What's next after CSI?

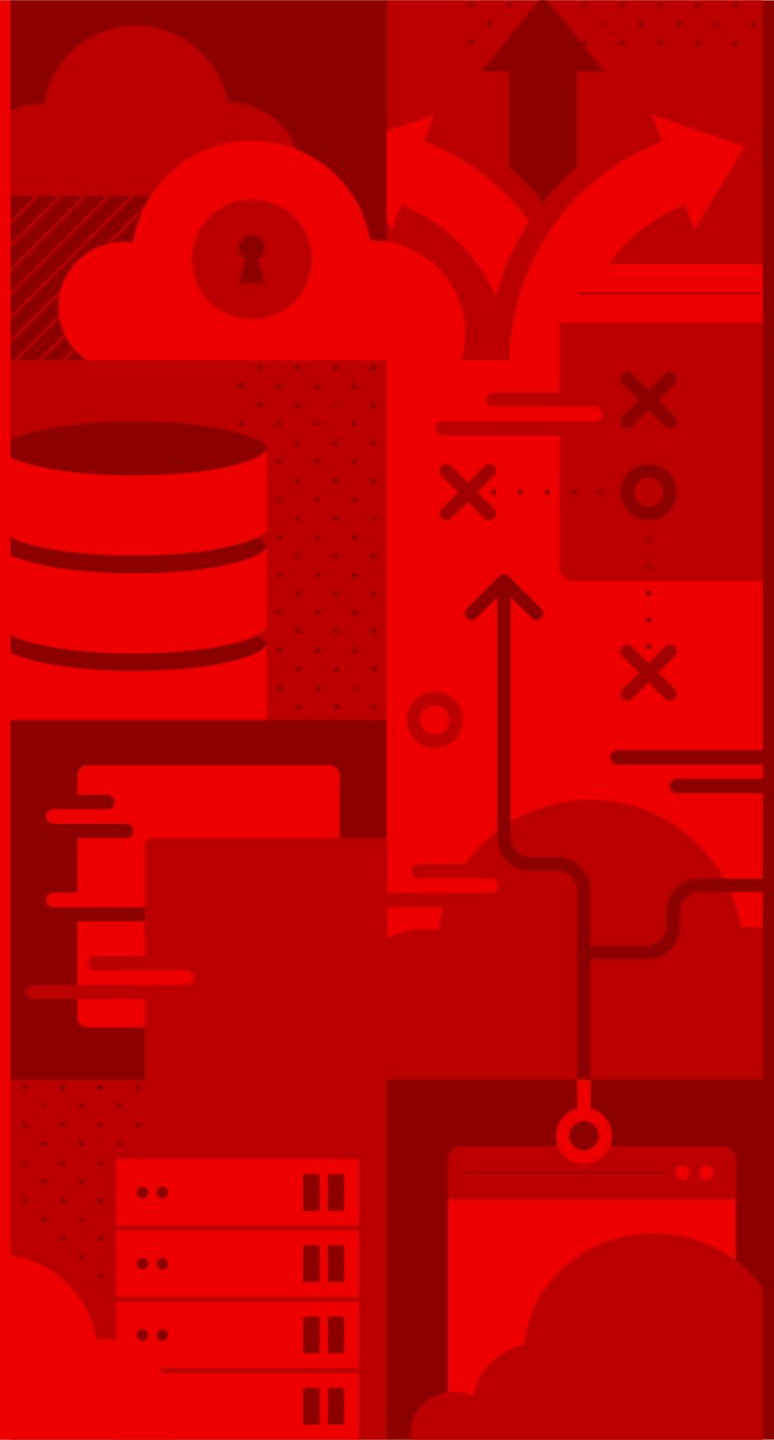
An introduction to Object Storage for Kubernetes. Moving beyond file and block storage in Kubernetes

Krish Chowdhary
Software Engineer

Jiffin Tony Thottan
Senior Software Engineer

What we'll discuss today

- Introduction to Object Storage
- CSI and Kubernetes
- History of Object Stores in Kubernetes
- Need for COSI
- COSI Architecture

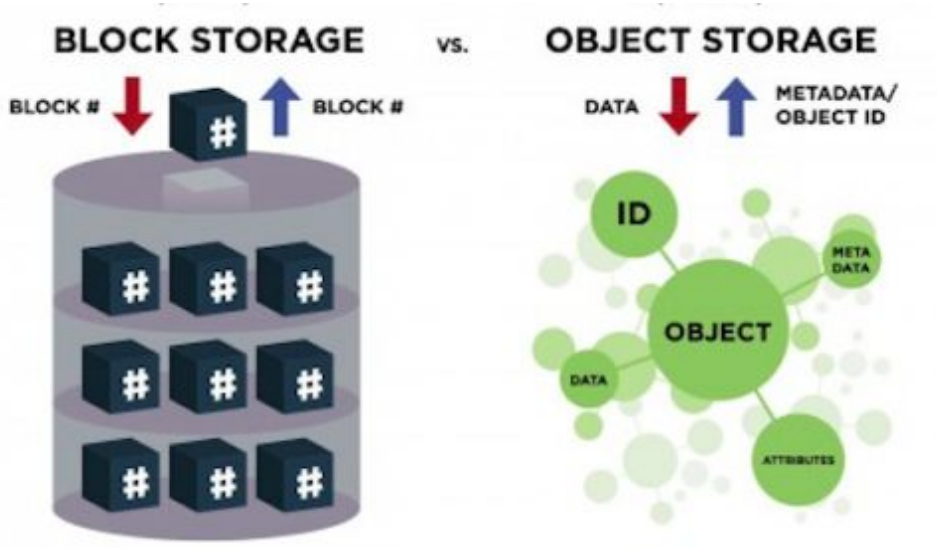


What is Object Storage? How does it differ from file + block storage?

Red Hat

What is Object Storage?

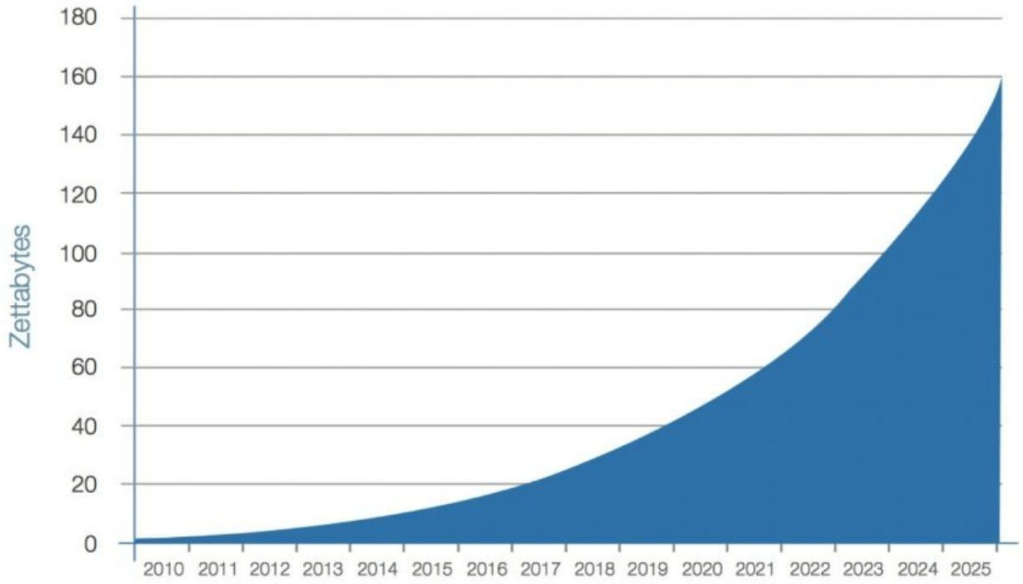
- Data is broken into small discrete units known as objects and stored in a flat architecture
- It can be accessed by simple network APIs
- Organized into logical containers which store the objects, commonly known as buckets
- It is cost efficient and can scale into extremely large quantities while maintaining quick access



Red Hat

What is the use case?

- Network focused, software defined storage is very flexible.
- Object storage is well suited for static data, always-connected mobile devices, deep learning and med-reduce analysis.
- There is no definitive protocol for consumption and creation of objects
- We can enforce more granular permissions based on bucket policies and namespacing.



Source:
https://cloudian.com/blog/trends-driving-massive-coming-demand-for-object-storage/
https://cloudian.com/blog/the-common-foundations-of-kubernetes-and-object-storage/

What is the role of the Container Storage Interface (CSI)?

What is CSI?

- Container Storage Interface provides platform to expose block and file storage systems.
- Prior to CSI, connecting to new volumes plugins needed to be directly a part of core Kubernetes. CSI allowed vendors to move this logic into separate drivers. Some popular CSI drivers expose Amazon EBS, Ceph, or Google Cloud Store.
- This meant more options for storage, and it made core Kubernetes more secure and reliable.

Source:
https://kubernetes.io/blog/2019/01/15/container-storage-interface-ga/

CSI Terminology

Key terms and concepts in the Container Storage Interface

Storage Class (SC)

Storage classes provide a way more Kubernetes admins to describe different classes of storage

Persistent Volume (PV)

Persistent volumes are pieces of storage that are provisioned statically by an administrator or dynamically through a SC

Persistent Volume Claim (PVC)

Persistent volume claims is a request for access to storage by a user. PVCs consume PV resources, they specify size and access modes.

Source:
https://kubernetes.io/blog/2019/01/15/container-storage-interface-ga/

History of Object Bucket Provisioning in Kubernetes

Motivation

- Provide a generic, dynamic provisioning API to consume object store
- App Pods can access the bucket in the underlying object-store like a PVC
- Implement k8s controller automation design with pluggable provisioners
- Present similar user/admin experience for new and existing buckets
- Be vendor agnostic (S3, RGW, Swift, GCS , etc..)
- It won't orchestrate/manage the backend object store, need to handle separately

Libbucket Provisioner (DEPRECATED as Feb 2020)

- Golang library wrapping a k8s controller
- It uses two custom resources to abstract bucket and claim/request made on it
- Consumed by Rook, Noobaa as external vendor/library
- Library handles:
  - watches on bucket claims/requests
  - reconciles/retries the requests
  - creates the artifacts such as configmap and secret consumed by app pod
  - deletes k8s resources generated on behalf of the claim

Terminologies

- Object Bucket Claim (OBC) is similar in usage to a PVC, it is namespaced and references a storage class which defines the object store provisioner.
- Object Bucket (OB) is equivalent to PV and is cluster scoped, typically not visible to end users, and it contains info pertinent to the provisioned bucket. OBs maintain persistent state information that may be needed by provisioners
- Storage Class (SC) referenced by the OBC may contain vendor specific keys, including region, bucket owner, credentials, etc. It also holds the reclaim policy for the buckets

Different Strategies

Creation of OBC

- Greenfield : Provisioning will result in creating new buckets
- Brownfield: Provisioning will consume existing buckets

Deletion of OBC

- Delete reclaim policy: results in deletion of bucket and its contents
- Retain reclaim policy: keeps the buckets and its contents

Features inherited from Object Store Vendor

- There are opaque fields in OBC and SC for the provisioner in which additional features provided by the storage vendor can be added
- OBC
  - additionalConfig: string map part of OBC CRD
  - User/Bucket specific features such as quota, acls, notification
- StorageClass
  - parameters: string map part of SC CRD
  - ObjectStore specific features such as Endpoint, Region can be included here

Limitations

- Must written in Go
- It’s more k8s specific
- Provisioner have rebuild with each library update
- Multiple provisioner artifacts inhibits scalability
- Access policies for Buckets were missing
- Limited access to API options, only supported Create/Delete

Container Object Store Interface (COSI)

COSI Terminology

What is the core terms used in the COSI project?

Bucket Class (BC)
Cluster scoped resource containing fields defining the provisioner for new buckets

Bucket (B)
Cluster scoped resource, referenced by a Bucket Request, contains connection information

Bucket Access (BA)
Cluster scoped resource for granting bucket access

Bucket Request (BR)
Namespaced resource which represents a request for a new or existing bucket

Bucket Access Class (BAC)
Cluster scoped resource which defines the provisioner and references ConfigMap policy

Bucket Access Request (BAR)
Namespaced resource requesting access to an existing bucket

Source:
https://github.com/kubernetes/enhancements/tree/master/keps/sig-storage/1979-object-storage-support

COSI vs. CSI

A brief comparison between CR types

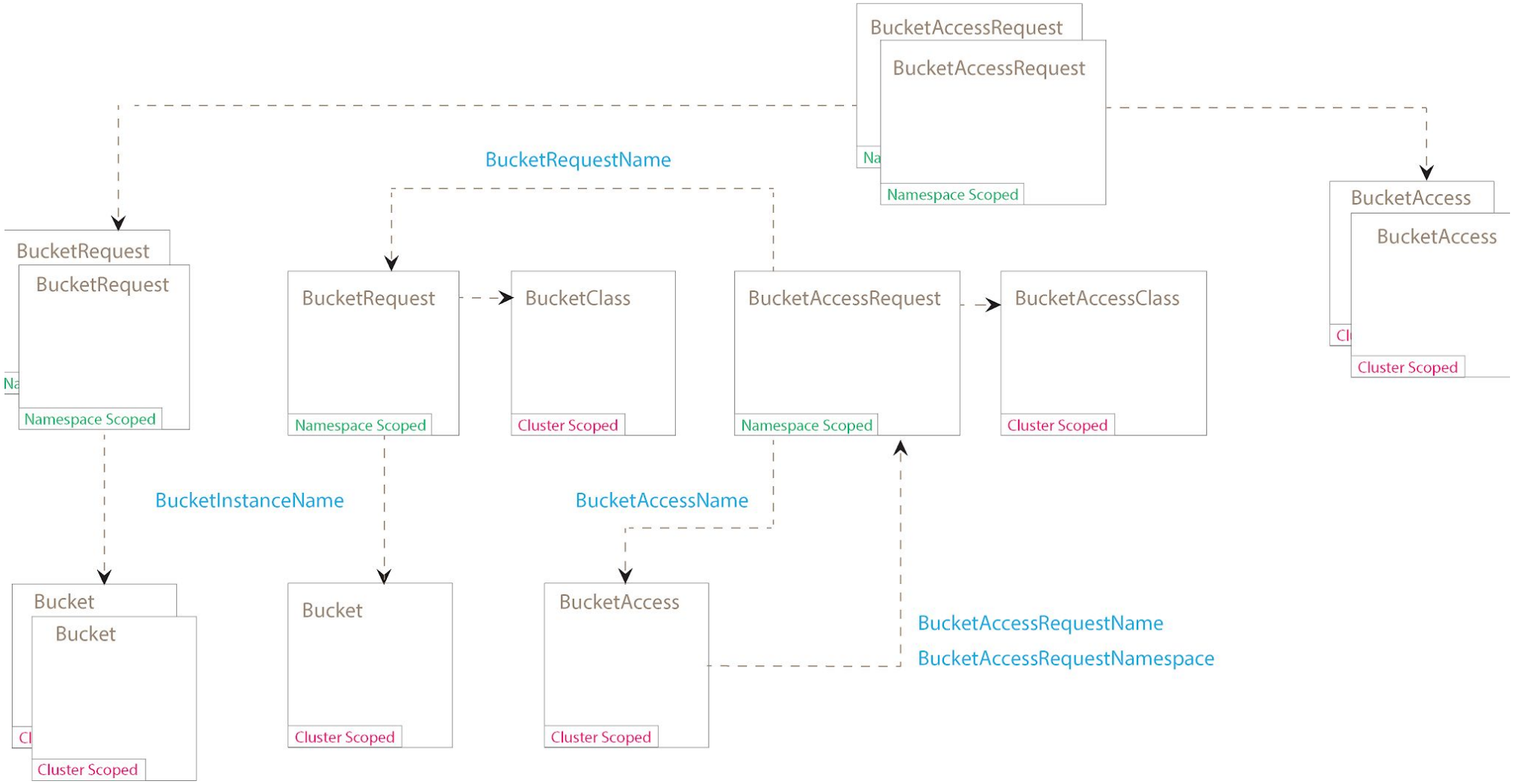
- Bucket Class
- Bucket
- Bucket Request
- Bucket Access Request

COSI emphasises the granularity of bucket access policies through BACs, BAs and BARs.

- Storage Class
- Persistent Volume
- Persistent Volume Claim

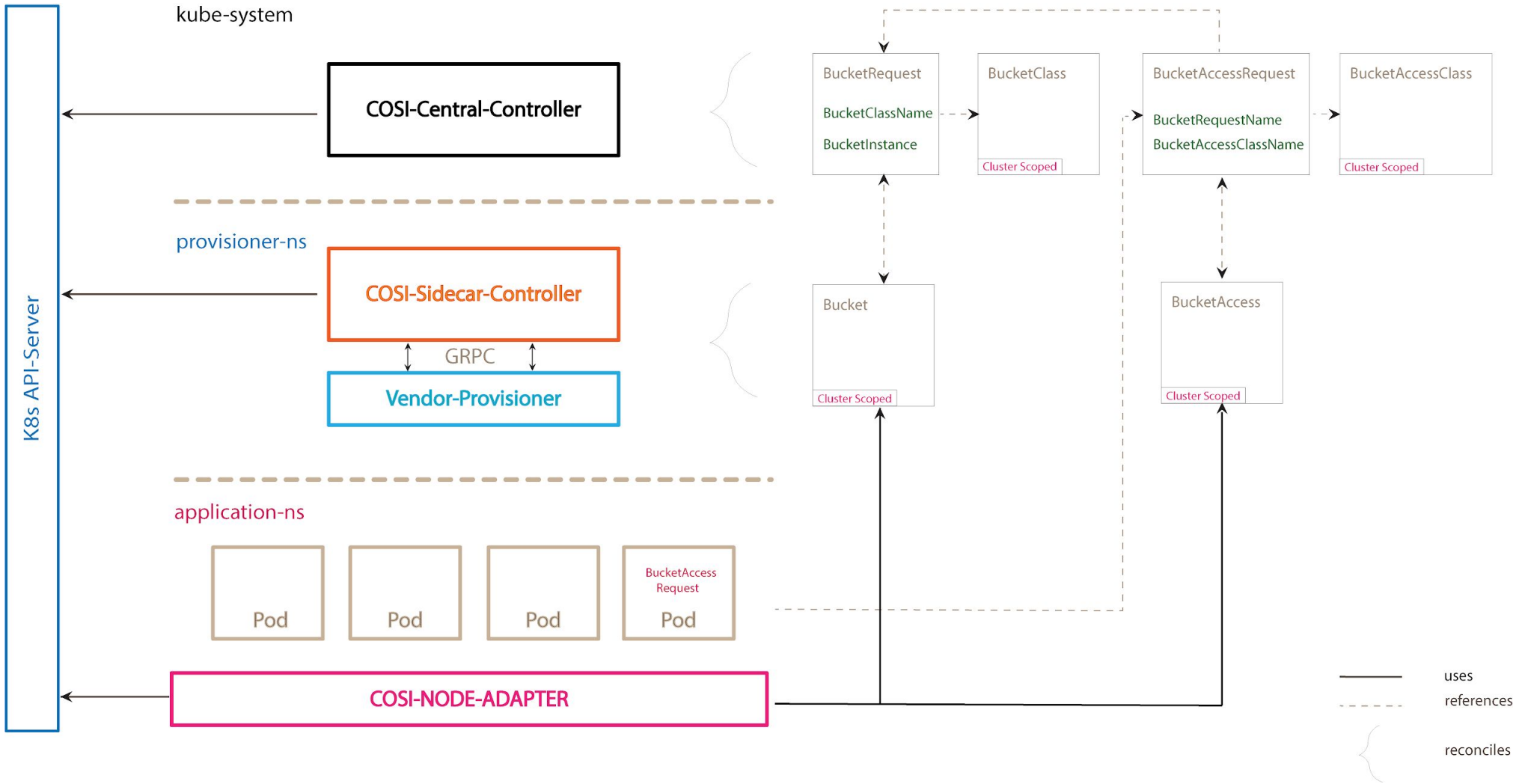
CSI has less granular access policies, and instead allows for the predefined access modes of: ReadWriteOnce, ReadOnlyMany, and ReadWriteMany.

COSI Object Relationships



Source:
https://github.com/kubernetes/enhancements/tree/master/keps/sig-storage/1979-object-storage-support#object-relationships

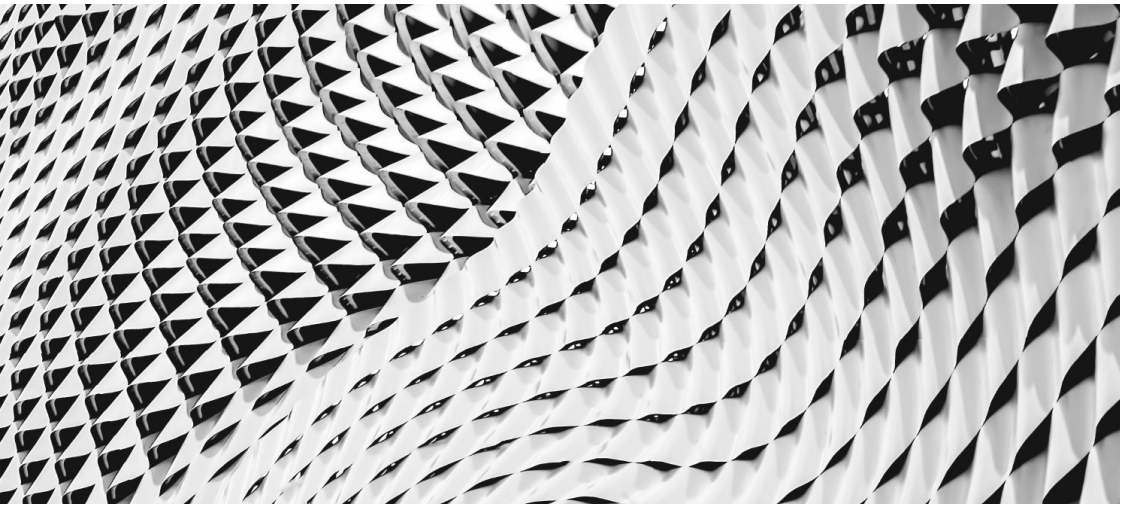
COSI Topology



Source:
https://github.com/kubernetes/enhancements/tree/master/keps/sig-storage/1979-object-storage-support#topology

COSI Preparation

Greenfield and brownfield consumption



- Admin creates the Bucket Classes to interface with the bucket provisioners
- Admin creates the Bucket Access Class defining the provisioner
- Brownfield Note: For brownfield access, the admin needs to directly create the backend buckets and Bs

	User	Central Controller	Sidecar Controller	Vendor Provisioner	Kube Admin
					CREATE BucketClass
T1	CREATE BucketRequest				
T2		CREATE Bucket			
T3			Call CreateBucket		
T4				CreateBucket	
T5					

Create Bucket Workflow

This workflow describes the automation supporting creating a new (greenfield) backend bucket. Although not pictured, the cosi node adapter is responsible for mounting the secret onto the app pod.

	User	Central Controller	Sidecar Controller	Vendor Provisioner	Kube Admin
					Copy Bucket for new NS
T1	Create BR for assigned B				
T2	Create BAR referring to BR				
T3			Create BA for the BAR		
T4	Using BAR in Pod				
T5					

Sharing COSI Created Bucket (Brownfield)

This is the greenfield to brownfield access use case, when COSI created the Bucket CR and backing bucket.

	User	Central Controller	Sidecar Controller	Vendor Provisioner	Kube Admin
	Delete BucketRequest				
T1		Mark Bucket Unavailable			
T2			Wait for Bucket Release		
T3			Call driver DeleteBucket		
T4				DeleteBucket	
T5					

Delete Bucket

This workflow describes the automation designed for deleting a Bucket instance and optionally the related backend bucket.

The delete workflow is described as a synchronous, but it will likely be asynchronous to accommodate potentially long delete times.

	User	Central Controller	Sidecar Controller	Node Adapter	Kube Admin
					Create Brownfield Bucket
T1	Create BR referencing B				
T2	Create BAR references the BR				
T3		Creates BA based on BAR and BAC			
T4			Calls driver to grant access to Bucket		
T5				Mounts the secret onto the App Pod	

Grant Bucket Access (Brownfield)

This workflow describes the automation supporting granting access to an existing backend bucket.

	User	Central Controller	Sidecar Controller	Node Adapter	Kube Admin
	Deletes the BAR				
T1		Delete the BA (finalizer blocks)			
T2			Calls driver revoke access		
T3		Removes BAR and BA finalizers		Kubelet calls NodeUnstageVolume	
T4				Removes finalizers from BA/BR	
T5					

Revoke Bucket Access

This workflow describes the automation supporting revoking access to an existing backend bucket, and the deletion of the cluster-scoped BucketAccess instance.

	User	Central Controller	Sidecar Controller	Node Adapter	Kube Admin
					Deletes BA (finalizer blocks)
T1			Calls driver to revoke access		
T2					Deletes app pod
T3				Kubelet calls NodeUnpublishVolume	
T4					
T5					

Delete Bucket Access

The most common scenario is likely the case where tokens are compromised and the admin needs to stop their use.

In this case the admin may terminate the app pod(s) and delete the BucketAccess instances.

Thank you

Join the #sig-storage-cosi channel on the Kubernetes Slack to get involved, or join our weekly meetings on Thursday @ 6PM GMT

github.com/kubernetes-sigs/container-object-storage-interface-spec


Krish Chowdhary, @krishchow_

Jiffin Tony Thottan


Jeff Vance


Sidhartha Mani


Srini Brahmarotu

Robert Rati


Tejas Parikh

Scott Creeley


Jon Cope

Kubernetes Slack: #sig-storage-cosi

Red Hat