openstack®

High Availability of Live Migration

Ala Raddaoui
Alexandra Settle
John Garbutt
Sarafraj Singh

openstack®

Table of contents

Introduction

Over the last few years, many enterprise customers have moved application workloads into public and private clouds, such as those powered by OpenStack. This trend is projected to grow significantly until 2020. Moving to the cloud offers customers lower costs and a consolidation of virtual estates, and they can benefit from OpenStack's increased manageability.

Host maintenance is a common task in running a cloud—rebooting to install a security fix, patching the host operating system, replacing hardware because of an imminent failure. In these cases, live migration enables the administrator to move a virtual machine (VM) to an unaffected host before such impacting maintenance is performed on the affected host, which ensures almost no instance downtime during the normal operations of the cloud.

To live-migrate an instance is to move its VM to a different OpenStack Compute server while the instance continues running. During the Ocata release, the OpenStack Innovation Center (OSIC) benchmark tested live migration to discover the best way to move forward with non-impacting cloud maintenance. Operators use live-migration to avoid VM downtime when doing host maintenance. This is particularly important when workloads on the VM are not architected to deal with VM failures.

The OSIC team deployed two 22-node OpenStack clouds using OpenStack-Ansible to test two types of live migration:

1. One with local storage where the team could test block storage live migration (migration of both VM memory (RAM) and disk)
2. One with a shared storage back end based on Ceph to test non-block storage live migration (migration of VM memory (RAM) only). The disk is on a remote shared medium and hence does not need copy over during live migration.

Problem description

The OSIC team set out to prove that live-migration could be successfully managed due to the misconception that live-migration is unreliable and it does not work with local storage. Until recently, there has been little upstream testing of live-migration, a problem that has been discussed at previous OpenStack summits.

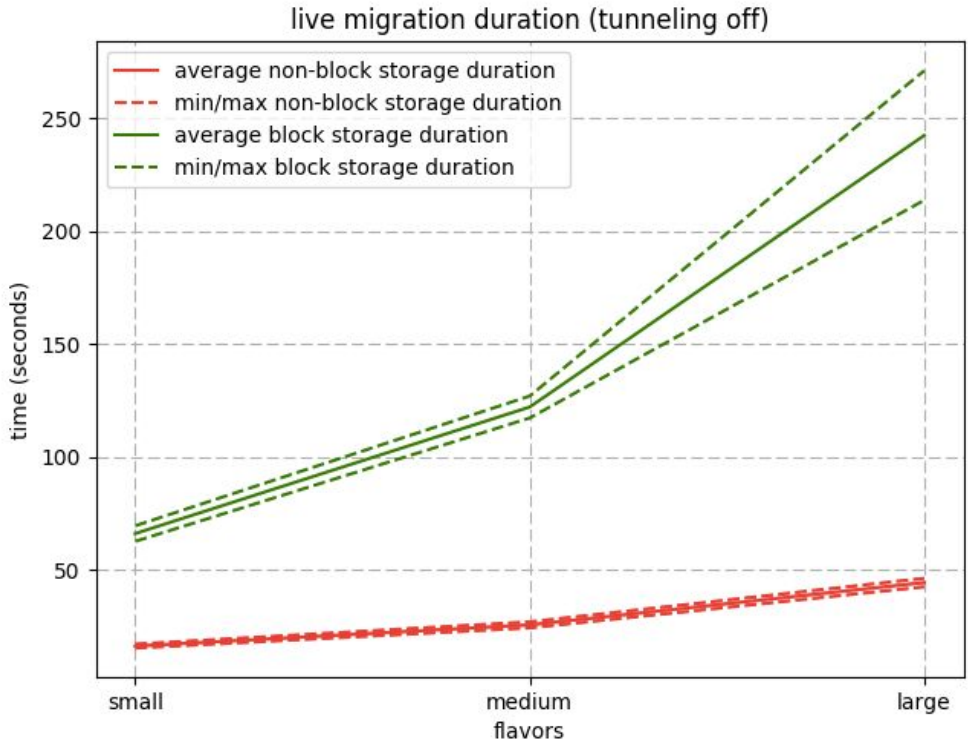
Early support for live-migration was added in Cactus, and in more recent releases, live-migration has improved stability. Before Mitaka (before API micro-version 2.24), the user had to specify an extra boolean argument to live migrate APIs if using local storage. If this variable was set to use local storage while source or destination is using shared storage, live migration would fail. This forced the user to figure out what kind of storage each VM is using before calling the live migrate API. This was causing issues with automation on larger cloud deployments.

What we found

Over the course of the Ocata release, we were able to prove that live migration works, both with and without the use of shared storage. When your cloud is not using a shared storage backend, VMs are created with an ephemeral disk local to the compute node and block storage will be used to perform the migration.

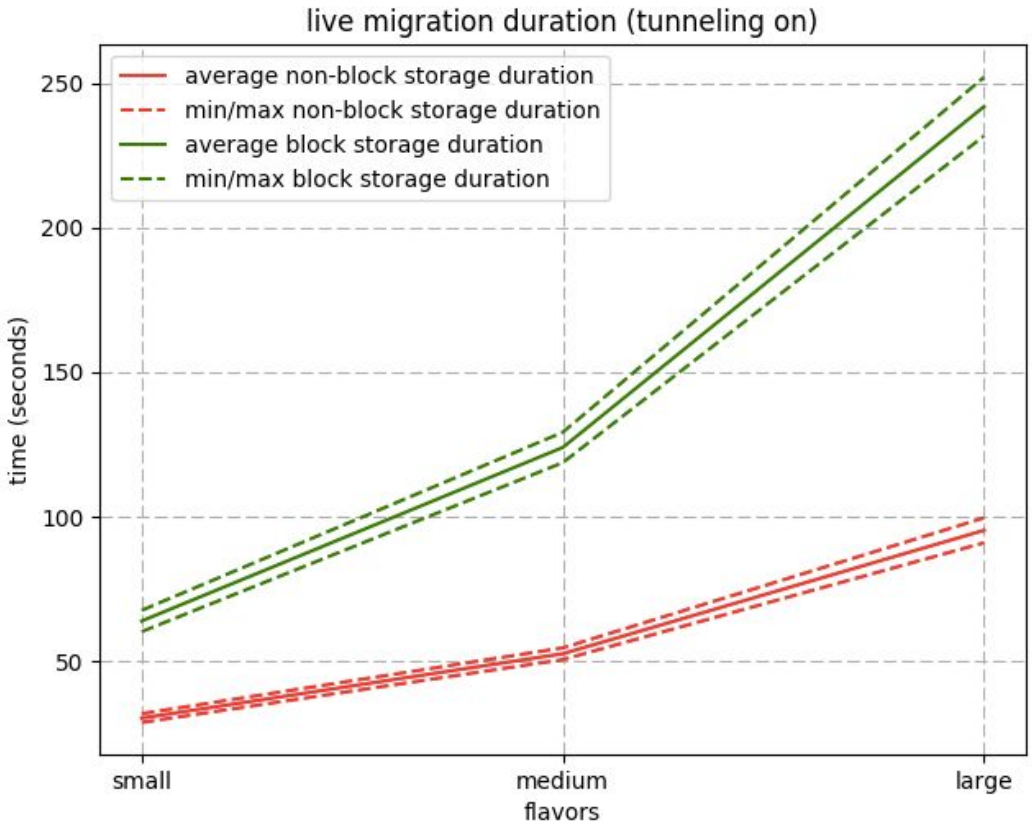
It is worth noting we do not have a graph for the number of failed live-migrations, because there were none during these test runs. Initial test runs uncovered some bugs that were fixed upstream in the Ocata branch, and backported where applicable. After the bugs were addressed, no failures were found. See the Discussion and Conclusion for further analysis of these bugs.

The following graph details the results of testing various sizes of VMs backed by either a local disk (block storage results) or a disk on shared storage (non-block migration results):



What this graph shows is that it is faster to live-migrate a VM that is using shared storage. This is because when performing a block migration, extra time is required to copy the disk between the source and destination host. In addition, the graph indicates that larger VMs take much longer than smaller VMs. The more memory and disk there is to copy between the source and destination, the longer it takes to perform the live migration.

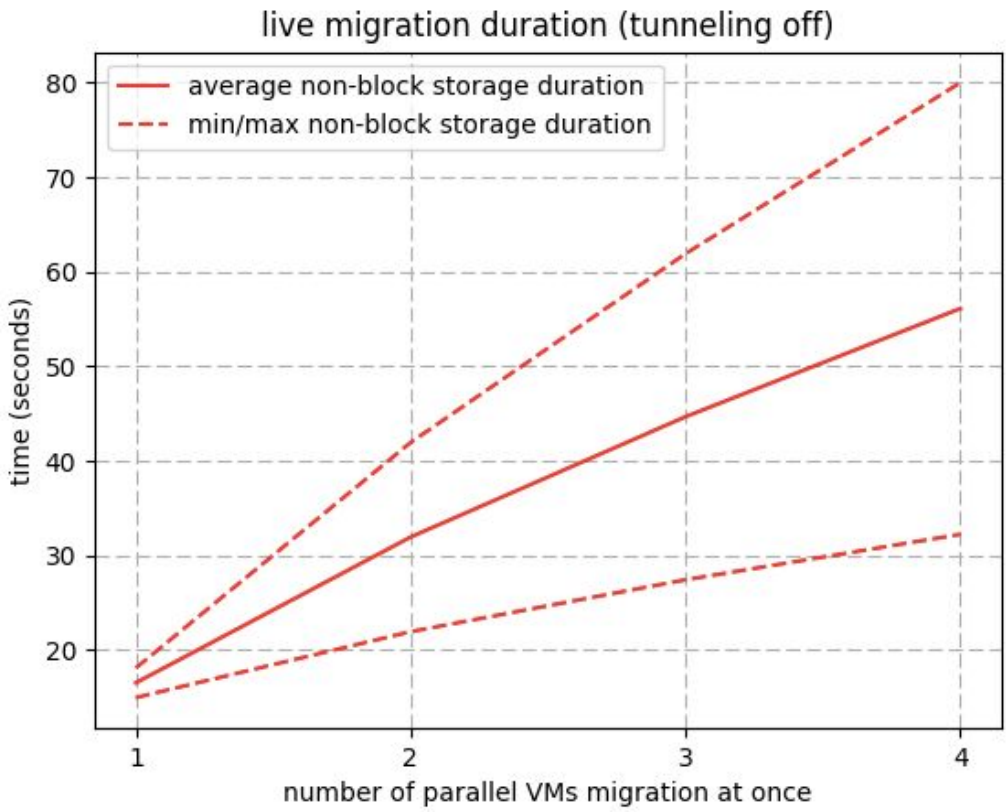
The next graph shows the same tests but with tunnelling turned on. This was executed by adding the following configuration option: *libvirt.live_migration_tunnelled=True*.



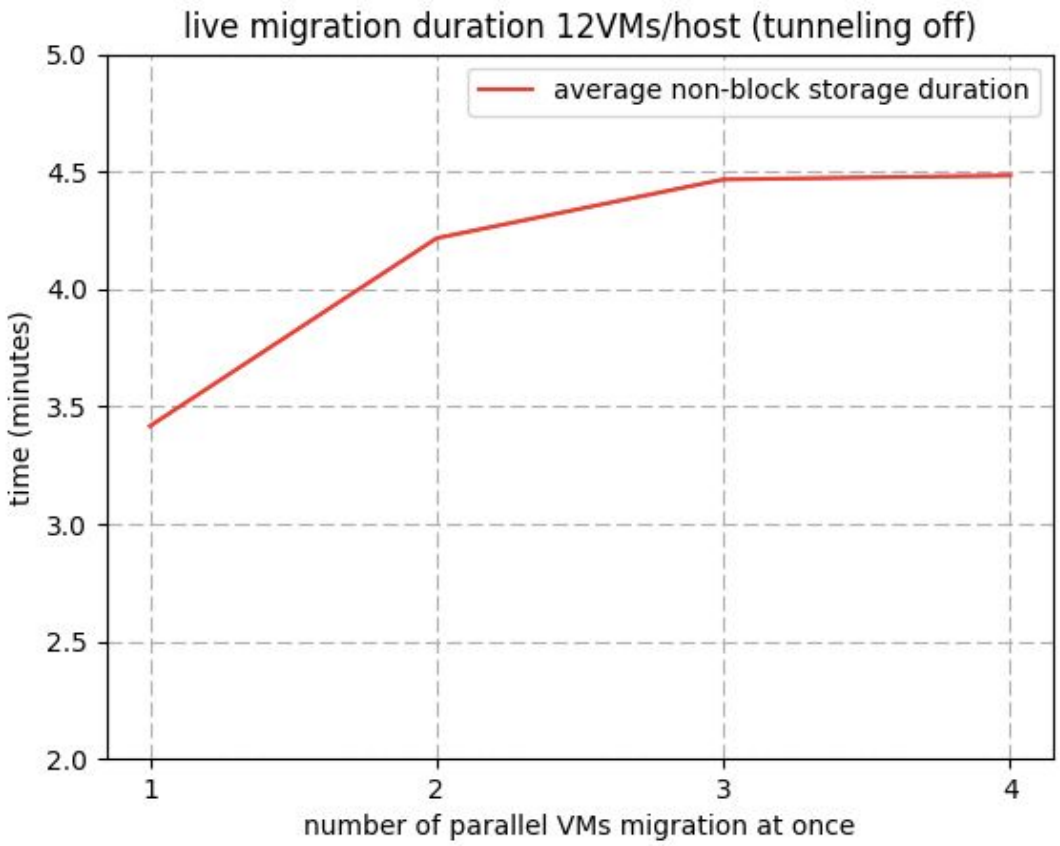
The above graph shows that the time taken is higher than with tunnelling turned off, particularly when performing a non-block live migration. Tunneling is off by default to improve the performance of live migration at the cost of not encrypting the memory copied between the source and destination host. The expectation was to see a bigger performance difference than was observed, however, these results have lead us to conclude that more investigation is likely required.

The team also monitored TCP streams connected to all the VMs being live-migrated. At no time did we see any loss of the TCP stream during the live-migration. As such, there are few results to show or discuss.

The final set of tests focus on the parallel live migrations. We wanted to work out how many live migrations should be done in parallel to move all VMs from one host to the next in the quickest time. These were all done using a medium VM with shared storage and tunnelling off.

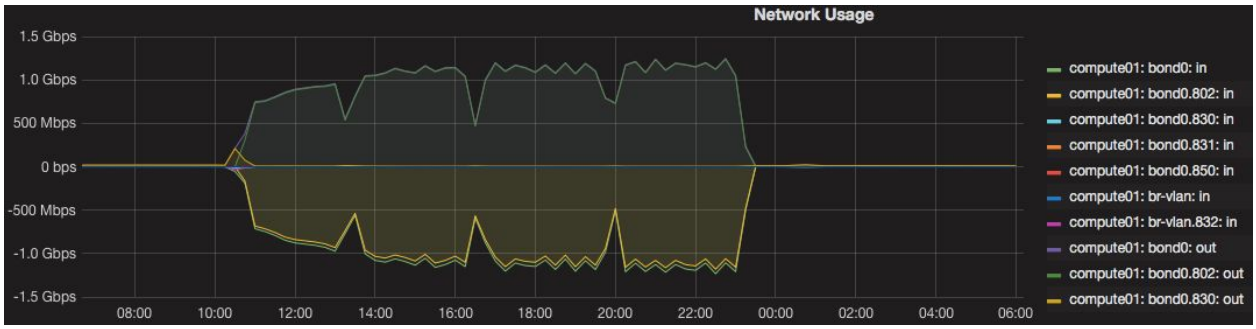


While we expected a live-migration to take longer when running two in parallel compare to running them serially, we were not expecting such a significant increase in the time taken to perform the live-migration. After further investigation, we found that we were using the default configuration setting of *max_concurrent_live_migrations=1*. This means we actually did not perform any live migrates in parallel. Nova silently queued the live migrations we requested to reduce the impact of the live migrations on the network and CPU resources of the hypervisor. If you want to do multiple live migrations in parallel, you need to modify that configuration.



This above graph shows the time it took to empty a host by live-migrating all its VMs in batches of 1, 2, 3, and 4; repeating this operation 40 times. The graph displays the average time taken to empty a host of all 12 medium VMs using live-migration, with the given batch size. Again, these results have been affected by our use of *max_concurrent_live_migrations=1* (the default), which means there were really no live-migration happening in parallel.

The graph below shows the CPU load during creation of the VMs and then the runs with a batch size of 1, 2, 3, then 4. We were wanting to check what additional load is put on the system because of the multiple migrations. You can see the CPU usage is quite similar in each of the runs, which matches the fact that we did not actually perform any live-migrations in parallel:



Methodology

Our goal was to measure the difference between block (local) and non-block (shared) live migration. 44 nodes were allocated to perform those tests. Each node had the following specification:

- Model: HP DL380 Gen9
- Processor: 2x 12-core Intel E5-2680 v3 @ 2.50GHz
- RAM: 256GB RAM
- Disk: 12 x 600GB 15K SAS - RAID10
- NICS: 2x Intel X710 Dual Port 10 GbE

The OSIC team deployed two 22-node OpenStack clouds using OpenStack-Ansible to test two types of live migration:

1. A 22-node Newton OpenStack Cloud (libvirt-1.3.1)
    a. Backed by ephemeral disks collocated with the instance itself in the same compute node.
    b. Tested block storage live migration (migration of both VM memory (RAM) and disk).
    c. The cloud layout is as follows:
        - 1 deployment node
        - 3 controller nodes
        - 3 logging node
        - 9 compute nodes

- 3 cinder nodes
- 3 swift nodes

2. A 22-node Mitaka OpenStack Cloud (libvirt-1.2.2)
   a. Remote storage backend based on Ceph (version 10.2.2).
   b. Tested non-block live migration where only VM memory (RAM) is migrated.
   c. The cloud layout is as follows:
      - 1 deployment node
      - 3 controller nodes
      - 3 logging nodes
      - 10 compute nodes
      - 5 ceph nodes.

The October 2016 OpenStack user survey showed increase use of High Performance Computing (HPC), Big data, Storage, and NFV (Network Functions Virtualization) workloads. When it comes to production-like workloads in the cloud, enough memory utilization, CPU, network I/O, and disk I/O are needed to achieve optimal performance.

To replicate production values during testing, the cloud is filled with workloads to ensure reliable results. While testing live migration, the OSIC DevOps team used Apache Spark Streaming as a driver for their workloads.

Apache Spark is a fast, in-memory cluster, computing framework that manages big data processing requirements with a variety of data sets. These data sets are the source of data (batch versus real-time streaming data) and diverse in nature (text data and graph data). When it comes to production-like workloads in the cloud, enough memory utilization, CPU, network I/O, and disk I/O are needed to achieve optimal performance. Spark uses in-memory data processing and optimizes the resources for the data processing, imitating real-time workloads.

Spark Streaming is an extension plugin for the core Spark API. It provides scalability, high throughput, and fault tolerance processing of live-streaming data. Spark Streaming takes the input in the form of streaming data, divides the data into batches, and feeds the data to the Spark engine. Spark Streaming makes it easy to simulate scalable fault-tolerant streaming of real-time applications.

The OSIC DevOps team developed a template based on OpenStack Heat to use Spark Streaming to simulate production-like workloads on the VMs.

Using a workload generator, the template deployed six VMs and performed the following steps in each VM, with each VM serving as independent Spark cluster and client:

1. Install the Spark cluster.
2. Fetch a large text file from the internet containing a large chunk of data (http://norvig.com/big.txt).
3. Start a netcat server.
4. Start the Spark cluster.
5. Continuously run the edited example *network_wordcount* (offered by the Apache Spark project), which is a Spark Streaming script in the cluster that calculates word count from the streaming data received from the netcat server and registers results to disk.

The OSIC DevOps team created a tool called LVM benchmarker for benchmarking live migration in the OpenStack cloud. It used the Spark-based streaming workload described earlier to simulate the production-like workloads in the VM and benchmark live migration. This simulation made the VM busy with the real-time workload, ensuring that live migration was slower than usual. Then, the tool uses rally to live migrate the workloads and capture the timing of each live migration operation.

Rally is a tool that automates benchmarking and profiling of OpenStack clouds. It is built in a pluggable way which makes it easier to add any testing capability. Taking advantage of that, a plugin was added that allowed the team to benchmark live migration with rally. Using rally for your benchmark tests allow you to capture the time taken for each function executed in the plugin built and therefore capturing the live migration time taken by each VM while benchmarking.

The LVM benchmarker acts as a wrapper of all pieces to perform bootstrapping of the OpenStack Cloud for live migration testing by adding all necessary images, keys, flavors, and eventually the installation of rally. Three flavors were created for the purpose of our testing:

- Small (2 VCPUs, 4GB RAM, 40GB DISK)
- Medium (4 VCPUs, 8GB RAM, 80GB DISK)
- Large(8 VCPUs, 16GB RAM, 160GB DISK)

The three flavours were then iterated over in the following ways, over the configured number of parallel migrations :

1. Launch the VM downtime and TCP stream continuity tests prior to start benchmarking to capture the performance of the live migration.
2. Fill one of the compute nodes with VMs of one of the flavors above and deploy all the Spark Streaming workloads inside them
3. Evacuate the compute node with the workloads to another empty compute node back and forth 20 times in a row using the rally plugin.
4. Register all results generated by the tests (downtime and tcp stream continuity for each VM) and by rally: live migration duration of each VM and finally retrieving all the logs from the compute nodes to debug any failure during the benchmark.

A solid monitoring stack was needed to capture the behaviour of the compute nodes and VMs and track their resource usage when live migration is in process. A TIGK stack (Telegraf, Influx, Grafana and Kapacitor) was put in place to capture, store, display metrics from the systems CPU, RAM, and I/O (network and disk). Each element in that stack has a specific role:

- T = **Telegraf**, a plugin-driven server agent for collecting and reporting metrics
- I = **InfluxDB**, a time series database built from the ground up to handle high write and query loads.
- G = **Grafana**, a web based dashboard that displays metric information.
- K = **Kapacitor**, a data processing framework proving alerting, anomaly detection and action frameworks.

Since the deployment was using OpenStack-Ansible, all OpenStack services are installed inside their own containers. Each component from the stack was installed in many containers residing in the logging hosts to make the stack highly available. Since Telegraf is the element that collects metrics and sends it to the Influx database, Telegraf is installed everywhere even in the created workloads.

Since most web application use protocols like HTTP, HTTPs, FTP, etc which run on top of the transport layer protocol TCP that ensure the reliability, order, and error-prone delivery of a stream of data between applications, another test was built to measure or detect any loss in the tcp stream. To implement this test, all VMs were started with a netcat server. A netcat server uses a TCP protocol for its client connections. The setup of the netcat server was performed by the Heat template. Launching the

test ensured the netcat server was connected, and after each time interval of 0.5 seconds, a consecutive number was sent to the netcat server. At the end of the live migration testing, the test checked if there were any numbers missing in the set received, and any loss was calculated accordingly. In our tests, no TCP connection drops were detected.

After that, to record the downtime of each VM while live migration, another test is built where we sent a ping request to the VM every 0.5 seconds, and recorded if any pings were not responded to. At the end of the live migration, the VM downtime was estimated using the number of non-received ping packets multiplied by the interval delta (0.5 seconds).

Our live migration testing was performed in two phases. In phase one, we tried to distinguish between block and non-block live migration. To do that, we ran the live migrate benchmarker tool on the first cloud backed by ephemeral disks, and again on the second cloud with the remote storage backend , which generate following raw results for block storage tunneling off and tunneling on and following results for shared storage tunneling off and tunneling on. In phase 2 we attempted to profile live migration when performed in parallel. This was conducted in batches of 1, 2, 3, and 4. On each of these batches, we ran the live migrate benchmarker tool after configuring it to run parallel testing on the cloud with the remote storage backend. You can find the results in the LVM tunnelling off file.

Discussion and conclusion

During live-migration we discovered some interesting bugs. Previously, Nova incorrectly tracked live migration progress. Nova had a logic that detected whether live-migration was making progress based on data returned by libvirt and took appropriate measures based on configuration option *live_migration_progress_timeout*. However, it turned out there are several problems with the way we monitored the progress. In production, and stress testing, having *live_migration_progress_timeout* > 0 caused random timeout failures for live migrations that take longer than *live_migration_progress_timeout.* One problem is that *block_migrations* appear to show no progress, as it seems we only look for progress in copying memory. Also, the way QEMU was queried via libvirt breaks when there are multiple iterations of memory copying. We deprecated this option and turned it off by default.

The second bug we found was that live migration was left in migrating as domain not found. During live migration, stress testing we found multiple “Domain not found” errors. There appeared to be a race

condition while trying to undefine the domain, leading to occasional live migration failures. This bug was not correctly handled and an instance was never set to the error state, so it just stayed in the migrating state. We released the fix for this Pike and backported the fix to Ocata.

In conclusion, we were able to prove that live migration works. You can use it to avoid the downtime needing to reboot a host for maintenance. If you decide to empty a host, one live migration at a time works best as it takes shorter time to live migrate the VM and has less impact on VM downtime. We recommend trying to use shared storage where possible. Finally, set the progress timeout to zero if you are using a release prior to Newton.

For more information on how we benchmarked live migration, our discoveries, projected next steps, and the ideal setup for your cloud to have an exceedingly available environment, watch the OpenStack in Motion: Live Migration session from the OpenStack Summit in Boston!

Resources

1. https://wiki.openstack.org/wiki/Heat
2. https://wiki.openstack.org/wiki/Rally
3. https://specs.openstack.org/openstack/openstack-user-stories/user-stories/proposed/ha_vm.html
4. https://github.com/osic/benchmarking_live-migration/blob/master/benchmarker_results/lvm_tun_on_block_storage_sml_after_patch.txt
5. https://github.com/osic/benchmarking_live-migration/blob/master/benchmarker_results/lvm_tun_off_shared_storage_sml_timeout_infinity.txt