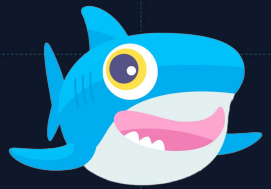
blockxchange [BXC]

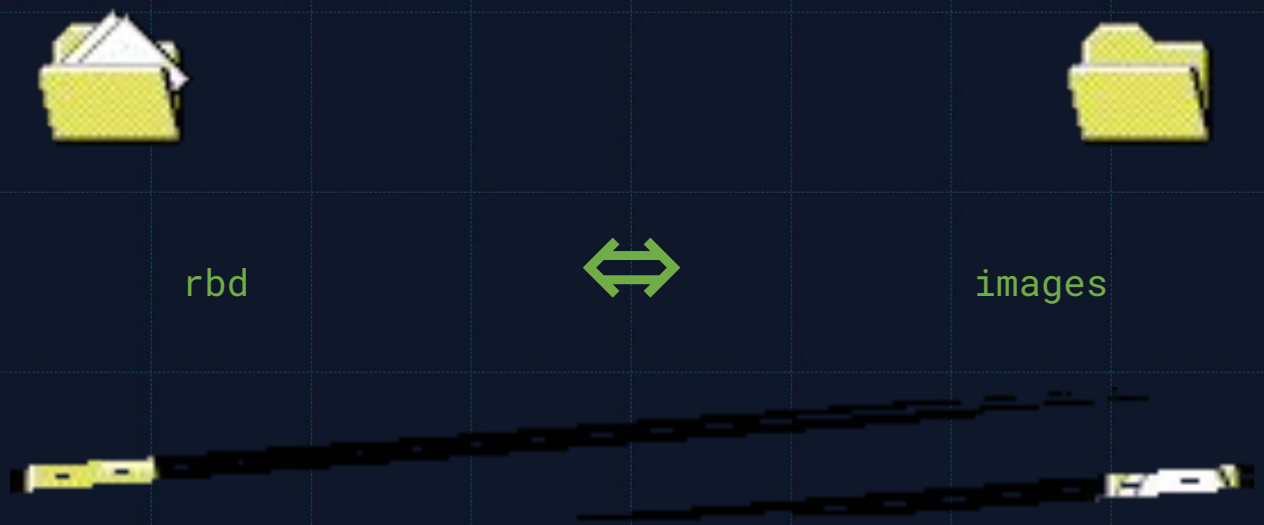
An Architecture for RBD Imaging and Snapshot Exports

Khayyam Saleem + Alisha KC

Storage @ DigitalOcean



[blockxchange?]



[Imaging RBD Volumes]

[The Problem with Seeding RBD Volumes with images]

Clone?

Golden parent images accumulate indefinitely — can't be deleted while children reference them, so cluster capacity grows monotonically with the image catalog.

Clone + Flatten?

Rewrites every parent object into the child (3x replicated); a burst of provisions from the same 50 GB image generates ~15 TB of write traffic competing with production I/O.

`curl -L http://images.server.com/ubuntu.raw | rbd import - rbd/disk-01` ?

Every provision pulls the full image from the image store, with zero deduplication

[Our Solution Architecture]

Image on-demand

RBD reads fault through to the backing image store, only when the guest actually touches a page. Instant provision, available for use.

RBD import-only live migrations with an HTTP source-spec

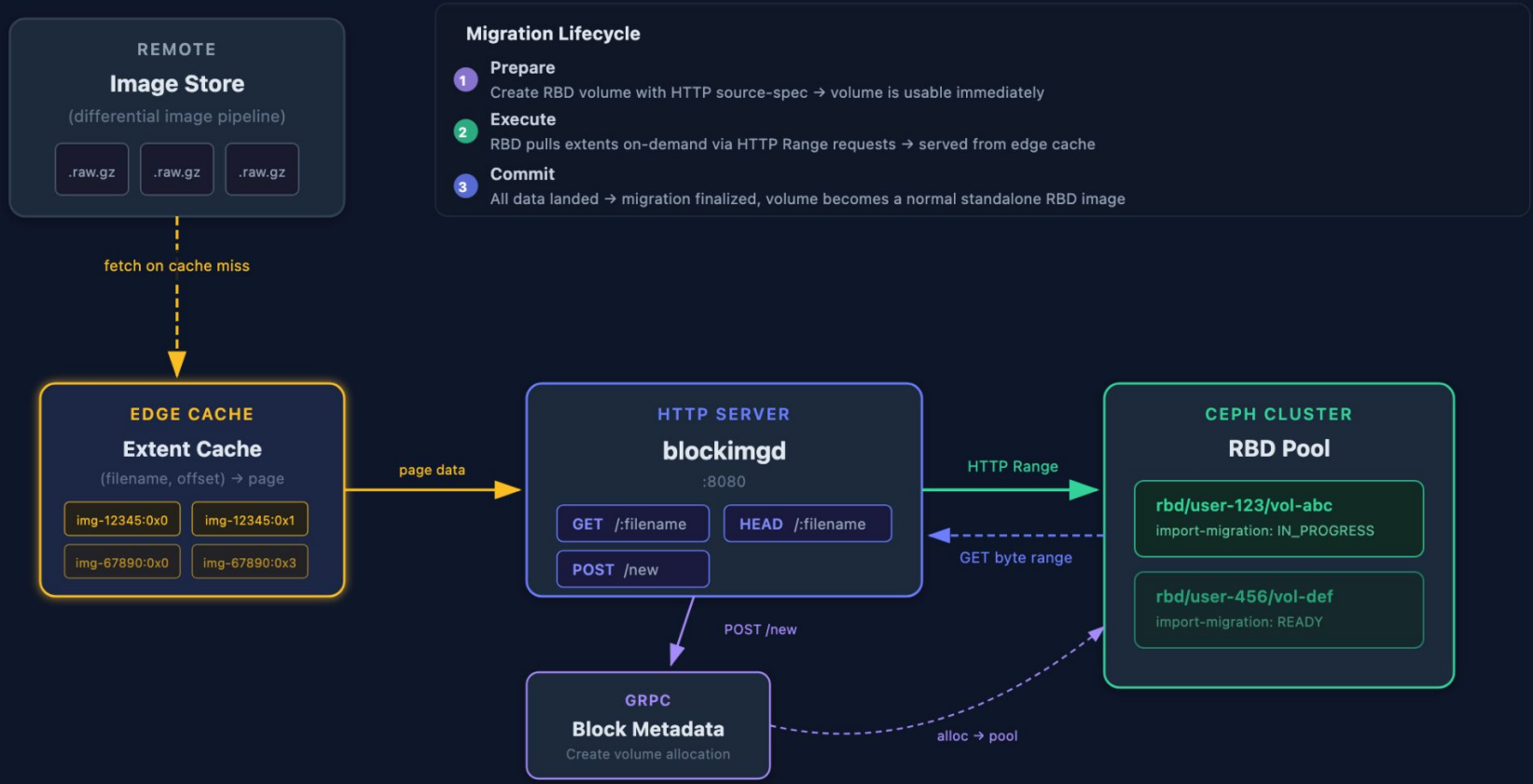
Ceph natively pulls extents from an HTTP endpoint on-demand (migration_prepare_import). Volume is usable while data streams in the background — no upfront bulk transfer.

Extent caching at the edge

Cache keyed by (filename, offset) deduplicates reads of the same image.
Horizontally scalable.

blockimg — Image-on-Demand Provisioning

RBD import-migration with HTTP source-spec + edge extent cache



```
// 1. Build the HTTP source-spec that tells RBD where to pull data from
sourceSpec := fmt.Sprintf(`{
"type": "raw",
"stream": {"type": "http", "url": "%s"}
}`, imageURL)

// 2. Connect to the Ceph cluster and open the RBD pool
conn, _ := rados.NewConnWithClusterAndUser(clusterName, "client.admin")
conn.ReadDefaultConfigFile()
conn.Connect()
defer conn.Shutdown()

ioctx, _ := conn.OpenIOContext("rbd")
defer ioctx.Destroy()

// 3. Prepare: create the RBD image backed by the HTTP source (volume is usable after this)
imageName := fmt.Sprintf("%d/%s", userID, volumeID)
rbd.MigrationPrepareImport(sourceSpec, ioctx, imageName, rbd.NewRbdImageOptions())

// 4. Execute: RBD pulls all extents from blockimgd via HTTP Range requests
rbd.MigrationExecute(ioctx, imageName)

// 5. (Optional) Poll migration status while execute runs
status, _ := rbd.MigrationStatus(ioctx, imageName)
// status.State == rbd.MigrationImageExecuted

// 6. Commit: finalize migration, volume becomes a standalone RBD image
rbd.MigrationCommit(ioctx, imageName)
```

[Disaster-Proof Differential Backups of RBD Volumes]

[The Problem with Traditional Ceph Backups]

Ceph Snapshots Are Not True Backups

Snapshots live **in-place on the same OSDs**. A pool or cluster failure destroys both the primary data and its snapshots simultaneously.

Full Image Exports Are Wasteful

Exporting the entire image every time wastes bandwidth and storage. A **500 GB volume** with only **2 GB of daily churn** still requires 500 GB transferred every backup cycle.

For Change-Tracking, External Solutions Add Complexity

External dirty-bitmap solutions (like QEMU CBT or NBD metadata) introduce additional moving parts and new failure modes to the infrastructure.

[Our Solution Architecture]

Export rbd image layers to a differential system

A **differential system** in this context is just any store that supports storing metadata relationships between layers, and storing blocks and offsets. Ex. OCI image stores (dockerhub).

Export data using go-ceph utilities

Our control plane for Ceph RBD is already written in Golang; go-ceph helps us integrate with the rest of our service architecture and build the data pipeline.

Orchestrate the data pipeline at scale with Temporal

To be able to manage + track progress for thousands of backups, we leverage temporal and its durable execution strategy.

[go-ceph – Building the Data Pipeline for Differential RBD Backups]

[1] Create a "Snapshot" + List Parents

The layer of the rbd image that we will use to track changed blocks

[2] DiffIterate

Walk the object map, collect all dirty extents since the parent layer into a channel.

[3] rbd image ReadAt

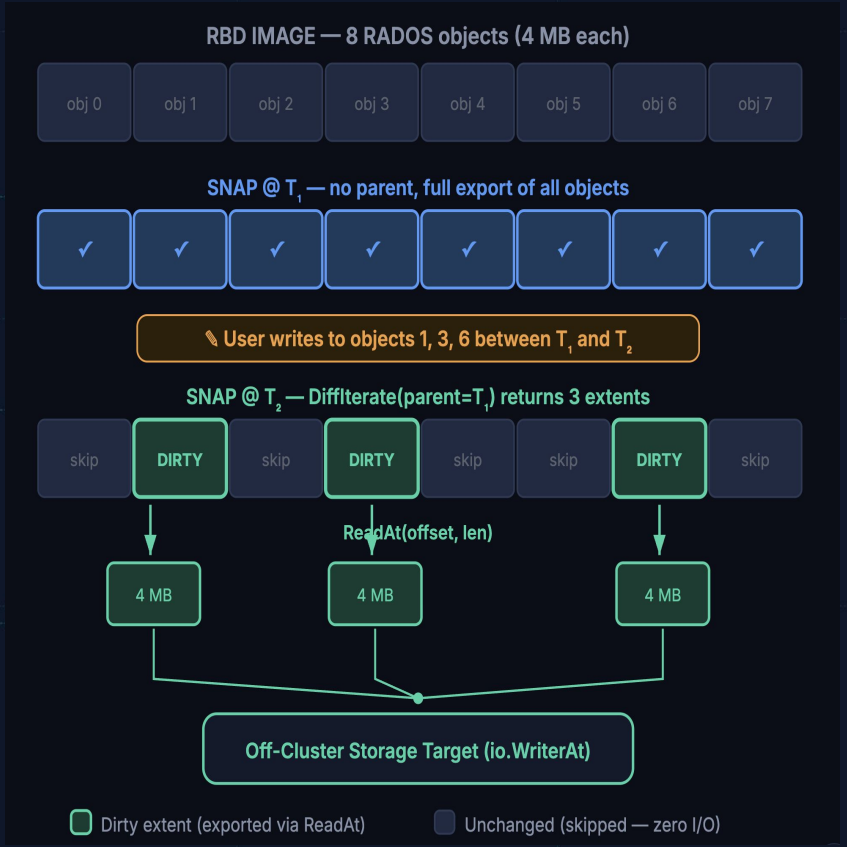
Using the extents from DiffIterate, configure a reader that will be used as a source of data extents for the export step.

[4] Concurrent CopyExtents

Fan out N goroutines reading from the extent channel and writing to any `io.WriterAt` target.

[5] Orchestrate

`ListSnapshots` → `CreateSnapshot` → `DiffIterate` → `CopyExtents` → `Commit` → `DeleteSnapshot` (rotate)



[go-ceph - Backup Pipeline]

[1] DISCOVER: Collect Dirty Extents

```
snap.DiffIterate(rbd.DiffIterateConfig{
    SnapName:    parentSnapName,  // "" = full
    Offset:      0,
    Length:      imageSize,
    WholeObject: rbd.EnableWholeObject,
    Callback: func(off, len uint64,
                   exists int, _ interface{}) int {
        extentCh ← snapExtent{
            offset: int64(off),
            length: len,
            ceph:   snap,  // holds rbd.Image ref
        }
        return 0
    },
})
```

[2] FETCH: ReadAt per Extent

```
func (s *snapExtent) Read(b []byte) (int, error) {
    if s.data == nil {
        s.data = make([]byte, s.maxLength)
        s.length, _ = s.ceph.ReadAt(s.data, s.offset)
    }
    n := copy(b, s.data[s.pos:s.pos+n])
    s.pos += n
    return n, nil
}
```

[3] EXPORT: Concurrent CopyExtents

```
func CopyExtents(dst io.WriterAt, src ExtentReader,
    chunkSize, concurrency int) {
  extents := make(chan Extent)
  for i := 0; i < concurrency; i++ {
    go func() {
      buf := make([]byte, chunkSize)
      for ext := range extents {
        writeExtent(dst, ext, buf)
      }
    }()
  }
  for {
    ext, err := src.NextExtent(ctx)
    if err == io.EOF { break }
    extents ← ext
  }
```

[4] FINALIZE: Orchestrate & Rotate

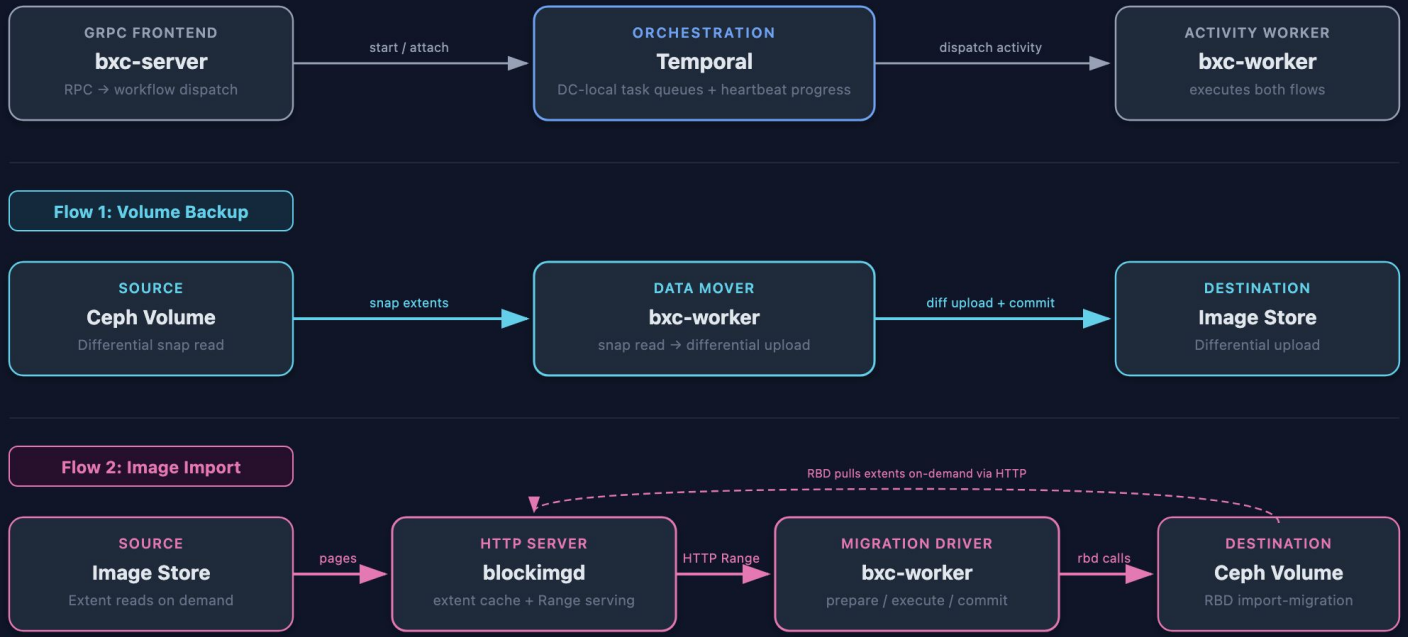
```
snaps := ceph.ListSnapshots(volume)
diffParent := findLatestBackupSnap(snaps) // or ""
ceph.CreateSnapshot(volume, newSnapName)

writer := target.OpenDifferential(newSnap, diffParent)
reader := snap.DifferentialReader(diffParent)

CopyExtents(writer, reader, 4*1024*1024, 8)

target.Commit(writer)
if diffParent ≠ "" {
    ceph.DeleteSnapshot(diffParent) // rotate
}
```

[Orchestration]



Data Paths

- **Volume Backup** — Ceph snap → bxc-worker reads + uploads → Image Store
- **Image Import** — Image Store → blockimgd → bxc-worker drives RBD import-migration

Key Difference

Backup: bxc-worker moves data (reads Ceph, writes image pipeline)
Import: bxc-worker drives the migration; Ceph pulls data from blockimgd directly via HTTP Range requests

[Thank you. Questions?]