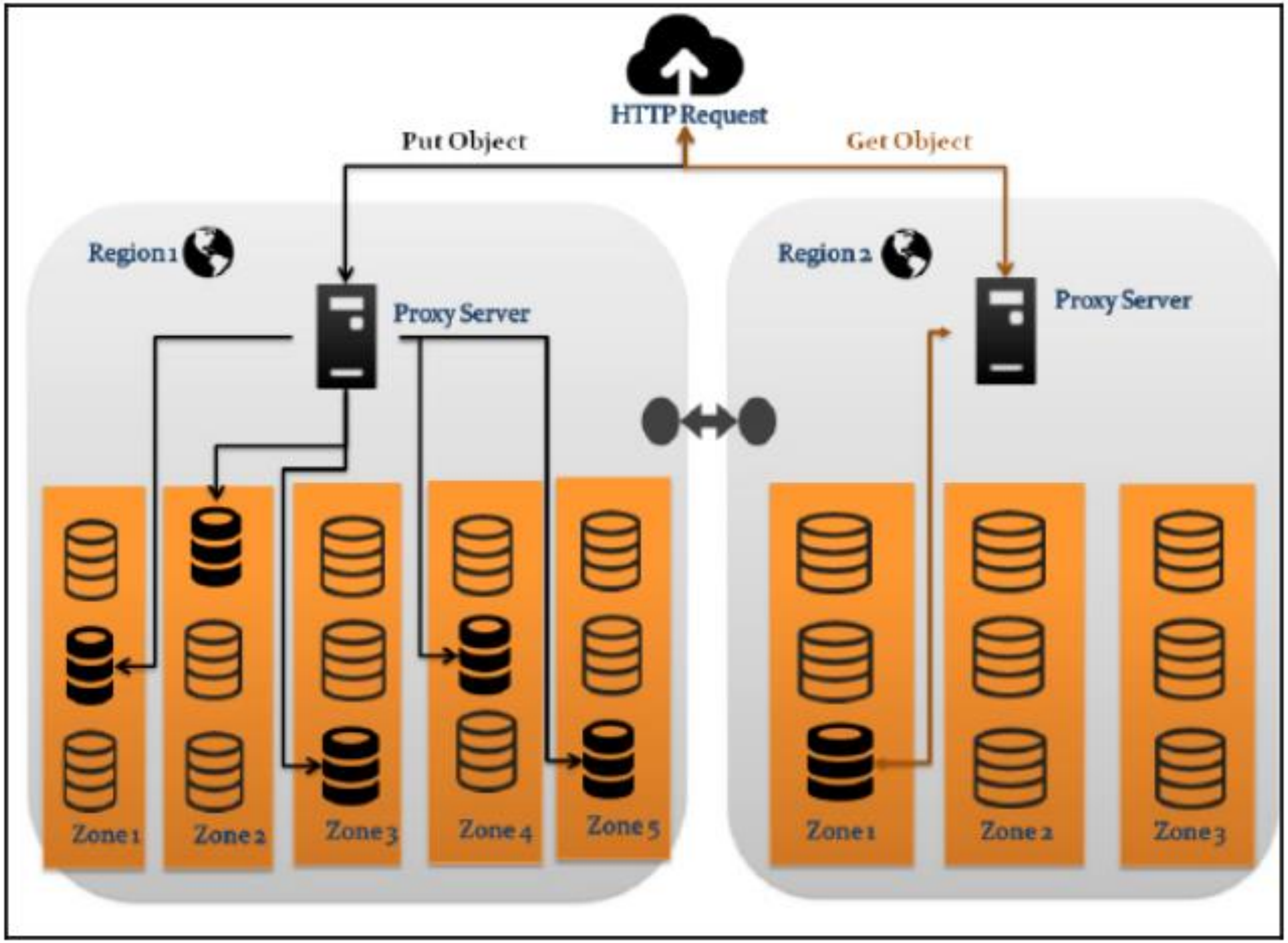
- **Rich API Access:**
- **Swift Proxy Process communicates outside the storage cluster, listening and speaking to a specific REST API.**
- **Swift provides language-specific libraries (PHP, Java, Python, etc.) for easy integration with applications, using HTTP calls to communicate with the Swift proxy.**
- **Object requests always require an authentication token, which can be configured through WSGI middleware, typically Keystone.**
- **Swift Gateways:**
- **Swift does not provide traditional interfaces like CIFS or NFS.**
- **Swift Filesystem Gateways act as an additional layer to interact with storage interfaces, enabling integration with traditional applications.**

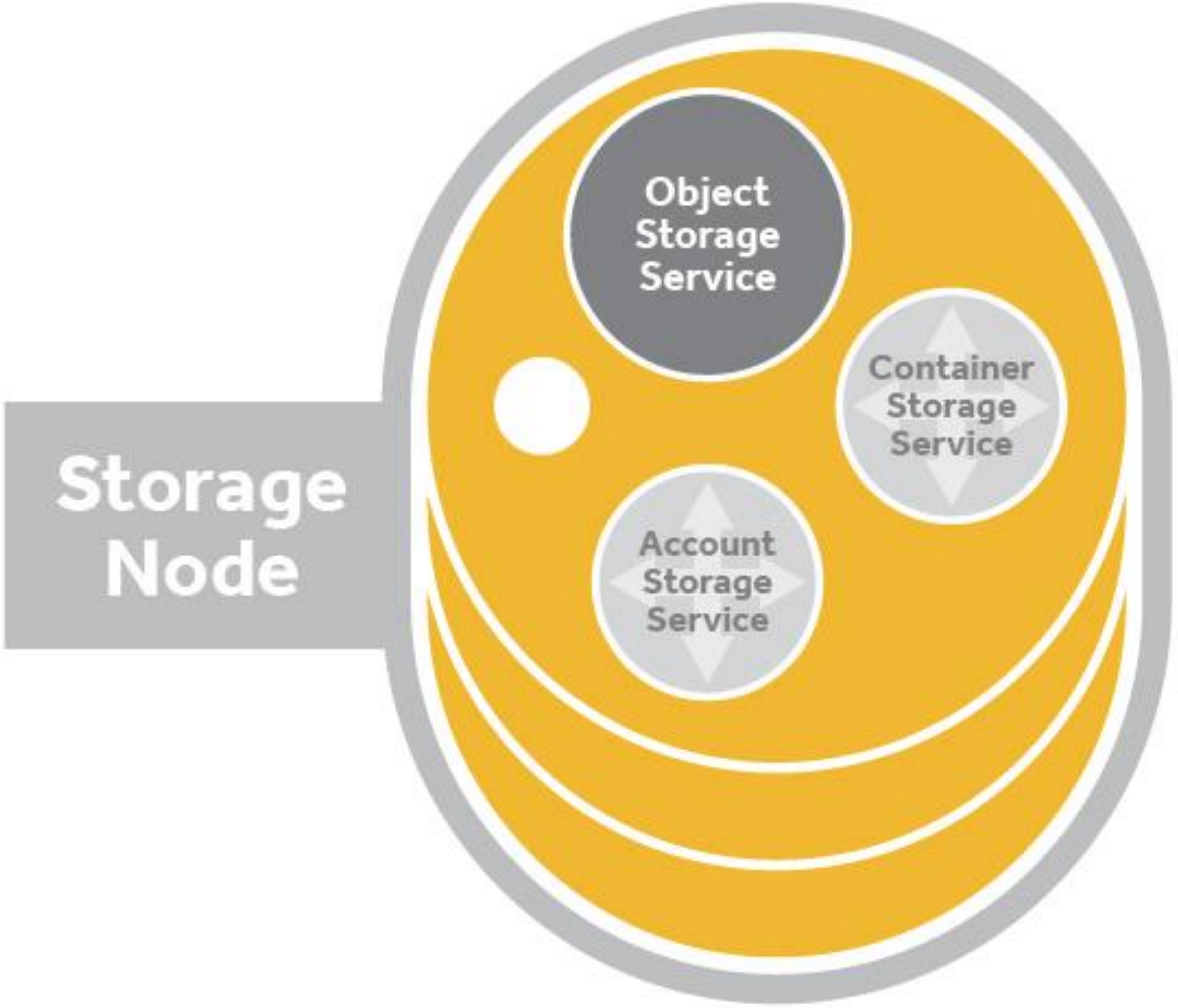
Physical design Considerations

- **Data Durability and Availability:**
- Swift focuses on ensuring data durability and availability as its primary goal.
- By default, Swift uses a replica count of three, spreading data across two redundant replicas to increase availability, albeit requiring more storage capacity.
- **Dedicated Storage Network:**
- A dedicated network for storage is selected to maintain logical network design organization and reduce network load.
- Dedicating a separate storage handler helps mitigate potential network congestion, especially in scenarios where large volumes of data need to be transferred remotely.
- **Consideration for Bandwidth Usage:**
- Bandwidth usage between storage servers and proxies is a crucial consideration.
- Swift's physical design and data organization must account for efficient data transfer, especially in urgent situations where immediate data replication is required.

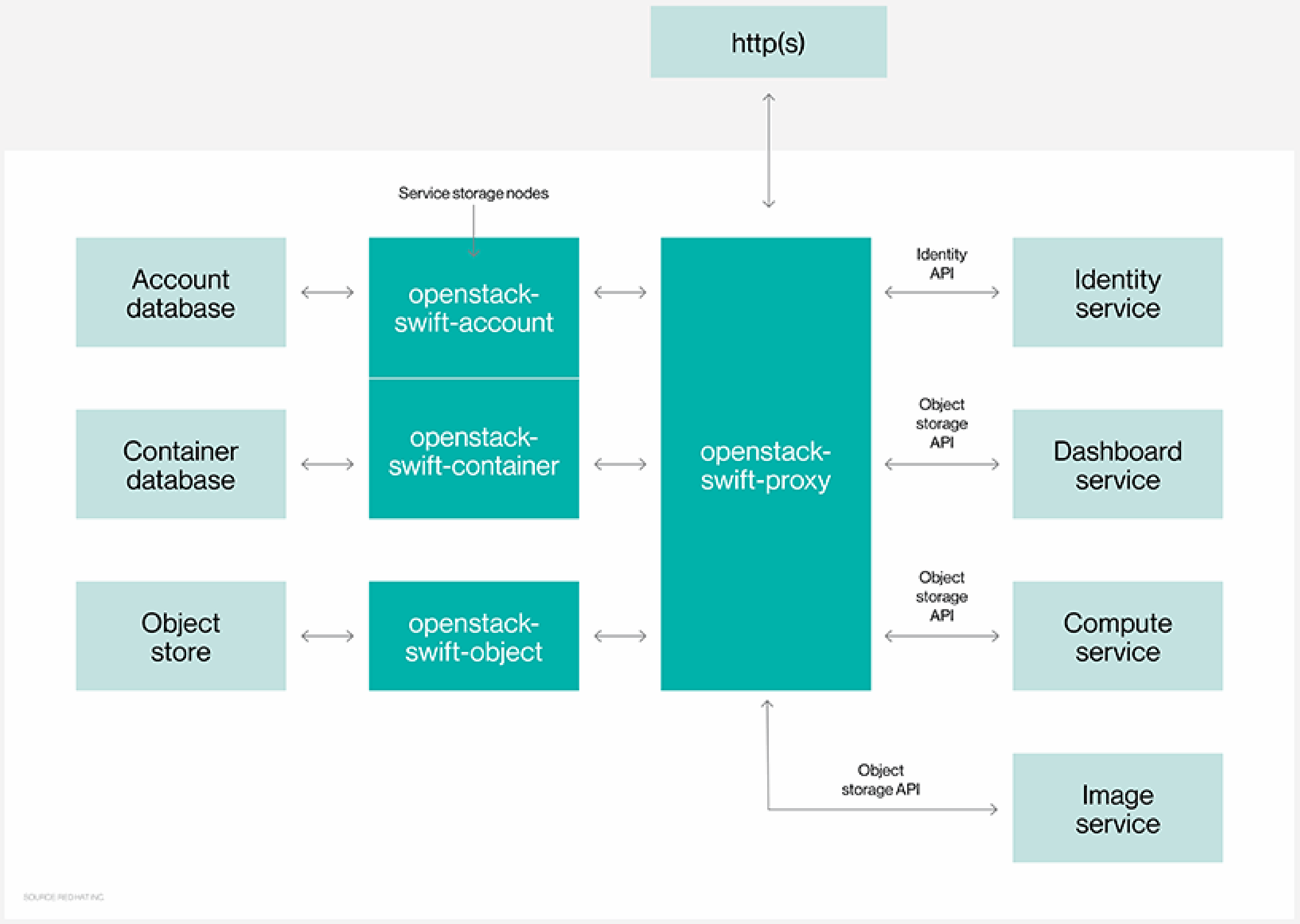
Physical design Considerations

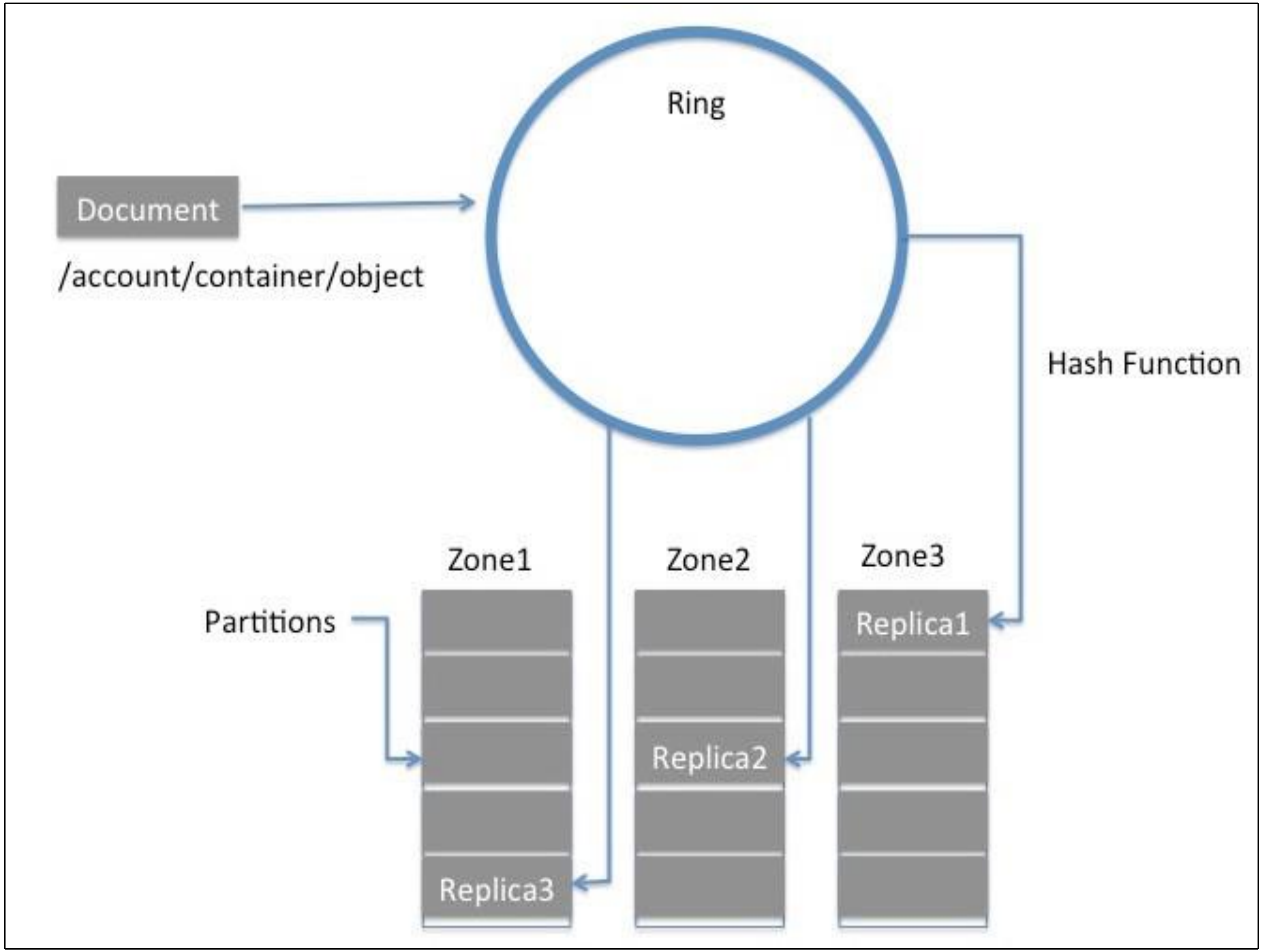
- Hierarchical Organization:
- Region: Geographically distributed data can be held in multiple nodes placed in different regions, forming a Multi-Region Cluster (MRC).
- Swift supports read/write affinity, favoring data closer to the reader and asynchronously transferring data to other regions.
- Zone: Regions encapsulate zones, defining the availability level. Zones group hardware items (e.g., racks or storage nodes) to isolate hardware failures from neighboring components.
- Logical Organization of Storage Nodes:
- Storage nodes extend the storage abstraction from regions to zones, defining clusters that run Swift processes.
- Each storage node stores an account, a container, object data, and associated metadata.
- Storage Device:
- The storage device is the smallest unit of the Swift data stack.
- It can be the internal storage node's device or an external stack of disks in a drive enclosure.


HTTP Request
Put Object
Get Object
Region 1
Proxy Server
Region 2
Proxy Server
Zone 1
Zone 2
Zone 3
Zone 4
Zone 5
Zone 1
Zone 2
Zone 3


Storage Node
Object Storage Service
Container Storage Service
Account Storage Service

OpenStack Object Storage Service




Ring
Document
/account/container/object
Hash Function
Zone1
Zone2
Zone3
Partitions
Replica1
Replica2
Replica3

Swift Ring

- Swift uses rings to manage data in the cluster.
- Rings define how data is organized and stored in Swift.
- **Logical Layout:**
  - In Swift, the logical layout of data is mapped to a path based on account, container, and object hierarchy.
  - Accounts are like tenants, containers are like folders, and objects are like files in traditional file systems.
- **Physical Mapping:**
  - Swift's rings map the logical layout of data to physical locations in the cluster.
  - Each tenant can have multiple containers, and objects belong to containers.
- **Separate Rings:**
  - Swift maintains separate rings for accounts, containers, and objects.
  - The Swift proxy uses the appropriate ring to find the physical location of the storage construct.
- **Ring Building:**
  - Rings are built using a tool called the "swift-ring-builder."
  - This tool takes an inventory of the Swift storage cluster and divides it into partitions or slots.
  - The following is the generic format of the ring builder command:
- **#swift–ring–builder <builder_file> create <qart_qower> <reqlicas><min_part_hours>**
- The <builder_file> can be one of account.builder, container.builder, or object.builder

Storage policy and erasure coding

- **Erasure Coding as Redundancy Policy:**
  - Swift allows defining redundancy policies other than replication.
  - Erasure coding is one such policy that uses parity to recreate lost data.
- **How Erasure Coding Works:**
  - When a user uploads a document with erasure coding, Swift breaks the data into segments.
  - PyECLib is then used to encode these segments into erasure-coded fragments.
- **Data Retrieval Process:**
  - To retrieve the data, the proxy server asks participating storage nodes for encoded fragments.
  - These fragments are decoded to reconstruct the object and sent back to the client.
- **Configurability:**
  - The library used for erasure coding is configurable as a plugin to PyECLib.
  - This allows flexibility in choosing the specific erasure coding mechanism.
- **Overhead Comparison:**
  - Erasure coding has an overhead of around 40% of the data size.
  - This overhead is much smaller compared to a replication-based scheme.

Storage policy and erasure coding

- **High Availability and Replication:**
  - Traditional Swift storage maintains multiple copies of data in different regions, zones, nodes, and disks for high availability.
  - This replication strategy provides redundancy but results in high storage costs.
- **Default Replication Level:**
  - By default, Swift stores three copies of each object for redundancy.
  - This approach reduces available storage to one-third of the actual capacity.
- **Storage Cost Challenge:**
  - Maintaining multiple copies is expensive, especially for critical data.
  - Affording an expensive Swift storage system can be a challenge.
- **Swift Storage Policy:**
  - Swift introduces storage policies to address the cost of redundancy.
  - The goal is to reduce storage costs while maintaining data availability and performance.
- **Differential Data Replication:**
  - Swift's storage policy uses a differential data replication strategy.
  - This strategy aims to provide redundancy in a more cost-effective manner compared to simply creating multiple identical copies

Swift Hardware Requirements:

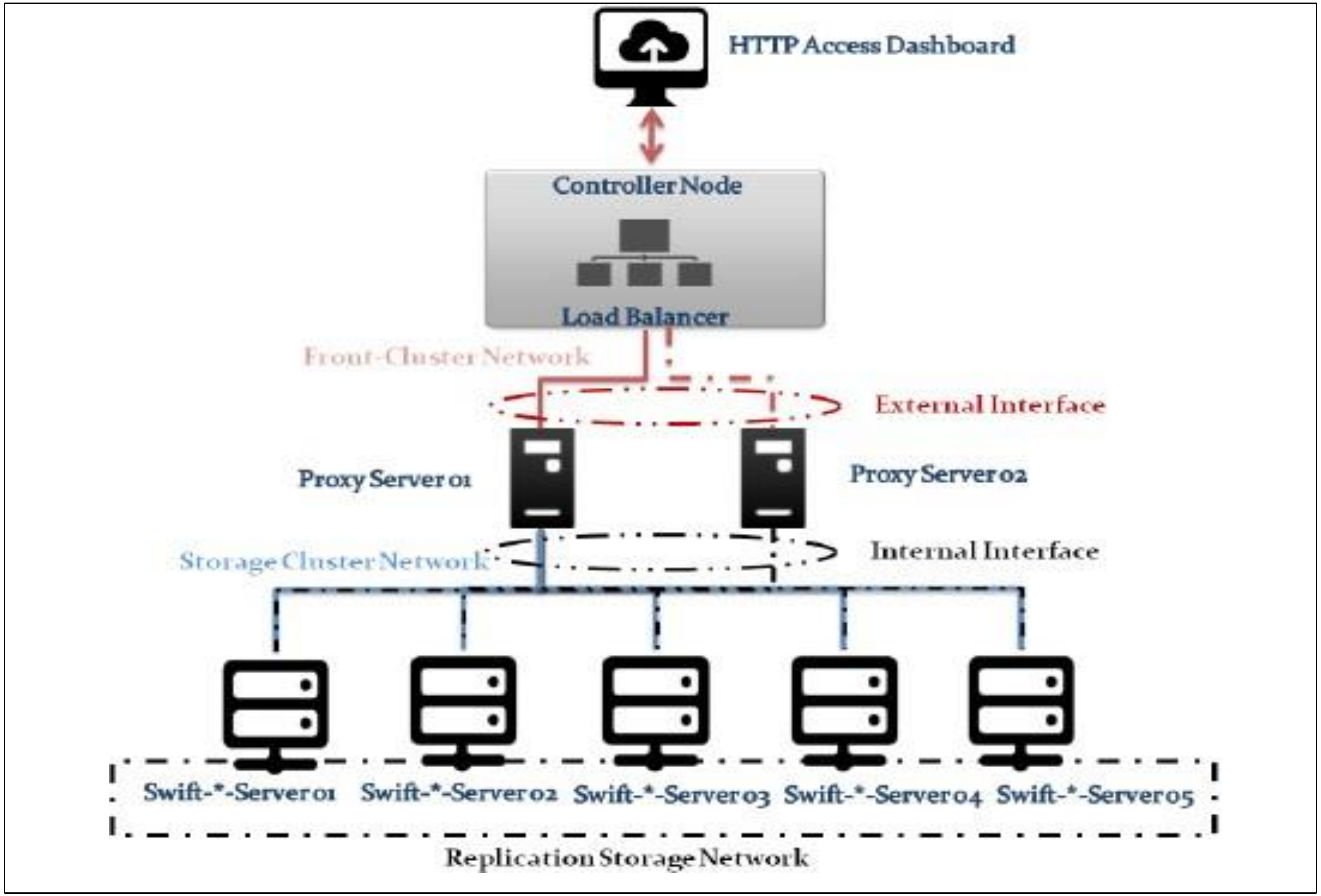
- **Logical Grouping:**
  - Group containers, accounts, and objects logically to form a storage tier.
  - Racks with storage tiers sharing a physical point of failure are grouped into the same zone.
- **Deployment Example:**
  - Object storage: 50 TB
  - Cluster replica: 3
  - Swift filesystem: XFS
  - Hard drive: 2.5 TB
  - Hard drive slots: 30 per chassis
- **Calculations for Storage Nodes:**
  - Total storage with 3 replicas: 50 TB * 3 = 150 TB
  - Accounting for metadata overhead: 150 TB * 1.0526 = 158 TB
  - Number of hard drives needed: 158 TB / 2.5 TB = 64 drives
  - Total storage nodes: 64 / 30 = 2.1333 → 3 nodes (rounded up)

Swift Hardware Requirements:

- **Node Roles and Recommendations:**
  - Proxy server: Handles client requests, may increase CPU utilization.
  - Storage nodes: Perform disk I/O operations; recommend more CPUs for Swift processes (replication, auditing).
  - CPU Calculation: Assuming 2 GHz processors with a 3:4 ratio of cores to drives, need 11.25 cores.
- **RAM and Filesystem:**
  - Swift recommends XFS filesystem, caching nodes into RAM.
  - RAM for caching: Start with 2 GB per server for faster object access.
- **Cost/Performance Consideration:**
  - Account and container servers can use SSDs for speed during data localization.
  - Object storage servers can use 6 TB SATA/ATA disks for a cost-effective solution.

Swift Network Design:

- **Dedicated Networks:**
  - Design assumes an additional network specifically for the storage system.
  - Swift is envisioned as a large infrastructure, resembling a big house with small rooms in an OpenStack deployment.
- **Extended Swift Network:**
  - **Front-Cluster Network:**
    - Proxy servers use this network for communication with external clients.
    - It handles traffic forwarding for external API access to the cluster.
  - **Storage Cluster Network:**
    - Enables communication between storage nodes and proxies.
    - Supports inter-node communication across multiple racks in the same region.
  - **Replication Network:**
    - Dedicates a network segment specifically for replication-related communication between storage nodes.


HTTP Access Dashboard
Controller Node
Load Balancer
Front-Cluster Network
External Interface
Proxy Server 01
Proxy Server 02
Storage Cluster Network
Internal Interface
Swift-*-Server 01
Swift-*-Server 02
Swift-*-Server 03
Swift-*-Server 04
Swift-*-Server 05
Replication Storage Network

CINDER BLOCK STORAGE SERVICE

CINDER

an OpenStack Community Project

Using Cinder Block Storage Service in OpenStack:

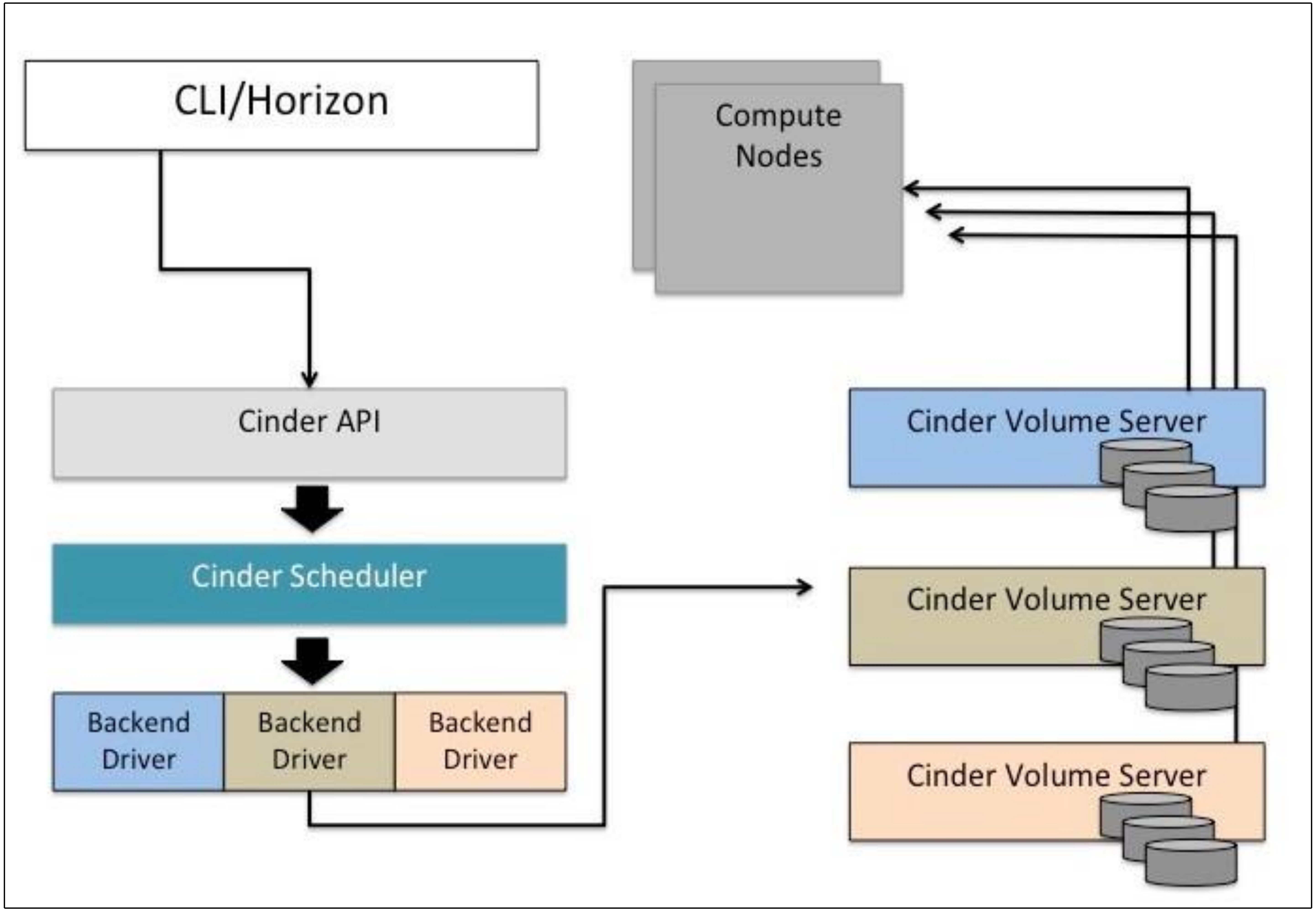
- Purpose of Cinder:
  - Cinder provides persistent storage management for virtual machine hard drives.
  - Virtual machines with Cinder volumes can be live-migrated and evacuated easily.
- Evolution of Block Storage:
  - Originally part of OpenStack Nova, block storage evolved into a separate service called Cinder.
  - Cinder volumes expose a raw block of storage, acting as an additional hard drive within virtual machines.
- Volume Attachment and Usage:
  - To use a Cinder volume in a virtual machine, It must be partitioned, laid with a filesystem, and mounted on the virtual machine's filesystem hierarchy.
- Protocols Used:
  - Cinder uses iSCSI, NFS, and fiber channels to present the block device to virtual machines.
- Quota Management:
  - Cinder helps manage quotas by limiting tenant usage.
  - Quotas can be set for total storage, snapshots, and volumes.

Volume Attachment and Usage

- Partitioning:
  - Logical Organization: Partitioning allows you to logically organize the storage space within a Cinder volume. You can create multiple partitions within a volume, each serving a specific purpose, such as separating system and user data.
- Mounting:
  - Access to Data: Mounting a file system on a partition makes the data within that partition accessible to the operating system. This is necessary for applications and services running on the VM to read and write data to the Cinder volume.
- File System Installation:
  - Data Organization: Installing a file system on a partition structures the storage space and provides a way to organize and access data. It defines how data is stored, retrieved, and managed on the Cinder volume.

Using Cinder Block Storage Service in OpenStack: Cinder Service Components:

- Cinder service consists of:
  - Cinder API server: Interacts with the outside world using REST.
  - Cinder scheduler: Chooses the volume server for hosting new volumes.
  - Cinder volume server: Nodes hosting the volumes.
- Storage Operations Behind the Scenes:
  - Steps for attaching Cinder volumes to Nova instances:
    - Create a Cinder volume specifying name and size.
    - Use the volume-attach command to attach the volume to a Nova instance.
    - Mark the volume as available to the Nova instance using the libvirt library.


CLI/Horizon
Compute Nodes
Cinder API
Cinder Scheduler
Backend Driver
Backend Driver
Backend Driver
Cinder Volume Server
Cinder Volume Server
Cinder Volume Server

Attaching Cinder Volumes to Nova Instances:

– Create a Cinder Volume:
  - Command: **#cinder create --display_name volume1 1**
  - This command creates a new Cinder volume with the name "volume1" and a specified size of 1 GB.
  - By default, the volume driver used is LVM (Logical Volume Manager) over iSCSI (Internet Small Computer System Interface).
  - The created logical volume (LV) is stored in the volume group (VG) named "cinder-volumes."

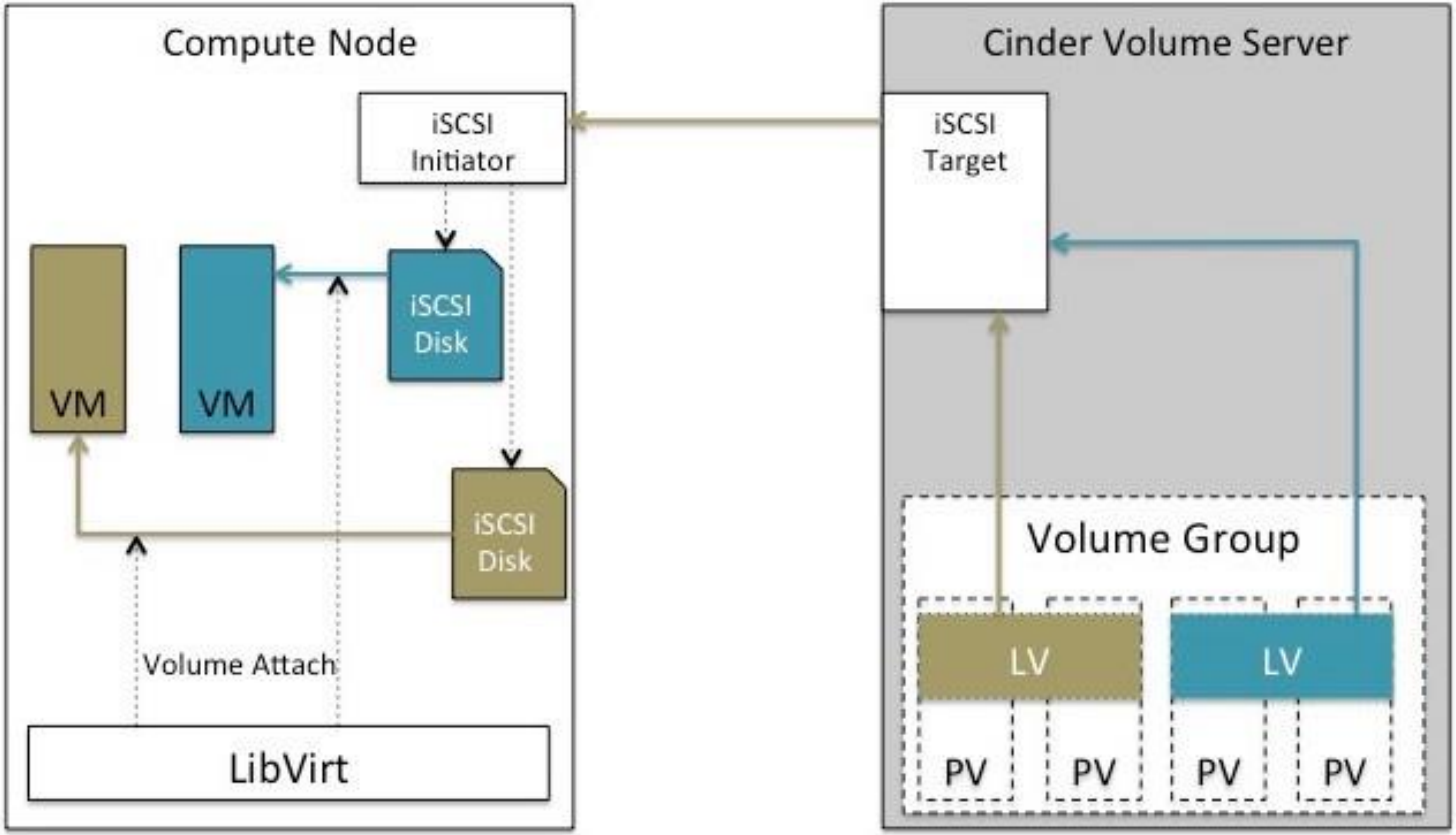
– Attach Cinder Volume to Nova Instance:
  - Command: **#nova volume-attach server_ID volume_ID device_name**
  - This command attaches the previously created Cinder volume to a Nova (Compute service in OpenStack) instance.
  - server_ID is the ID of the Nova instance to which you want to attach the volume.
  - volume_ID is the ID of the Cinder volume you created earlier.
  - device_name is the name of the device that will be seen inside the Nova instance.

Attaching Cinder Volumes to Nova Instances:

– Mark Volume as Available to Nova Instance:
  - This step involves using the libvirt library, which is a toolkit used for managing virtualized platforms.
  - Libvirt is utilized to present the iSCSI drive (the Cinder volume) as an additional block device to the virtual machine (Nova instance).
  - The iSCSI Qualified Name (IQN) is generated to uniquely identify the storage resource, and libvirt handles the presentation of this resource to the compute node where the Nova instance is running.

– Note:

– The same concept is followed for various use cases supported by Cinder, such as creating a volume from an image and booting an instance from a volume.


Compute Node
iSCSI Initiator
VM
VM
iSCSI Disk
iSCSI Disk
Volume Attach
LibVirt
Cinder Volume Server
iSCSI Target
Volume Group
LV
LV
PV
PV
PV
PV

Cinder Block Storage Service - Summary

- ❖ Cinder is the block storage service that provides persistent block-level storage devices for use with virtual machines (VMs). Cinder volumes play a crucial role in OpenStack's architecture by decoupling the storage from the compute resources, allowing for more flexibility and scalability.
- ❖ When a VM is terminated or moved to another host, any data stored on local disks is typically lost. Cinder volumes provide a way to store data persistently, even when VMs are moved or terminated.
- ❖ Cinder volumes support features like snapshots and cloning. Snapshots allow you to create point-in-time copies of volumes for backup or testing purposes, and cloning enables the rapid provisioning of new volumes based on existing ones.
- ❖ Cinder volumes can be attached to and detached from VMs dynamically. This enables administrators to move volumes between VMs, providing a level of abstraction and flexibility in managing storage resources.

Amal Jyothi College of Engineering

Thank You

navyamolkt@amaljyothi.ac.in