Swiftacular: Swift Deployment, Monitoring, and Real-time Performance Analysis of Regular vs. BlueStore Backends

David Sariel & Adrian Fusco

OpenInfra Summit Europe - Oct 19th, 2025

Red Hat

About Us

David Sariel
Senior Software Engineer

Building integration pipelines for RHOSO 18 and crafting monitoring utilities to keep things running smoothly and efficiently.

Adrian Fusco
Software Engineer

– DevOps soul, hartened with Perl and Bash.
– Passionate for traveling, food and immersions into different cultures.
– Fluently speaks Español, English, Galego, Italiano, learning Turkçe.

Agenda

Introduction to Swiftacular

- An overview of Swiftacular for OpenStack Swift deployment and monitoring.

Core Swiftacular Functions

- Deploying Swift clusters.
- Monitoring Swift performance and health.

Swift and Ceph Storage Backend Architectures

- The traditional Swift backend.
- The Ceph BlueStore architecture.

Performance considerations

- Previous attempts to improve storage backend.
- BlueStore as a high-performance and memory efficient storage backend for Swift.

Development Environment

Future work

Final remarks

Introduction to Swiftacular

Swiftacular - Key Terminology

- **Swift - Object storage** service for the OpenStack cloud platform.
- **Swiftacular** - A project that allows OpenStack Swift deployment and monitoring using **Ansible** and **Vagrant.**.
- **Ansible -** open source automation project that reduces complexity and runs everywhere. Using Ansible lets you automate virtually any task
- **BlueStore** - A high-performance storage backend designed specifically for **Ceph**, managing data directly on disk to optimize performance for object storage workloads.
- **Grafana** - An open-source platform used for creating interactive dashboards for **monitoring and observability**
- **RocksDB** - A high performance embedded database for key-value data.

What is OpenStack Swift?

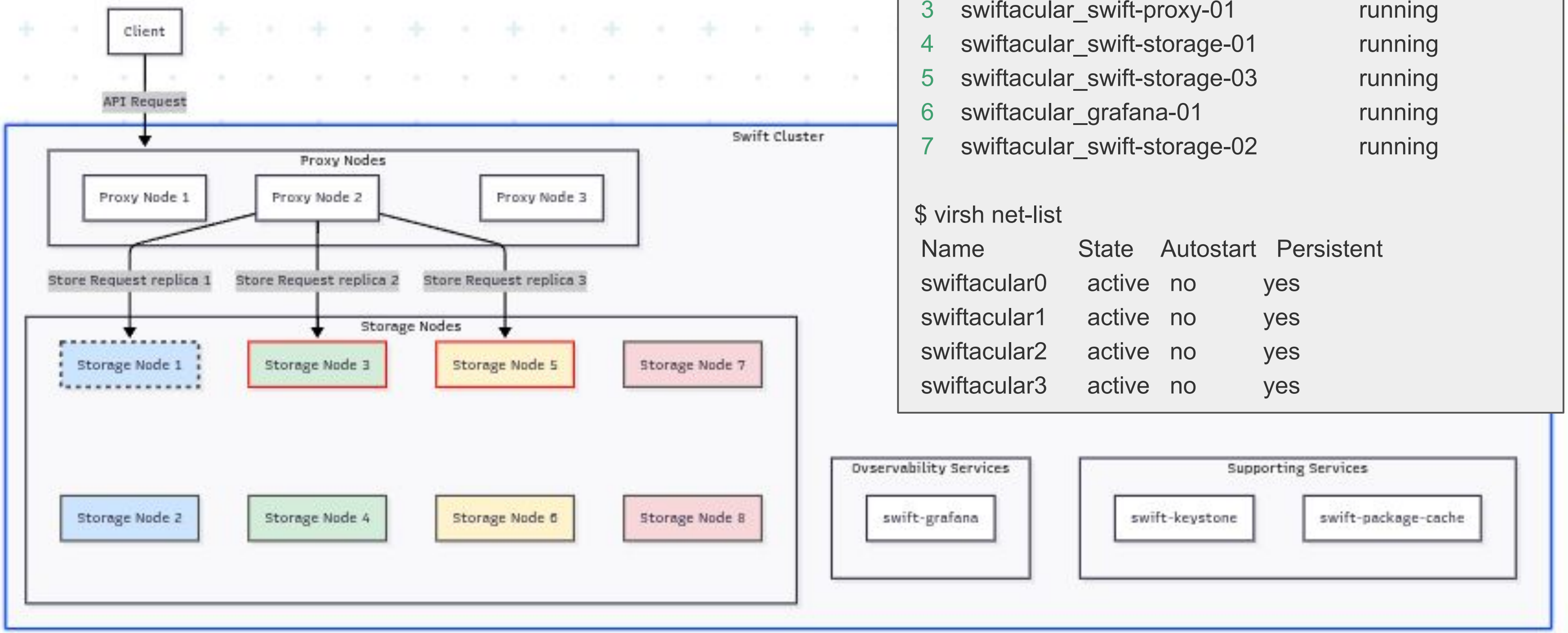
- OpenStack Object Storage Service Project - Open Source
- Global Cluster Capability
- Middleware Architecture - Add new functionalities
- Partial Object Retrieval
- Scale vertically and horizontally distributed storage. Backs up and archives large amounts of data with linear and higher performance.
- Enable direct browser access to content, such as for a control panel.

More info in: [Welcome to Swift's documentation!]

Swiftacular - Components

```
$ virsh list
 Id   Name                                 State
 1    swiftacular_swift-package-cache-01   running
 2    swiftacular_swift-keystone-01        running
 3    swiftacular_swift-proxy-01           running
 4    swiftacular_swift-storage-01         running
 5    swiftacular_swift-storage-03         running
 6    swiftacular_grafana-01               running
 7    swiftacular_swift-storage-02         running

$ virsh net-list
 Name           State    Autostart   Persistent
 swiftacular0   active   no          yes
 swiftacular1   active   no          yes
 swiftacular2   active   no          yes
 swiftacular3   active   no          yes
```



💡 We support Fedora and Ubuntu and IPv6 isn't supported.

Core Swiftacular Functions

Deploying a Swift Cluster with Swiftacular

Why Ansible?

- **Simple & Understandable:** Playbooks are easy to read and write.
- **Agentless:** Executes over standard **SSH**, no special client software needed on remote systems.

Prerequisites:

- Vagrant
- Libvirt
- Ansible
- An internet connection

Note: Vagrant is used to provision VMs, network connectivity between VMs and ssh keys setup. But one can run deployment playbooks on any preprovisioned and interconnected machines (VMs, baremetal)

Core swiftacular functionality - What You Get: A Resilient Swift Cluster

A few simple commands to get everything running

```
# Clone swiftacular repo
$ git clone https://github.com/davidsaOpenu/swiftacular.git
$ cd swiftacular

# Install prerequisites on the host (requires sudo)
$ ./1_install_prereqs.sh

# Prepare storage nodes with precompiled bluestore
$ ./scripts/compile_and_test_bluestore.sh --ceph

# Provision VMs
$ ./2_provision_VMs.sh

# Deploy Swift and monitoring dashboards
$ ./3_deploy_swift_with_monitoring.sh
```

```
$ vagrant box list
eurolinux-vagrant/centos-stream-9      (libvirt, 9.0.48)
eurolinux-vagrant/centos-stream-9-ceph (libvirt, 0)
```

💡 If you don't use Vagrant, make sure to check interfaces ens6-nns9 (in theory it works)

Core Swiftacular Functions - Monitoring Swift performance and health.

- **Grafana**: Visualization & Dashboarding Platform that offers several kinds of visualizations, unifies data from different sources and provides built-in alerting system.
- **Grafonnet:** Jsonnet library for generating Grafana dashboards (dashboards as a code).

Swiftacular Monitoring

- Host Overview dashboard contains common metrics included in PCP, such as memory and CPU utilization.
- To gather metrics related to the databases Swift uses for storage, we have a PMDA called swiftdbinfo and we have a specific dashboard for it.

More info:

- Performance Co-Pilot
- 2. Writing A PMDA — pcp 7.0.2-1 documentation

One of Object DBs

Core Swiftacular Functions - Monitoring Swift performance and health.

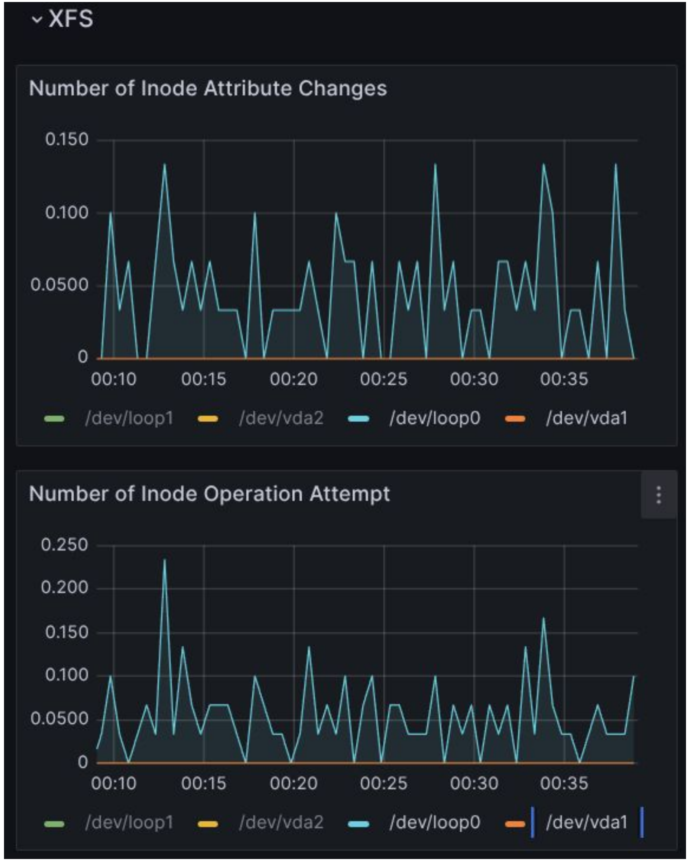
Visualizations:

- Total of objects to track data growth, identify the largest containers, and understand the on disk overhead of metadata.
- Dedicated log panel that marks the exact time a container's database is "sharded" by Swift.
- Distribution of the objects into size buckets.

PCP - Performance Co-Pilot dashboard

Inode Performance

- **Missed Inode Ops**
  - Shows slow disk reads for file metadata
  - Rising value → small-file slowdown
- **Cache Hit Ratio**
  - Measures hits vs. disk reads
  - Higher = faster performance
- **Memory Use**
  - Tracks inode cache RAM usage
  - Shows memory cost of small files
- **Insight**
  - Trade-off: **speed vs. memory**
  - Goal: **cut memory use, keep speed**



💡 Swift recon and other metrics to be added see references slides

Swift and Ceph Storage Backend Architectures

Swift storage backend

```
An implementation of the object server that wants to use a different
DiskFile class would simply over-ride this method to provide that
behavior.
"""
return self._diskfile_router[policy].get_diskfile(
    device, partition, account, container, obj, policy, **kwargs)
```

```
class DiskFile(BaseDiskFile):
    reader_cls = DiskFileReader
    writer_cls = DiskFileWriter
```

```
xattr.setxattr/xattr.getxattr
```

OpenStack Swift - Layered Architecture

Client Request

API Calls: GET, PUT, DELETE, POST

Proxy Server

Port 8080 • Authentication • Request Routing • Load Balancing

Ring (Hash Ring)

Maps Object/Container/Account Ring to storage node/s

Storage Node(s) - object-server

Port 6200 • Multiple nodes (3+ replicas) • Handles object operations • Implemented in `obj/server.py`

obj/diskfile.py

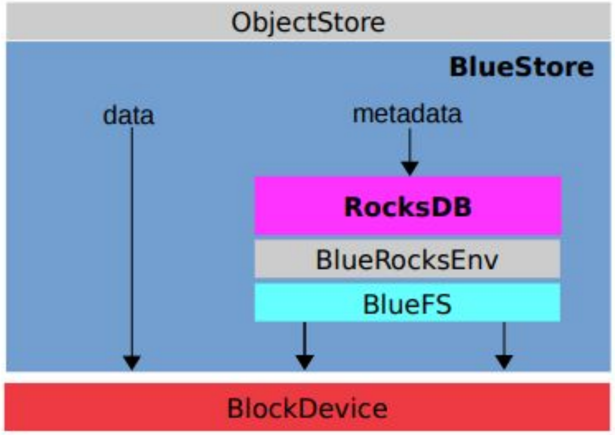
Storage Backend Interface • Manages disk I/O operations

XFS Filesystem

Physical Storage • Extended Attributes • Optimized for large files

Ceph storage backend

- BlueStore = **Bl**ock + **NewStore**
  - consume raw block device(s)
  - key/value database (RocksDB) for metadata
  - data written directly to block device
  - pluggable block Allocator (policy)
  - pluggable compression
  - checksums, ponies, ...
- We must share the block device with RocksDB



Slide #14 from Sege Weil's talk

More reading:
1) Part - 1 : BlueStore (Default vs. Tuned)
2) BlueStore - IBM documentation

Ceph storage backend

```
int mount_existing() {
    if (mounted) return 0;
    if (!store) {
        struct stat st;
        if (stat((base_path + "/block").c_str(), &st) != 0) return -1;
        store = new BlueStore(cct.get(), base_path);
    }
    if (store->mount() < 0) return -1;
    mounted = true;
    ch = store->open_collection(cid);
    return ch ? 0 : -1;
}

int write_object(const string& obj_name, const string& data) {
    if (!mounted && mount_existing() < 0) return -1;
    ghobject_t oid(hobject_t(obj_name, "", CEPH_NOSNAP, 0, 0, ""));
    bufferlist bl;
    bl.append(data);
    ObjectStore::Transaction t;
    t.write(cid, oid, 0, data.size(), bl);
    return store->queue_transaction(ch, std::move(t));
}

int read_object(const string& obj_name) {
    if (!mounted && mount_existing() < 0) return -1;
    ghobject_t oid(hobject_t(obj_name, "", CEPH_NOSNAP, 0, 0, ""));
    bufferlist bl;
    int r = store->read(ch, oid, 0, 1024*1024*1024, bl);
    if (r < 0) return r;
    cout.write(bl.c_str(), bl.length());
    cout.flush();
    return 0;
}
```

```
int create() {
    struct stat st;
    if (stat(base_path.c_str(), &st) == 0) {
        cerr << "BlueStore exists at " << base_path << std::endl;
        return -1;
    }
    if (mkdir(base_path.c_str(), 0755) != 0) {
        cerr << "Failed to create " << base_path << std::endl;
        return -1;
    }
    string block_path = base_path + "/block";
    int fd = ::open(block_path.c_str(), O_CREAT|O_RDWR|O_TRUNC, 0644
    if (fd < 0) return -1;
    ::ftruncate(fd, 10LL*1024*1024*1024); # 2 GB
    ::close(fd);

    store = new BlueStore(cct.get(), base_path);
    if (store->mkfs() < 0) return -1;
    if (store->mount() < 0) return -1;
    mounted = true;

    ch = store->create_new_collection(cid);
    ObjectStore::Transaction t;
    t.create_collection(cid, 0);
    return store->queue_transaction(ch, std::move(t));
}
```


BlueStoreUtil code

Performance considerations

Outdated projects for Swift benchmarking

- Benchmarking tool for OpenStack Swift. Mirror of code maintained at opendev.org.
- GitHub - intel-cloud/cosbench: a benchmark tool for cloud object storage service
- ssbench - Benchmarking tool for Swift clusters

Previous work

- A swift object uses at least two inodes on the filesystem.
- Clusters with many small files (10 million objects per disk) suffer from performance degradation the directory structure does not fit in memory.
- Replication / auditor operations trigger a lot of IO.
- Over 40% of disk activity may be caused by "listdir" operations.
- Smallfiles page
- Implementation
- Slides

Hmm... why not to try BlueStore?

Recall why ceph replaced FileStore with BlueStore:

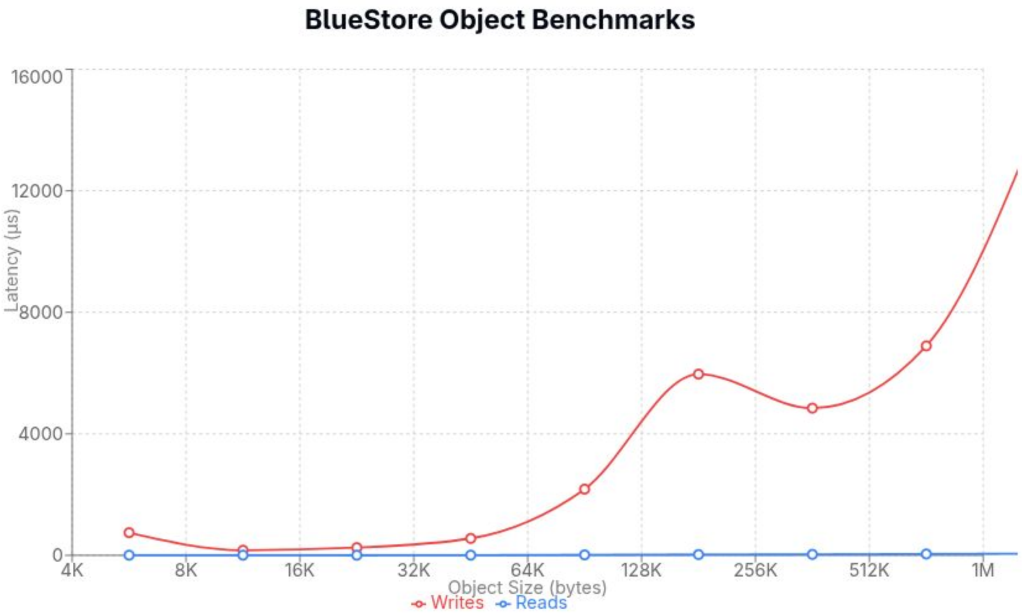
Filesystem overhead - FileStore relied on local filesystems (typically XFS) which added metadata overhead and couldn't be optimized for Ceph's specific workload. BlueStore manages raw devices directly, giving it full control.

Inconsistent performance - FileStore's journal on SSDs with data on HDDs created complex I/O patterns. BlueStore's architecture provides more predictable performance with better separation of metadata and data.

Object enumeration slowness - Listing objects in FileStore required filesystem operations that didn't scale well. BlueStore uses RocksDB for metadata, making object enumeration much faster.

But first let's verify if the direction is correct and perform some basic benchmarks

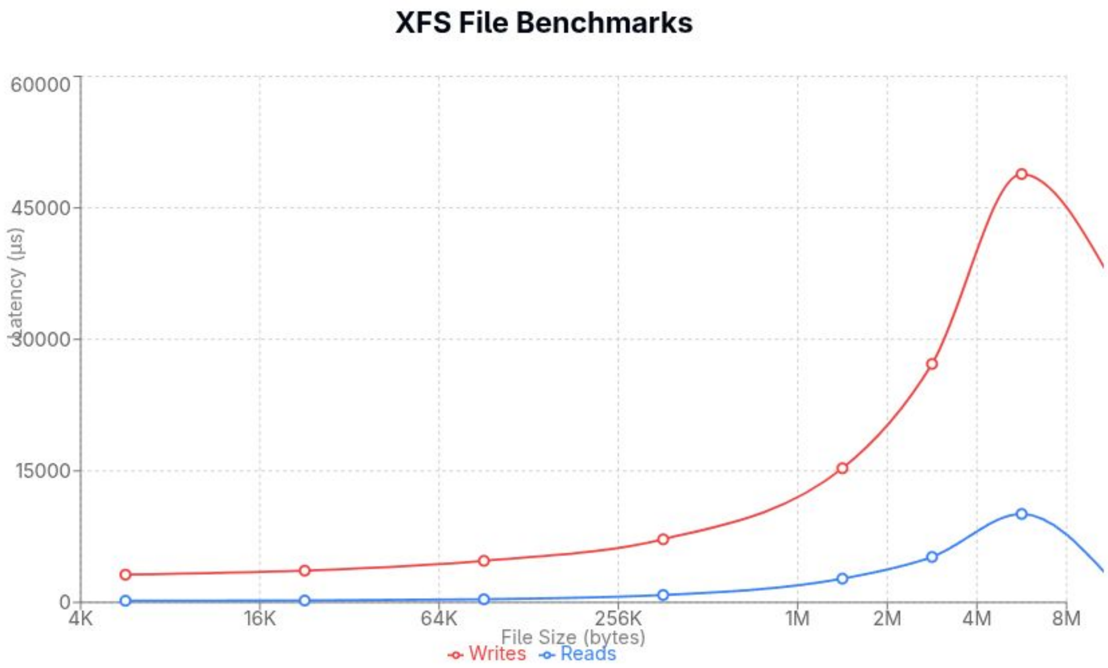
BlueStore reads/writes




```
==== BlueStore Object Benchmarks ====
Object size: 4096 B | Write: 744.716 µs (5.24529 MB/s) | Read: 4.52534 µs (863.195 MB/s)
Object size: 8192 B | Write: 167.977 µs (46.5094 MB/s) | Read: 5.3618 µs (1457.07 MB/s)
Object size: 16384 B | Write: 252.92 µs (61.7784 MB/s) | Read: 4.35084 µs (3591.26 MB/s)
Object size: 32768 B | Write: 556.307 µs (56.174 MB/s) | Read: 5.64484 µs (5536.03 MB/s)
Object size: 65536 B | Write: 2178.71 µs (28.6867 MB/s) | Read: 14.9655 µs (4176.26 MB/s)
Object size: 131072 B | Write: 5964.74 µs (20.9565 MB/s) | Read: 25.6363 µs (4875.89 MB/s)
Object size: 262144 B | Write: 4845.27 µs (51.5967 MB/s) | Read: 30.9 µs (8090.63 MB/s)
Object size: 524288 B | Write: 6893.33 µs (72.5339 MB/s) | Read: 43.6421 µs (11456.8 MB/s)
Object size: 1048576 B | Write: 14465.5 µs (69.1302 MB/s) | Read: 53.8074 µs (18584.8 MB/s)
```

[Benchmark code]

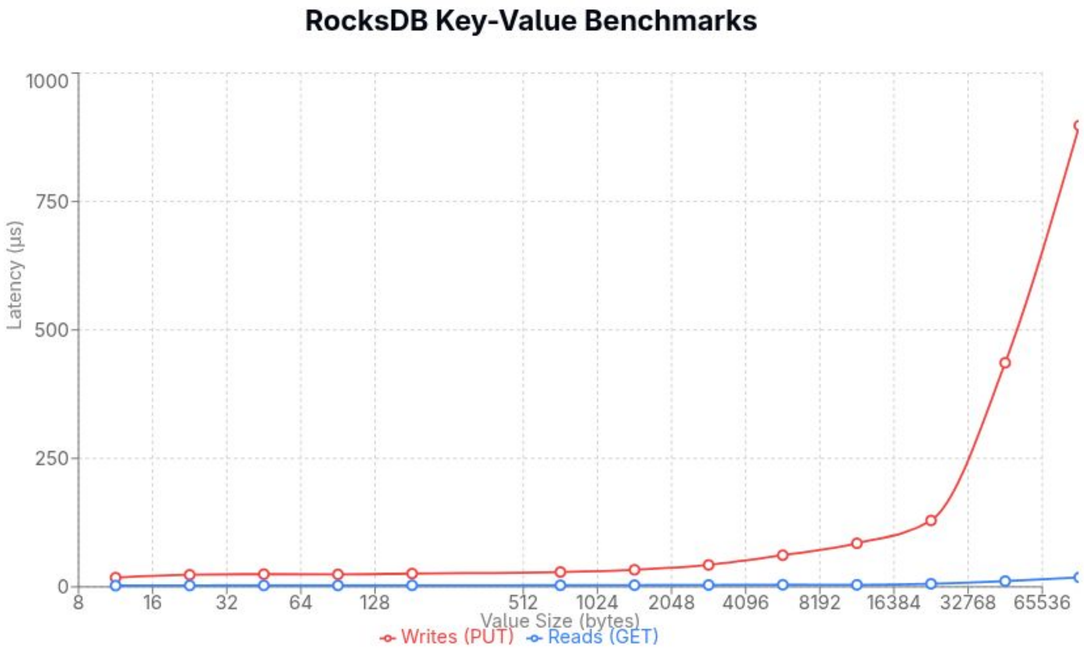
XFS reads/writes



Size	Write (us)	Write (MB/s)	Read (us)	Read (MB/s)
4 KB	3160.97	1.24	190.54	20.50
16 KB	3623.69	4.31	214.53	72.83
64 KB	4741.97	13.18	356.66	175.24
256 KB	7196.22	34.74	844.97	295.87
1 MB	15299.07	65.36	2711.28	368.83
2 MB	27202.22	73.52	5184.49	385.77
4 MB	48857.53	81.87	10088.24	396.50
8 MB	36772.71	217.55	2603.04	3073.33

Benchmark code

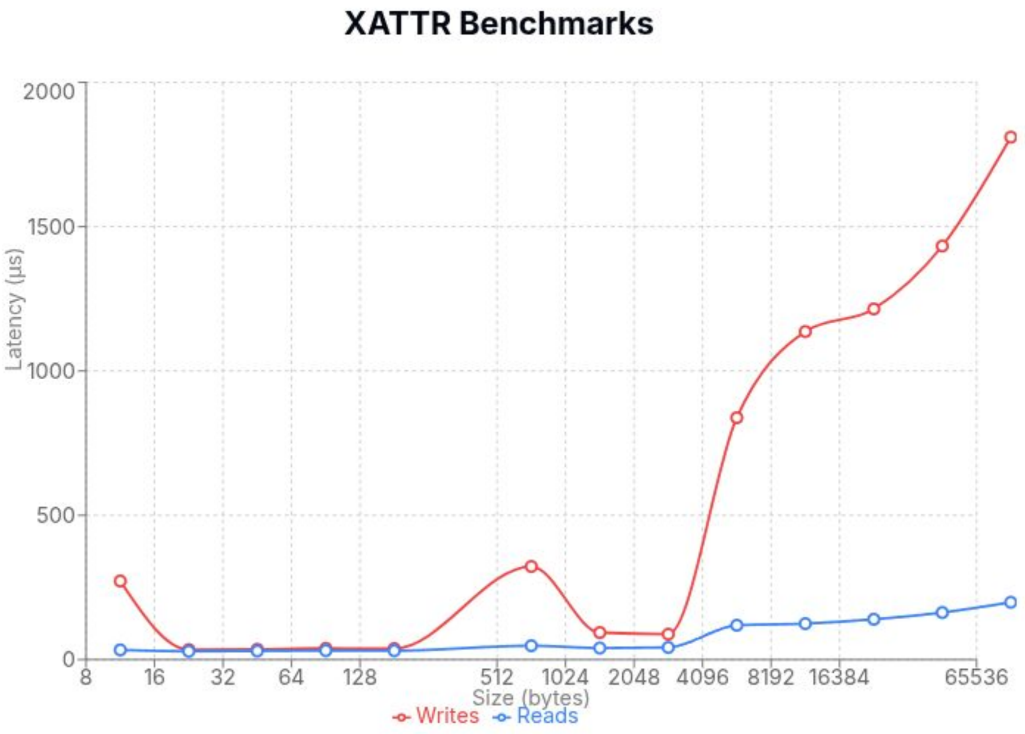
RocksDB set/get latences



value_size	avg_put	avg_get
8 B	17.955 µs	2.427 µs
16 B	23.466 µs	2.653 µs
32 B	24.632 µs	2.810 µs
64 B	24.272 µs	2.739 µs
128 B	25.880 µs	2.923 µs
512 B	28.867 µs	3.056 µs
1024 B	33.196 µs	3.300 µs
2048 B	42.624 µs	3.580 µs
4096 B	61.553 µs	4.024 µs
8192 B	84.788 µs	3.876 µs
16384 B	129.208 µs	5.951 µs
32768 B	436.043 µs	11.109 µs
65536 B	897.789 µs	18.357 µs

[Benchmark code]

XATTR set/get latences




```
Benchmarking xattr size = 8 bytes ...
attr_size=    8 B  avg_set= 272.221 µs  avg_get=  33.538 µs
Benchmarking xattr size = 16 bytes ...
attr_size=   16 B  avg_set=  34.585 µs  avg_get=  28.733 µs
Benchmarking xattr size = 32 bytes ...
attr_size=   32 B  avg_set=  35.690 µs  avg_get=  29.762 µs
Benchmarking xattr size = 64 bytes ...
attr_size=   64 B  avg_set=  38.949 µs  avg_get=  31.056 µs
Benchmarking xattr size = 128 bytes ...
attr_size=  128 B  avg_set=  38.278 µs  avg_get=  30.623 µs
Benchmarking xattr size = 512 bytes ...
attr_size=  512 B  avg_set= 322.652 µs  avg_get=  48.055 µs
Benchmarking xattr size = 1024 bytes ...
attr_size= 1024 B  avg_set=  93.677 µs  avg_get=  40.077 µs
Benchmarking xattr size = 2048 bytes ...
attr_size= 2048 B  avg_set=  88.134 µs  avg_get=  42.108 µs
Benchmarking xattr size = 4096 bytes ...
attr_size= 4096 B  avg_set= 838.043 µs  avg_get= 118.872 µs
Benchmarking xattr size = 8192 bytes ...
attr_size= 8192 B  avg_set=1136.415 µs  avg_get= 124.630 µs
Benchmarking xattr size = 16384 bytes ...
attr_size=16384 B  avg_set=1214.879 µs  avg_get= 139.670 µs
Benchmarking xattr size = 32768 bytes ...
attr_size=32768 B  avg_set=1432.968 µs  avg_get= 163.015 µs
Benchmarking xattr size = 65536 bytes ...
attr_size=65536 B  avg_set=1810.127 µs  avg_get= 198.532 µs
```

[Benchmark code]

Core swiftacular functionality - What You Get: A Resilient Swift Cluster

XFS Block Allocations: Spike in block allocation activity starting at 08:19:38, with values jumping to 104,693 and 62,632 in one of the columns, and later to 14,035 and 11,361 in another.

XFS Extent Allocations: Shows a similar, though less dramatic, increase in extent allocations around the same time. The values spike to 225, 4,010, and 668 in the final two readings.

Explanation on XATTR benchmark

Why 8 Bytes Is Slower?

- Cache Alignment and Copy Cost
- Inode Space Layout in XFS
- Fixed Per-Call System Call Overhead
- xattr Library and Python Overhead

Why it slows down at 512-byte xattrs?

XFS stores extended attributes once they exceed the inline storage space available in the inode.

Not complicated to replace writes to XFS with

```
# ./bs_util create bs
# echo "Hello BlueStore" > /tmp/testfile
# cat /tmp/testfile | ./bs_util write bs test_object
# ./bs_util read bs test_object
```

And xattrt updates with

```
db = rocksdb.DB("test.db", rocksdb.Options(create_if_missing=True))
db.put(b"key", b"value")
db.get(b"key")
```

At the cost of 2-10 µs overhead for bs_utill call from *diskfile.py*

Q: But maybe to rewrite obj/server.py obj/diskfile.py as a service on C++?

Development Environment

A 6-Step Workflow for Swift Development

Configure SSH Access

- Add Vagrant VM connection details to local SSH config: **vagrant ssh-config swift-storage-01 >> ~/.ssh/config**

Install VS Code Extension

- Open **Extensions** and install "**Remote - SSH**".

Connect to Remote Host

- Open "**Remote Window**", select "**Remote-SSH**: Connect to Host..." and choose swift-storage-01.

Open Folders & Edit Code

- In the remote VS Code window, go to File > Open Folder... and enter your code path (e.g., ~/ceph or /usr/lib/python3.9/site-packages/swift).

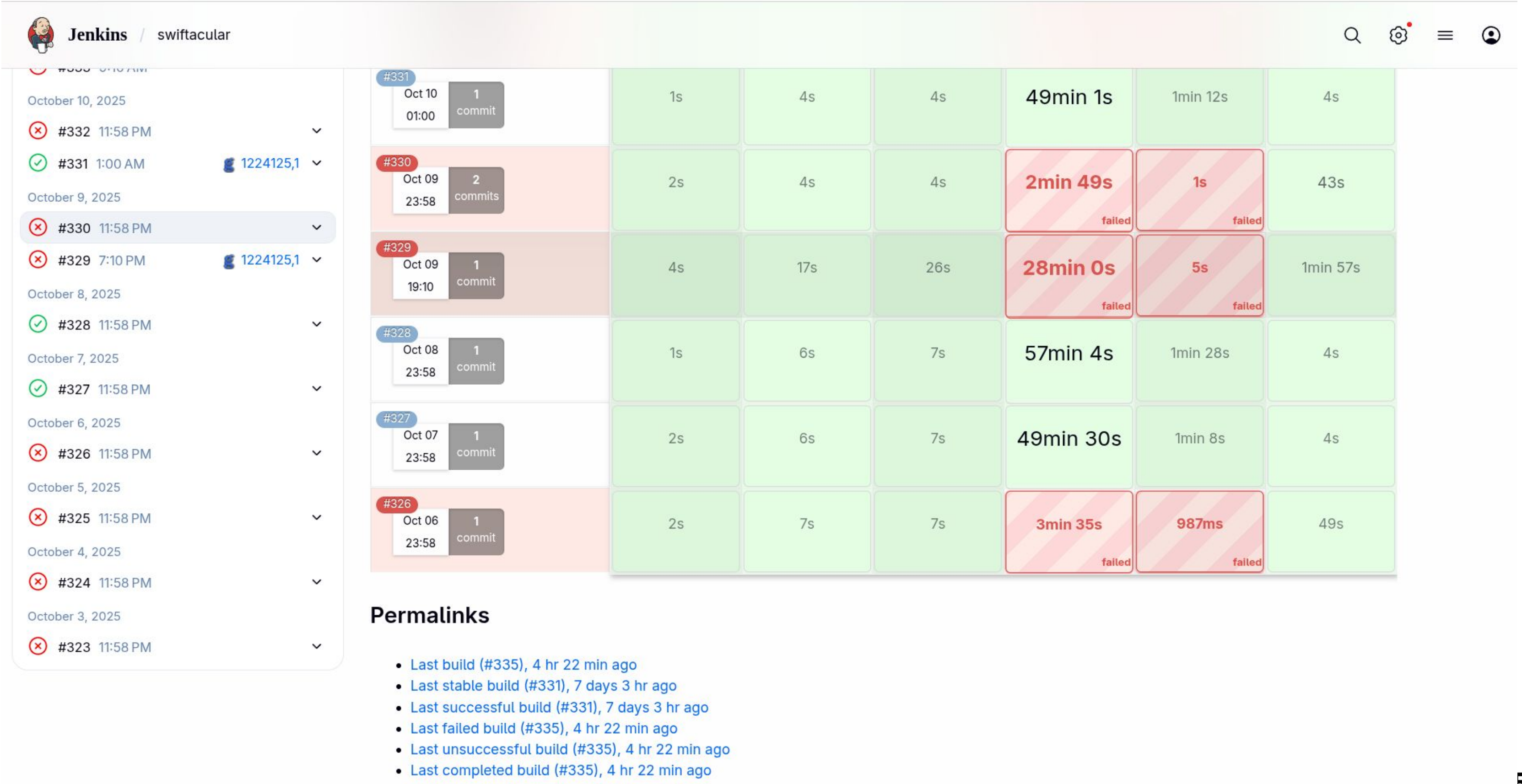
Create Patches from Commits

- After committing changes, export them as .patch files (e.g. g**it format-patch HEAD~N**).

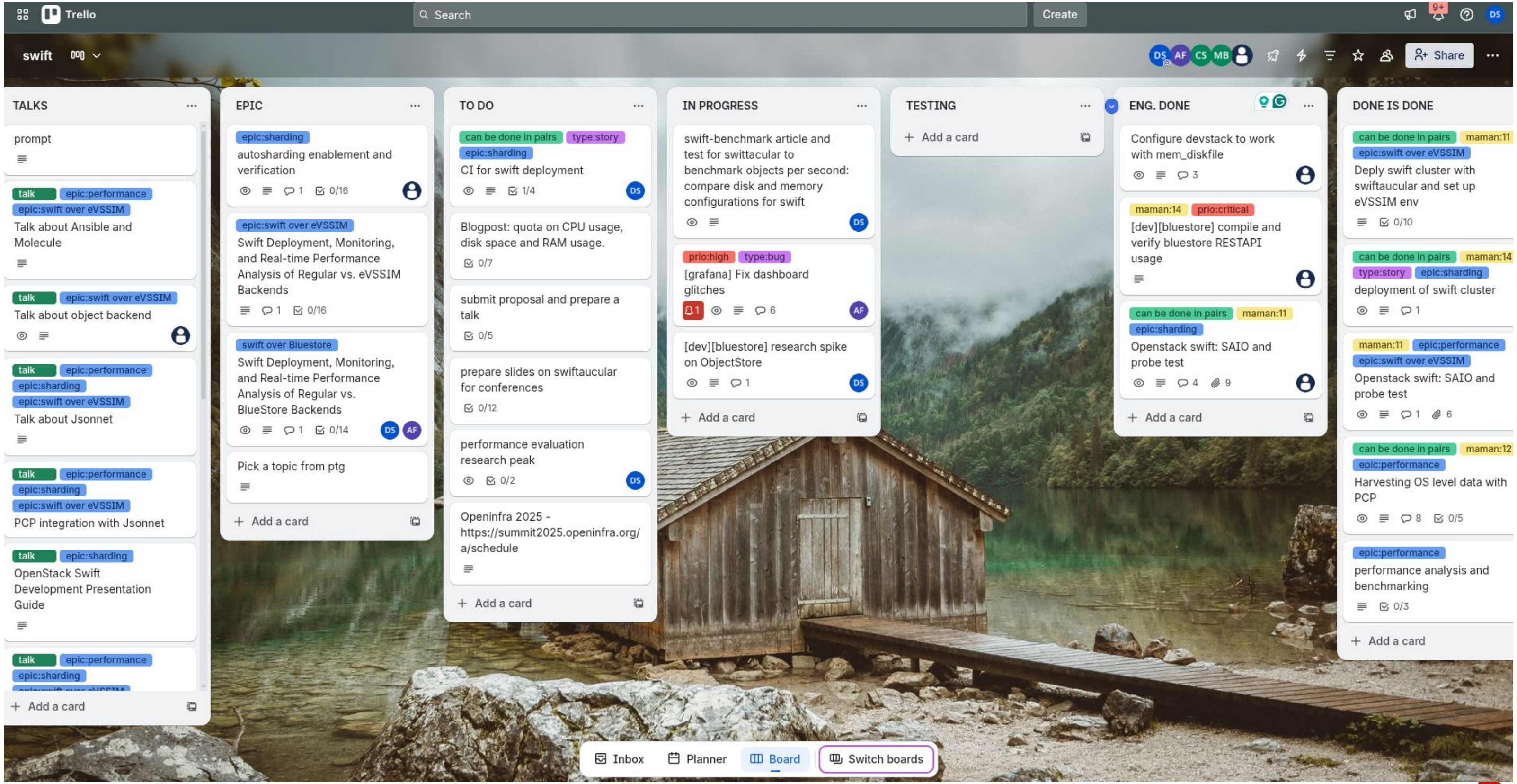
Apply Patches

- Move .patch files to the target project (e.g., Swiftacular) and apply them like in this example

Continuous Integration



Project management board



Future Work

Future Work

- All items marked with 💡

And also

- erasure code support
- Swift over eVSSIM

Final remarks

Red Hat

Summary

Swiftacular is instrumental in deploying Swift clusters that are not DevStack deployments but allows deployment of multi-node clusters, and its dashboard provides insights that suggest the following architectural changes: to replace the swift-object service that is implemented in Python with a service that is implemented in C++ that serves as an endpoint to the Bluestore backend. In any case, before proceeding in that direction, all benchmarks have to be executed on a real device (currently we compared devices that point to a file in both cases).

Another important aspect of deployment with Swiftacular is that it allows code (Bluestore and/or Swift-related) changes on a running multi-node cluster in a very easy way.

References

General

- OpenStack Swift Documentation
- GitHub - Swiftacular
- Curtis Collicutt's page on Swiftacular

Monitoring

- Performance Co-Pilot metrics (PCP)
- Object Storage monitoring
- Statsd Metrics

Repository

- Swiftacular Git Repository

Thanks

Access to the Slides & Project

Swiftacular: Swift Deployment, Monitoring, and Real-time Performance Analysis of Regular vs. BlueStore Backends

David Sariel & Adrian Fusco

OpenInfra Summit Europe - Oct 19th, 2025

Red Hat