Configuration and Deployment Guide for OpenStack* Swift* Object Storage on Intel® Atom™ Processor and Intel® Xeon® Processor Microservers

About this Guide

This *Configuration and Deployment Guide* explores designing and building an object storage environment based on OpenStack* Swift*. This guide focuses on object storage in cloud environments, where responsiveness and cost are critical factors. High-performance, energy-efficient microservers, such as those based on the latest generation of Intel® Atom™ processors and Intel® Xeon® E3 processors, meet these requirements. The guide uses data from recent benchmarks conducted by Intel® Software and Services Group on Intel Atom processor and Intel Xeon E3 processor-based microservers.

Table of Contents

Introduction 3
Driving Forces and Strategic Considerations 3
A New Look at Storage 4
Scalability 4
Durability 4
Flexibility 5
Choice of Components 5
Compatibility and Familiarity 5
Public or Private Cloud 5
Overview of the Swift Architecture 6
Primary Objective: High Availability and Fast Access to a Wide Variety of Data 6
Theory of Operation 6
Upload 6
Download 7
Replication 7
Other Housekeeping Services 7
Configuration and Deployment 8
Swift Topology 8
Access Tier 8
Storage Nodes 9
Factors to Consider 9
Performance Considerations 10
Overall Strategy 10
Nodes 10
Ring Configuration 10
General Server Configuration 11
Memcached Considerations 11
Storage Server Benchmarking for Swift 11
COSBench Platform 12
System Under Tests (SUT) Configurations 12
Results 14
Tuning and Optimization 16
General Service Tuning 16
Filesystem Considerations 16
Summary 16
Learn More 17
Appendix A – Additional Resources 17

Introduction

OpenStack* Swift* Object Storage (swift.openstack.org) enables a scalable, redundant storage filesystem for cloud infrastructures, accessible using web protocols at incredible speeds with very high availability. Swift Object Storage started as a project at the cloud service provider, RackSpace*; it was released to the OpenStack (www.openstack.org) community. It is available under the Apache* 2 license.

In Swift, objects are written to multiple disk drives spread across a cluster of storage servers in the data center. But systems do not mount Swift storage like traditional storage area network (SAN) or network attached storage (NAS) volumes. Instead, applications use a representational state transfer (REST) application programming interface (API) to request data stored on the Swift cluster.

The Swift Object Storage cluster scales in a near-linear fashion by adding new servers. The software ensures data replication and integrity throughout the cluster. Should a storage server or hard drive fail, Swift Object Storage software replicates content from other active servers to new locations in the cluster. Because the software ensures data replication and distribution across different storage devices, data center architects can use industry-standard hard disk drives (HDDs), solid state disk drives (SSDs), and servers, including microservers running on Intel Xeon E3 processors or Intel Atom processors.

Other key characteristics of Swift Object Storage include:

- All objects stored in Swift have a URL.
- All objects stored are replicated three times in unique-as-possible zones, which can be defined as a group of drives, a node, a rack, etc.
- Each object has its own metadata.
- Developers interact with the object storage system through a REST HTTP API.
- Object data can be located anywhere in the cluster.
- The cluster scales by adding nodes, without sacrificing performance, which allows a more cost-effective linear storage expansion versus fork-lift upgrades.
- Data doesn't have to be migrated to an entirely new storage system.
- New nodes can be added to the cluster without downtime.
- Failed nodes and disks can be swapped out with no downtime.
- Swift runs on cost-effective industry-standard hardware.

Driving Forces and Strategic Considerations

With rapidly-growing mobile user-bases for social media and a variety of other cloud services, a wide variety of structured and unstructured data needs to be instantly accessible, secure and redundant, possibly stored forever, and available through a variety of devices for a variety of applications. Storage silos utilizing protocols that are tied to specific applications no longer meet the needs of web-based applications. Social media, online video, user-uploaded content, gaming, and software-as-a-service (SaaS) applications are just some of the forces driving data centers to take a new look at how data is stored.

Swift meets these types of demands, from small deployments for storing virtual machine (VM) images, to mission-critical storage clusters for high-volume websites, to mobile application development environments, custom file-sharing applications, big data analytics, and private storage Infrastructure as a Service (IaaS). Swift originated in a large-scale production environment at RackSpace, so it was designed for large-scale, durable operations.

A New Look at Storage

When considering different scalable storage solutions, several things are important to keep in mind.

- Scalability
- Durability
- Cost
- Choice of components
- Compatibility and familiarity
- Cloud or in-house

Scalability

To easily grow with expanding content and users, storage systems need to handle web-scale workloads with many concurrent readers from and writers to a data store, writing and reading a wide variety of data types. Some data is frequently written and retrieved, such as database files and virtual machine images. Other data, such as documents, images, and backups, are generally written once and rarely accessed. Unlike traditional storage file systems, Swift Object Storage is ideal for storing and serving content to many, many concurrent users. It is a multi-tenant and durable system, with no single point of failure, designed to contain large amounts of unstructured data at low cost and accessible via a REST API. In Swift, every object has its own URL.

Swift can easily scale as the cluster grows in the number of requests and data capacity. The proxy servers that handle incoming API requests scale in a near-linear fashion by adding more servers. The system uses a shared-nothing approach and employs the same proven techniques that have been used to provide high availability by many web applications. To scale out storage capacity, nodes and/or drives are added to the cluster. Swift can easily expand from a few gigabytes on a couple machines to dozens of petabytes on thousands of machines scattered around the planet, simply by adding hardware. Swift automatically incorporates the new devices into its resources.

Since all content in the Object Store is available via a unique URL, Swift can easily serve content to modern web applications, and it also becomes very straightforward to either cache popular content locally or integrate it with a CDN, such as Akamai*.

Durability

Downtime is costly. Lack of content can affect a wide range of users, preventing them from doing their work or using the service to which they subscribe. Swift can withstand hardware failures without any downtime and provides operations teams the means to maintain, upgrade, and enhance a cluster while in flight. To achieve this level of durability, Swift distributes objects in triplicate across the cluster. A

write must be confirmed in two of the three locations to be considered successful. Auditing processes maintain data integrity, and replicators ensure a sufficient number of copies across the cluster, even if a storage device fails.

Swift also can define failure zones. Failure zones allow a cluster to be deployed across physical boundaries, each of which could individually fail. For example, a cluster could be deployed across several nearby data centers, enabling it to survive multiple datacenter failures.

Flexibility

Proprietary solutions can drive up licensing costs. Swift is licensed freely under the Apache 2 open source license, ensuring no vendor lock-in**,** providing community support, and offering a large ecosystem.

Choice of Components

Proprietary storage solutions offer turnkey benefits, but at a higher purchase price and little or no choice of components within the solution. Open source Swift software is built on proven, industry-wide components that work in large-scale production environments, such as rsync, MD5, SQLite, memcached, xfs, and Python*. Swift runs on off-the-shelf Linux* distributions, including Fedora*, RedHat Enterprise Linux* (RHEL), openSUSE*, and Ubuntu*.

From a hardware perspective, Swift is designed from the ground up to handle failures, so that reliability of individual components is less critical. Thus, Swift clusters can run on a broad range of multi-socket servers and low-cost, high-performance microservers, with commodity hard disk drives (HDDs) and/or solid-state drives (SSDs). Hardware quality and configuration can be chosen to suit the tolerances of the application and the ability to replace failed equipment.

Compatibility and Familiarity

If an operator considers moving from a public cloud service to an internal installation, the method used by services to access the storage becomes more important. Access to the Swift Object Storage system is through a REST API that is similar to the Amazon.com* S3* API and compatible with the Rackspace Cloud Files* API. This means that: (a) applications currently using S3 can use Swift without major re-factoring of the application code; and (b) applications taking advantage of both private and public cloud storage can do so, as the APIs are comparable.

Because Swift is compatible with public cloud services, developers and systems architects can also take advantage of a rich ecosystem of available commercial and open-source tools for these object storage systems.

Public or Private Cloud

Not every organization can—or should—use a public cloud to store sensitive information. For private cloud infrastructures, Swift enables organizations to reap the benefits of cloud-based object storage, while retaining control over network access, security, and compliance.

Cost can deter bringing cloud storage in-house. Public cloud storage costs include per-GB pricing and data transit charges, which grow rapidly for larger storage requirements. With the declining costs of

data center servers and drives and the emergence of new high-efficiency microservers based on the same x86 instruction set architecture as the other servers in the data center, the total cost of ownership (TCO) for a Swift cluster can be on par with AWS S3 for a small cluster and much less than AWS S3 for a large cluster.

In addition, the network latency to public storage service providers may be unacceptable, especially for time-based content. A private deployment can provide lower-latency access to storage.

Overview of the Swift Architecture

The Swift Object Storage architecture is distributed across a cluster to prevent any single point of failure and to allow easy horizontal scalability. A number of periodic processes perform housekeeping tasks on the data store. The most important of these are the replication services, which ensure consistency and availability throughout the cluster. Other periodic processes include auditors, updaters, and reapers.

Authentication is handled through configurable Web Server Gateway Interface (WSGI) middleware—usually the Keystone* Architecture.

Primary Objective: High Availability and Fast Access to a Wide Variety of Data

Swift provides highly durable storage for large data stores containing a wide variety of data that a large number of clients can read and write.

Theory of Operation

Swift software architecture includes the following elements:

- **Proxies:** Handle all incoming API requests.
- **Objects:** The data being stored.
- **Containers:** An individual database that contains a list of object names.
- **Accounts:** An individual database that contains the list of Container names.
- **Ring:** A map of logical names of data (defined by the Account/Container/Object) to locations on particular storage devices in the cluster.
- **Partition:** A 'bucket' of stored data. Multiple Partitions contain the Objects, Container databases, and Account databases. Swift replication services move and copy partitions to maintain high availability of the data.
- **Zone:** A unique isolated storage area contained in the cluster, which might be a single disk, a rack, or entire data center. Swift tries to maintain copies of partitions across three different Zones in the cluster.
- **Auth:** An optional authorization service, typically run within Swift as WSGI middleware.

To see how these elements interact, let's look at a couple scenarios.

Upload

Through a REST API, a client makes an HTTP request to `put` an object into an existing Container. The Proxy passes the request to the system. Using the Account and Container services, the logical name of

the Object is identified. The Ring then finds the physical location of the partition on which the object is stored. Then, the system sends the data to each storage node, where it is placed in the appropriate Partition. At least two of the three writes must be successful before the client is notified that the upload completed successfully. If storage areas are not available, the Proxy requests a handoff server from the Ring and places it there. Finally, the system asynchronously updates the Container database to reflect that there is a new object in it.

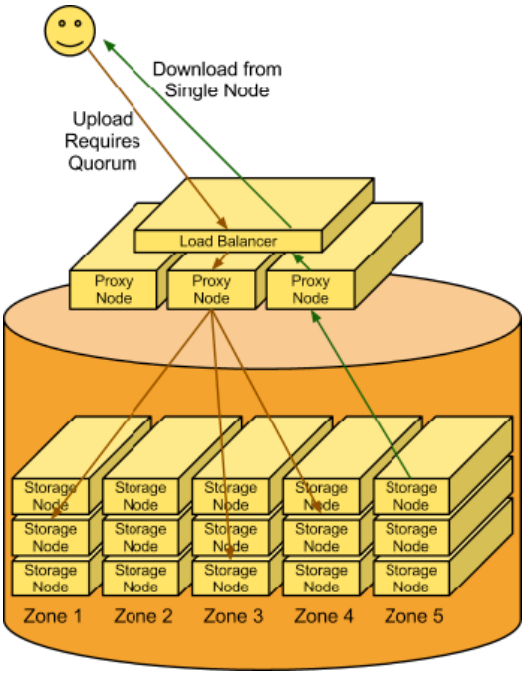


Figure 1. Swift Cluster Software Architecture

Download

Through the API, a client requests an object from the data store. The object's name and partition location are identified in the same manner, and a lookup in the Ring reveals which storage node contains the Partition. A request is made to one of the storage nodes to fetch the object. If the download fails, requests are made to the other nodes. Streamed data to or from an object store is never spooled by the Proxy.

Replication

In order to ensure there are three copies of the data everywhere, replicators continuously examine each Partition. For each local Partition, the replicator checks copies in other Zones for any differences. When it discovers differences among Partitions, it copies the latest version across the Zones.

Other Housekeeping Services

Other housekeeping services run to maintain system durability and keep files updated. For more information on housekeeping services, refer to the Swift documentation (swift.openstack.org).

Configuration and Deployment

Swift architecture comprises many services running on various hardware nodes. Depending on the infrastructure, business processes, and available hardware, some services can reside on the same hardware.

Swift Topology

Swift software runs on many servers/nodes (see Figure 2):

- Proxy node(s) – Server(s) that run Proxy services.
- Auth node - an optionally separate node that runs the Auth service separately from the Proxy services.
- Storage node(s) – One or more server(s) with local storage that run Account, Container, and Object services.

The Proxy and Auth nodes face the public network, with a private switch isolating the storage nodes from the public network.

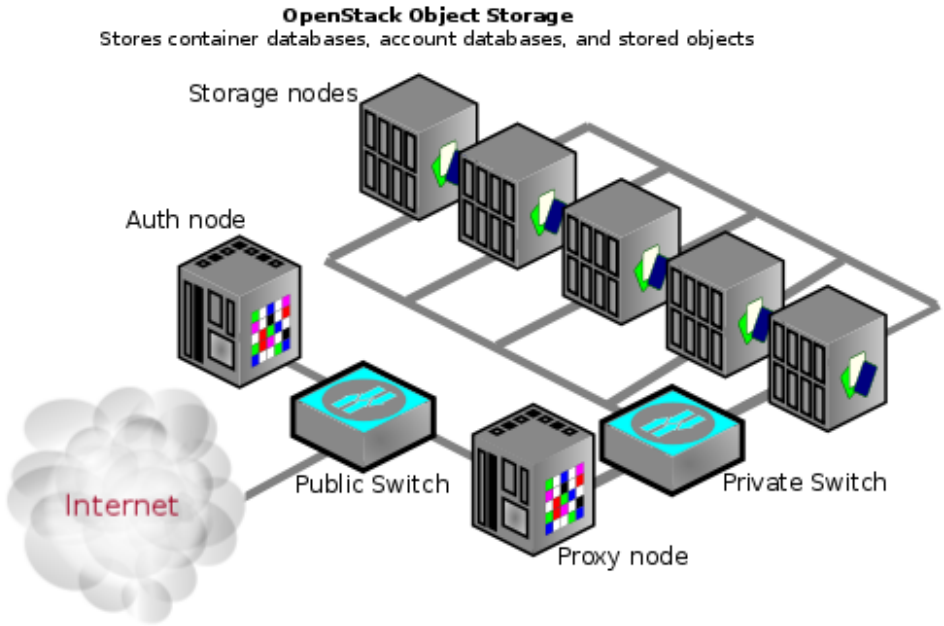


Figure 2. Swift Topology

Access Tier

Large-scale deployments often isolate an "Access Tier" to field incoming API requests, move data in and out of the system, provide front-end load balancers, Secure Sockets Layer* (SSL) terminators, authentication services, and to run the proxy server processes. Having these servers in their own tier enables read/write access to be scaled out independently of storage capacity. As this is an HTTP addressable storage service, a load balancer can be incorporated into the access tier.

This tier typically comprises a collection of Intel Xeon E3 processor-based single-socket servers or Intel® Xeon® E5 processor-based dual-socket servers. Machines in this tier use a moderate amount of RAM and are network I/O intensive. Proxy nodes should be provisioned with at least two 10 Gbps network interfaces, or multiples thereof. One faces the incoming requests on the public network and the other accesses the private network to the object storage nodes. Depending on the expected activity level, other nodes only facing the public network can have single or multiple interfaces.

Factors to Consider

For most publicly facing deployments as well as private deployments on a wide-reaching corporate network, SSL should be considered. However, SSL adds significant server processing load, unless the server processor has built-in hardware support for encryption/decryption, such as Intel® Advanced Encryption Standard—New Instructions (Intel® AES-NI). More capacity in the access layer might be needed if hardware support is lacking. SSL may not be required for private deployments on trusted networks.

Storage Nodes

Storage servers should contain an equal amount of capacity on each server. Depending on your objectives and needs, these can be distributed across servers, racks, and even data centers.

Storage nodes use a reasonable amount of memory and CPU. Metadata needs to be readily available to quickly return objects. The object stores run services not only to field incoming requests from the Access Tier, but to also run replicators, auditors, and reapers. Object stores can be provisioned with a single 1 Gbps or 10 Gbps network interface, depending on the expected workload and desired performance. More interfaces might be appropriate for servers with more processor cores.

Currently 2TB or 3TB Serial Advance Technology Attachment (SATA) disks deliver good price/performance value. Desktop-grade drives offer affordable performance when service and support are responsive to handle drive failures; enterprise-grade drives are an option when this is not the case.

Factors to Consider

The latest generation of Intel Atom processor-based microservers and microservers built on the new Intel Xeon E3 processor work well for storage servers. With up to eight-thread multi-processing, they are responsive and highly energy efficient, and cost less than higher-end multi-socket servers. With up to 16 PCIe ports, they also support an ample number of Intel® Solid State Drives (Intel® SSDs) or traditional HDDs, while delivering the performance necessary to service requests, even under heavy workloads.

Swift does not use Redundant Array of Inexpensive Disks (RAID); each request for an object is handled by a single disk. Therefore, disk performance impacts response rates, and Intel SSDs should be implemented for ultra-fast responsiveness when desired.

To achieve apparent higher throughput, the object storage system is designed to support concurrent uploads and downloads. Multi-core processors, like the Intel Atom processor and Intel Xeon E3 processor, offer multi-processing for concurrent operations. However, the network I/O capacity should match the desired concurrent throughput needs for reads and writes.

Performance Considerations

Overall Strategy

A simple deployment is to install the Proxy Services on their own servers and all of the Storage Services across the storage servers. This allows easy deployment of 10 Gbps networking to the proxy and 1 Gbps to the storage servers. It also helps keep load balancing to the proxies more manageable. Storage services scale as storage servers are added, and it's easy to scale overall API throughput by adding more Proxies.

If you need more throughput to either Account or Container Services, you can deploy the services on their own servers and use faster SAS or SSD drives to get quicker disk I/O.

Nodes

In the front-end, the Proxy Services are CPU- and network I/O-intensive. Select hardware accordingly to handle the amount of expected traffic on these nodes. More cores can service more requests. As already pointed out, if you are using 10 Gbps networking to the proxy, or are terminating SSL traffic at the proxy, greater CPU power will be required if the server does not support hardware-enhanced encryption and decryption.

The Object, Container, and Account Services (collectively, the Storage Services) are disk- and network I/O-intensive. As already mentioned, SSDs should be considered for maximum responsiveness from storage nodes. As shown later in this guide, Intel Atom processor- and Intel Xeon E3 processor-based microservers are quite capable as storage nodes.

Load balancing and network design is left as an exercise to the reader, but this is a very important part of the cluster, so time should be spent designing the network for a Swift cluster.

Ring Configuration

It is important to determine the number of partitions that will be in the Ring prior to configuring the cluster. Consider a minimum of 100 partitions per drive to ensure even distribution across the drives. Decide the maximum number of drives the cluster will contain, multiply that by 100, and then round up to the nearest power of two.

For example, for a cluster with a maximum of 5,000 drives, the total number of partitions would be 500,000, which is close to 2^{19}, rounded up.

Keep in mind, the more partitions, the more work the replicators and other backend jobs have to do, and the more memory the Rings consume. The goal is to find a good balance between small rings and maximum cluster size.

It's also necessary to decide on the number of replicas of the data to store. Three (3) is the current recommendation, because it has been tested in the industry. The higher the number, the more storage is used, and the less likely you are to lose data.

Finally, you'll need to determine the number of Zones in the cluster. Five (5) is the current recommendation according to Swift documentation. Having at least five zones is optimal when failures occur. Try to configure the zones in as high a level as possible to create as much isolation as possible. Consider physical location, power availability, and network connectivity to help determine isolated Zones. For example, in a small cluster you might isolate the Zones by cabinet, with each cabinet having its own power and network connectivity. The Zone concept is very abstract; use it to best isolate your data from failure.

General Server Configuration

Swift uses `paste.deploy` (http://pythonpaste.org/deploy/) to manage server configurations. The configuration process for `paste.deploy` is rather particular; it is best to review the documentation to be sure you configure servers optimally.

Memcached Considerations

Memcached is an open-source, multi-threaded, distributed, Key-Value caching solution that can cache "hot" data and reduce costly trips back to the database to read data from disk. It implements a coherent in-memory RAM cache that scales across a cluster of servers. Several Swift Services rely on Memcached for caching certain types of lookups, such as auth tokens and container/account existence.. Memcached exploits the fact that the typical access time for DRAM is orders of magnitude faster than disk access times, thus enabling considerably lower latency and much greater throughput.

Swift does not cache actual object data. Memcached should run on any servers that have available RAM and CPU capacity. The *memcache_servers* config option in the *proxy-server.conf* file should contain all memcached servers. For more information, see Intel's "Configuration and Deployment Guide For Memcached on Intel® Architecture".

Storage Server Benchmarking for Swift

Object storage in cloud infrastructure can easily scale quickly as storage demands rapidly increase, especially considering the growing need to contain video and other large media. The power demands at scale become critical and more strongly drive hardware choices for storage nodes. Thus, energy-efficient, high-performance servers, such as microservers, become more attractive in cloud deployments.

Intel completed studies on storage node performance from microservers built with the latest generation of Intel Atom processors and Intel Xeon E3 processors. The results show the value of these latest generation microservers to enable high-performance and energy-efficient Swift object storage nodes.

Because SSDs offer much faster response times over traditional spinning HDDs, Intel also compared storage node performance between HDDs and Intel SSDs. The results support a strong argument for SSDs where storage responsiveness is desired.

COSBench Platform

The Swift storage benchmarks reported here used the COSBench* platform to create loads and gather performance data. COSBench is an open source, Intel-developed benchmarking tool to measure Cloud Object Storage performance, including Amazon S3, OpenStack, and Swift. COSBench is freely distributed under the Apache V2 license and is available at https://github.com/intel-cloud/cosbench. You can learn more about COSBench at http://www.zmanda.com/pdf/cosbench-openstack.pdf

System Under Tests (SUT) Configurations

The microservers systems under test (SUT) used in the benchmarks comprise a range of servers offering important characteristics needed in today's cloud storage solutions. Only the storage servers were tested in these studies; Proxy server, while part of the configuration, was not measured. Table 1 lists the storage server configurations.

Table 1. Storage Server Benchmark Configurations

Configuration	Intel® Xeon® E3-1220L v3 Processor	Intel® Xeon® E3-1230L v3 Processor	Intel® Xeon® E3-1265L v3 Processor	Intel® Atom™ Processor (codenamed Avoton)	Intel® Atom™ Processor S1260
Platform	See Table 2	See Table 2	See Table 2	See Table 2	Codenamed DoubleCove
BIOS	S1200RP.86B.01.01.2003	S1200RP.86B.01.01.2003	S1200RP.86B.01.01.2003	EDVLINT1.86B.0018.D05.1306021543_MPK	
BIOS Setting	Cstate off, Intel® HT on	Cstate off, Intel® HT on	Cstate off, Intel® HT on	C state and p states disabled "Active Refresh" "CKE Power Down"	Default
CPU GHz	1.1	1.8	2.5	2.4	2
# Cores	2	4	4	8	2
# Nodes	3	3	3	3	3
Sockets/ Node	1	1	1	1	1
Memory(GB)/ Node	32	32	32	16	8
# DIMMs	4	4	4	2	2
DIMM size (GB)	8	8	8	8	4
# Channels	2	2	2	2	
Memory Speed	1600	1600	1600	1600	1333
OS	Ubuntu Server 12.04	Ubuntu Server 12.04	Ubuntu Server 12.04	Ubuntu Server 12.04	Ubuntu Server 12.04
NIC Port Speed	10 Gbps	10 Gbps	10 Gbps	1 Gbps	1 Gbps
NIC Ports/ Node	1	1	1	4	2
NIC Location	Niantic* 10G	Niantic* 10G	Niantic* 10G	On board	On board
Boot Drive					
Quantity	1	1	1	1	1
RPM/SSD	SSD	SSD	SSD	SSD	SSD
Size	160GB	160GB	160GB	160GB	160 GB
SATA	Yes	Yes	Yes	Yes	Yes
Storage Drive (SSD)					

Quantity	5	5	5	3	1
RPM/SSD	SSD	SSD	SSD	SSD	SSD
Size	250GB	250GB	250GB	160GB	160 GB
SATA Speed	3Gb/s	3Gb/s	3Gb/s	3Gb/s	3Gb/s
Storage Drive (HDD)					
Quantity	5	5	5	3	NA
RPM/SSD	HDD	HDD	HDD	HDD	NA
Size	1TB	1TB	1TB	1TB	NA
SATA Speed	3Gb/s	3Gb/s	3Gb/s	3Gb/s	NA

Table 2. Storage System Under Test (SUT) Platform Specifications

Component Supported	Specification
CPU Cores	8
Core Frequency Base / Turbo	2.4 GHz/2.6 GHz
Memory Channels	2
DIMMs/Channel	2
Memory Type	DDR3L
Memory Frequency	1600
Maximum Memory Capacity	32 GB
PCIe* Lanes – Maximum	16
PCIe Controllers	4
Gb Ethernet Ports/Speed	4, 2.5 Gb Ethernet
SATA 3 Ports	2
SATA 2 Ports	4
USB Ports	4

Five workloads (Table 3) stressed different aspects of the configurations.

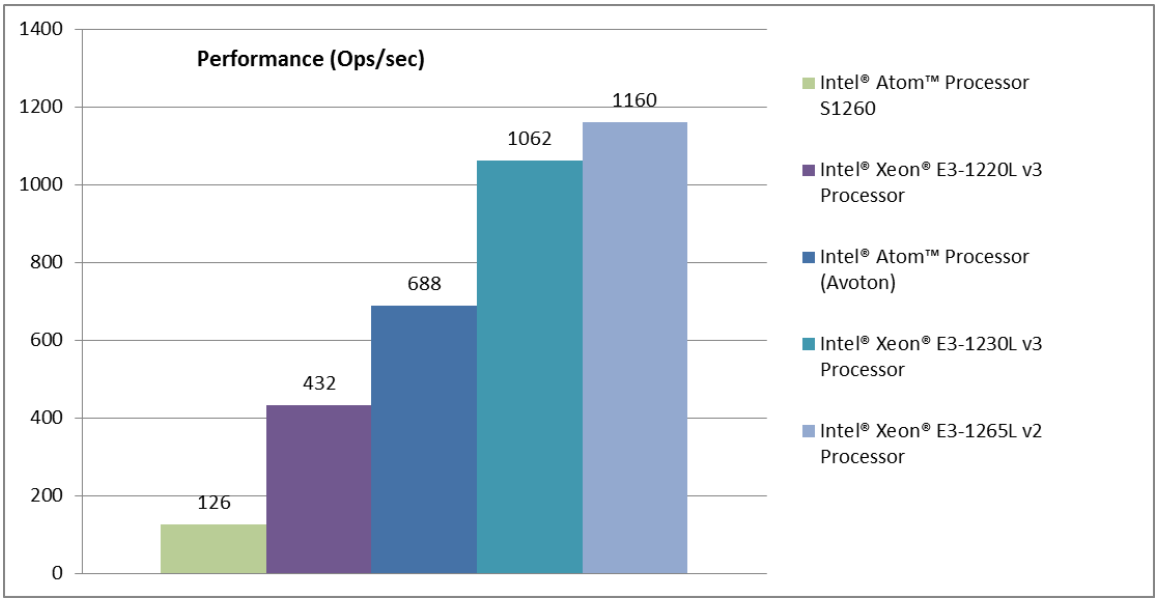
Table 3. Benchmark Workloads

Workload	Test	Rationale
LrgRead	Randomly read objects of 1MB, 100MB, and 1GB	Stress Read IO bandwidth from the object store server
LrgWrite	Randomly Write objects of 1MB, 100MB, and 1GB	Stress Write IO bandwidth to the object store server
SmlWrite	Randomly write objects of 100B, 1KB, and 10KB from multiple clients	Stress the container servers ability to track user file data in the SQL-lite database, by applying a high write rate, which is replicated across 3 servers
SmlRead	Randomly read objects of 100B, 1KB, and 10KB from multiple clients	Stress the Proxy servers ability to process user file data
LrgRWMix	Present a randomly selected R/W mix of 90/10 of 1MB, 100MB, and 1GB files sizes	Test for locks inside the databases and other stores due to concurrent reads and writes to the system

Results

Benchmark results[1,2] (see Figure 3) clearly show that the new Intel Atom processor codenamed Avoton-based microserver outperforms the previous-generation Intel® Atom Processor S1260 server by over 5X, and does so more efficiently—more than 3.5X performance/watt improvement. The latest generation Intel Xeon E3 processor-based microservers offer both performance above the latest generation Intel Atom processor-based microserver and good energy efficiency.

In large cloud data centers, where server counts can scale quickly, conserving power by using energy-efficient hardware becomes more critical. Microservers built on the Intel Atom processor codenamed Avoton deliver the highest efficiency with good performance, making this configuration an excellent balance where performance, price, and power consumption are important factors.



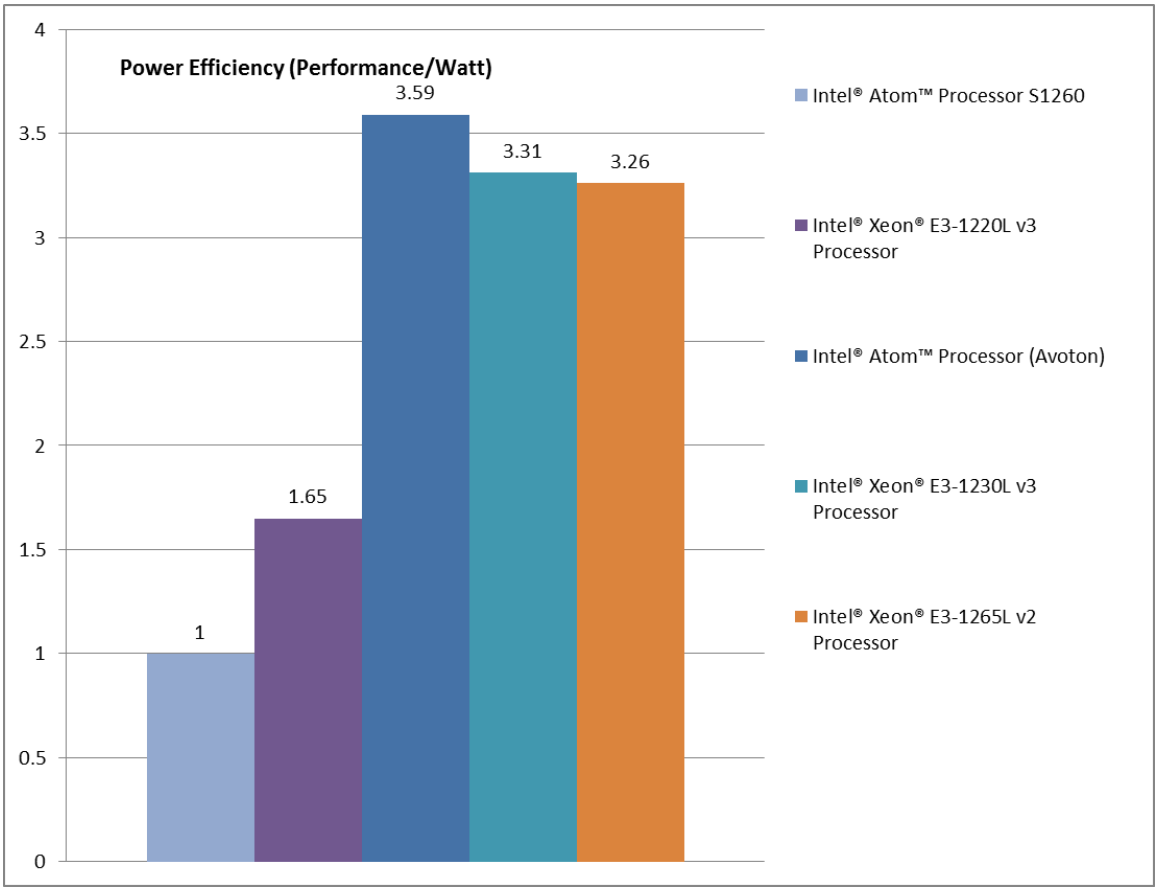


Figure 3. Benchmark Results

These results were achieved using Intel SSDs for both boot and storage drives. Additionally, the benchmarks revealed the impact SSDs have on performance and efficiency compared to SATA 7200 rpm and Serial Attached SCSI (SAS) 10k rpm storage drives. SSDs performed as much as 4.9X faster, as shown Table 4, with more than 5X efficiency.

Table 4. SSD vs. HDD Performance Results

	Performance (Ops/sec)			
Processor	**Intel® SSD**	**7.2k HDD**	**10k SAS HDD**	**Improvement**
Intel Atom Processor (Avoton)	688	140	341	4.9X/2.1X
Intel Xeon E3-1220L v3 Processor	432	145	NA	3X
Intel Xeon E3-1265L v3 Processor	1160	238	NA	4.9X
Intel Xeon E3-1230L v3 Processor	1062	327	NA	3.2X
	Efficiency (Performance/Watt)			
Intel Atom Processor (Avoton)	3.59	0.66	1.56	5.4X/2.3X
Intel Xeon E3-1220L v3 Processor	1.65	0.46	NA	3.6X
Intel Xeon E3-1265L v3 Processor	3.26	0.64	NA	5.1X
Intel Xeon E3-1230L v3 Processor	3.31	0.93	NA	3.6X

Finally, benchmarks offered a view into performance benefits of adding storage drives to each storage node as shown in Table 5, with nearly linear scaling for each drive added.

Table 5. Storage Drive Scaling Results (Intel Atom processor codenamed Avoton)

Metric	Number of Drives		
	1	2	3
Performance (Operations/sec)	184	327	638
Efficiency (Performance/Watt)	0.95	1.61	3.05

These benchmarks reveal the capability of latest-generation Intel Atom processor- and Intel Xeon processor-based microservers when used for object storage servers in a Swift cluster, making these attractive systems for scalable cloud infrastructure.

Tuning and Optimization

General Service Tuning

Most services support either a *worker* or *concurrency* value in the settings. This allows the services to make effective use of the cores available. A good starting point to set the concurrency level for the proxy and storage services is twice (2X) the number of cores available. If more than one service shares a server, then some experimentation may be needed to find the best balance.

Set the *max_clients* parameter to adjust the number of client requests an individual worker accepts for processing. The fewer requests being processed at one time, the less likely a request will consume the worker's CPU time or block the OS. The more requests being processed at one time, the more likely one worker can utilize network and disk capacity.

On systems that have more cores and more memory, you can run more workers. Raising the number of workers and lowering the maximum number of clients serviced per worker can lessen the impact of CPU-intensive or stalled requests.

The above configuration settings are suggestions; test your settings and adjust to ensure the best utilization of CPU, network connectivity, and disk I/O. Always refer to the Swift documentation for the most recent recommendations.

Filesystem Considerations

Swift is rather filesystem agnostic; the only requirement is the filesystem must support extended attributes (xattrs). XFS has proven to be the best choice. Other filesystem types should be thoroughly tested prior to deployment.

Summary

Open source Swift object storage software provides a solution for companies interested in creating their own private cloud storage service or building a service for other production purposes. Swift is easy to use, and it supports web-scale storage services compatible with web applications used today. Swift is provided under the Apache 2 open source license, is highly scalable, extremely durable, and runs on

industry-standard hardware, such as microservers based on the latest generation of Intel Atom processor or Intel Xeon E3 processors. Swift also has a compelling set of tools available from third parties and other open source projects.

Server power consumption has become a critical driving factor in data centers. In benchmarks, microservers based on the latest generation of Intel Atom processor and Intel Xeon E3 processors deliver exceptional performance and performance/watt as Swift storage servers. These multi-core microservers outperformed previous generation, highly efficient Intel Atom processor S1260 servers. With massive object storage deployments typical today, the latest generation of Intel Atom processor- and Intel Xeon E3 processor-based microservers offer attractive choices for Swift cluster deployments.

Learn More

For more information, visit the home of Swift at the OpenStack web site (www.openstack.org) and SwiftStack*, a Swift object storage service provider (www.swiftstack.com). For more information on Intel Atom processor-based servers, visit http://www.intel.com/content/www/us/en/servers/microservers.html

Appendix A – Additional Resources

http://docs.openstack.org/developer/swift/ for Swift documentation.

http://docs.openstack.org/developer/swift/deployment_guide.html for Swift deployment, configuration, and tuning information.

“Configuration and Deployment Guide For Memcached on Intel® Architecture”, Intel Corporation

[1] Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

[2] Configurations: see Tables 1 and 2. **For more information go to http://www.intel.com/performance.**

Results have been estimated based on internal Intel analysis and are provided for informational purposes only. Any difference in system hardware or software design or configuration may affect actual performance.

Intel® Hyper-Threading Technology available on select Intel® Core™ processors. Requires an Intel® HT Technology-enabled system. Consult your PC manufacturer. Performance will vary depending on the specific hardware and software used. For more information including details on which processors support HT Technology, visit http://www.intel.com/info/hyperthreading.