A Cluster-Imbalance-Aware Scheduler for VM Live Migration

Adaptive Trigger, Victim and Placement Decisions in an Multi-Node, HA OpenStack Lab

Course: CSE606 – Cloud and Large Scale Computing / CSE540 – Cloud Computing
Supervisor: Prof. Sanjay Chaudhary, Ahmedabad University

Aagam Sheth (AU2544006)
Ahmedabad University
Ahmedabad, India
aagam.s6@ahduni.edu.in

Mahima Parekh (AU2544003)
Ahmedabad University
Ahmedabad, India
mahima.p2@ahduni.edu.in

Yash Shah (AU2340078)
Ahmedabad University
Ahmedabad, India
yash.s14@ahduni.edu.in

Jash Agicha (AU2320253)
Ahmedabad University
Ahmedabad, India
jash.a@ahduni.edu.in

Abstract

OpenStack via Nova supports VM live migration. It does not provide an independent method of tracking resource consumption. It does not rebalance active virtual machines while taking migration expenses into consideration. We present CloudShift. It is a CPU utilisation imbalance aware scheduler. It determines when a migration is required. It decides which virtual machine should be moved. It decides where it should be placed. The goal is to reduce downtime. The goal is to reduce network overhead. The goal is to reduce needless migrations. This work also includes a reproducible multi node OpenStack environment. The environment has 3 controllers and 3 compute hosts. It has a monitoring pipeline. It has shared GlusterFS storage. It has usage based billing. It has a highly available control plane.

Keywords: OpenStack, live migration, scheduling, Kolla-Ansible, GlusterFS

1 Project Statement

1.1 Problem Definition

Cloud providers routinely over-commit CPU and memory, so host utilisation becomes skewed: a few compute nodes saturate while others stay underutilised, degrading tenant performance. Nova's scheduler optimises initial placement but does not rebalance a running cluster, and live migration is typically triggered reactively by an administrator or external tooling [2]. The result is a gap between the migration capability Nova already provides and the workload-aware rebalancing that a self-managing cloud needs. CloudShift closes this gap by continuously monitoring the cluster, selecting a suitable victim VM, choosing a destination that satisfies placement constraints, and executing the migration with minimal downtime and network overhead.

- **Reactive migration:** moves are triggered manually or by external tooling rather than by a closed control loop.
- **Limited runtime balancing:** the Nova Scheduler handles initial placement, not steady-state load.
- **Imbalance under overcommitment:** CPU and memory skew produces hot spots and wasted capacity.
- **Migration complexity:** effective rebalancing must jointly consider victim selection, destination fit, bandwidth, and migration overhead.

1.2 Objectives

1. Set up a reproducible multi-node OpenStack testbed using Kolla-Ansible.
2. Validate baseline KVM/QEMU/libvirt live migration before automation.
3. Develop a real-time monitoring pipeline for host and VM visibility.
4. Design a scheduler that decides when, which VM, and where to migrate.

1.3 Scope of the Project

The project focuses on deployment, shared storage, monitoring, scheduling, migration tooling, and billing on a lab with 3 controller and 3 compute hosts. The project does not implement a custom hypervisor, or a production distributed scheduler. Our focus remains on reproducibility, high availability, and rebalancing. The lab can be used for educational and experimental purposes, and the scheduler is validated first in dry-run mode.

Intended Outcomes.

- An OpenStack cluster deployed with Kolla-Ansible that has end to end live migration validation.
- An independent adaptive scheduler for migration, integrated with OpenStack through its APIs.

- Comparative data on automated vs. manual rebalancing.
- Practical experience with high availability control planes, shared storage, telemetry and billing.

2 Project Design

2.1 System Architecture and Design Overview

CloudShift is an OpenStack based private cloud [3]. It adds an automated scheduling mechanism for managing VM distribution across compute nodes. It combines OpenStack native management services with a custom monitoring and live migration system. This system detects and addresses resource imbalances.

The infrastructure has three controller nodes. These provide management and API services. Three compute nodes host VMs using KVM, QEMU and libvirt. GlusterFS provides shared storage across the compute nodes.

CloudShift works alongside OpenStack native services. The Nova Scheduler selects a host during initial VM creation. CloudShift monitors running workloads on its own. It starts migration when required.

The platform runs on bare metal. Every node boots Ubuntu Server directly on physical hardware. The operating system gets direct access to CPU, memory and disks. KVM is integrated into the Linux kernel. It acts as a type 1 bare metal hypervisor on the compute nodes. Guest VMs run directly on the hardware with near native performance. There is no type 2 hosted hypervisor such as VirtualBox or VMware Workstation in the stack. The OpenStack services themselves are not installed on bare metal binaries. Kolla Ansible deploys them as Docker containers on the controllers. Compute agents run on the compute nodes. This isolates and standardises the control plane. KVM remains the bare metal hypervisor for guest VMs. This combination keeps virtualisation overhead low. It gives libvirt direct control of devices and NUMA topology. It makes live migration of running VMs feasible across the compute nodes.

2.2 Cloud Infrastructure Deployment Design

The infrastructure is deployed using Ansible playbooks and Kolla Ansible [4]. This automates configuration across the nodes. Figure 1 shows the end to end seven phase system workflow. Figure 2 shows the step by step command execution and decision flowchart for setting up the physical lab and deploying the cluster.

The deployment starts with configuring the TP-Link managed switch. Trunk VLANs 10, 20, 30, 40, 50, 60 and 70 are set up on all ports. The controller node is then bootstrapped using `scripts/00-controller01-bootstrap.sh`. SSH key distribution and Ansible installation on the deploy host follow. The Ansible preflight playbook `00-preflight.yml` checks connectivity. It checks network configuration. It checks hostnames and system compatibility. If any preflight check fails, the operator fixes the netplan, hostname or Wi-Fi subnet issues and reruns the playbook. Once preflight passes, `scripts/cpu-check.sh` verifies common Nehalem CPU compatibility across the compute nodes. GlusterFS disks are then prepared using `scripts/storage-manager.sh` and the storage playbook `03a-prepare-glusterfs.yml`. The main site playbook `site.yml` then configures VLAN netplans, the Keepalived floating gateway, lab DNS and GlusterFS replicas.

Once the base infrastructure is ready, `kolla/install.sh` installs Kolla Ansible and merges the inventory. The deployer `kolla/deploy.sh` runs prechecks across all nodes. Any precheck failure is debugged before retrying. Following successful prechecks, Kolla Ansible deploys the OpenStack services as Docker containers. Post deployment configuration generates administrative credentials through `/etc/kolla/admin-openrc.sh`. The cloud setup script `openstack/01-cloud-setup.sh` then creates virtual networks, flavors and images.

Monitoring exporters and billing rules are deployed last using `05-exporter.yml` and `02-billing-rates.sh`. Finally, `live-migration/test.sh` runs an end to end live migration test to verify that the cluster is operational.

2.3 OpenStack Logical Architecture and VM Provisioning

OpenStack follows a service oriented architecture. Individual services communicate through APIs. They also use message queues and shared databases. Figure 3 (page 4) shows the physical and VLAN system architecture. Figure 4 shows the logical service interaction across user access, VM creation, shared storage, monitoring and automated live migration.

Administrators and tenants access OpenStack through Horizon or the CLI. Requests pass through HAProxy and Keepalived. They use the internal VIP 10.0.10.100. Then they reach the Nova API on port 8774. Keystone handles authentication and token validation. When a user requests a VM, the Nova API receives the configuration (Step 1 in Figure 4). This includes the selected image, flavor and network details. The provisioning workflow involves RabbitMQ. RabbitMQ carries asynchronous requests between OpenStack services and forwards the request to the Nova Conductor (Step 2). The Nova Conductor coordinates the provisioning operation and asks the Nova Scheduler for a host (Step 3). The Nova Scheduler queries the Placement API for resource allocation candidates (Step 4). The Placement API returns eligible compute hosts (Step 5). The Nova Scheduler selects a suitable compute host and returns it to the Conductor (Step 6). All four control plane components persist state in the three node MariaDB Galera cluster. The Conductor then dispatches the build request over RabbitMQ RPC (Step 7) and delivers it to `nova-compute` on the selected compute host (Step 8).

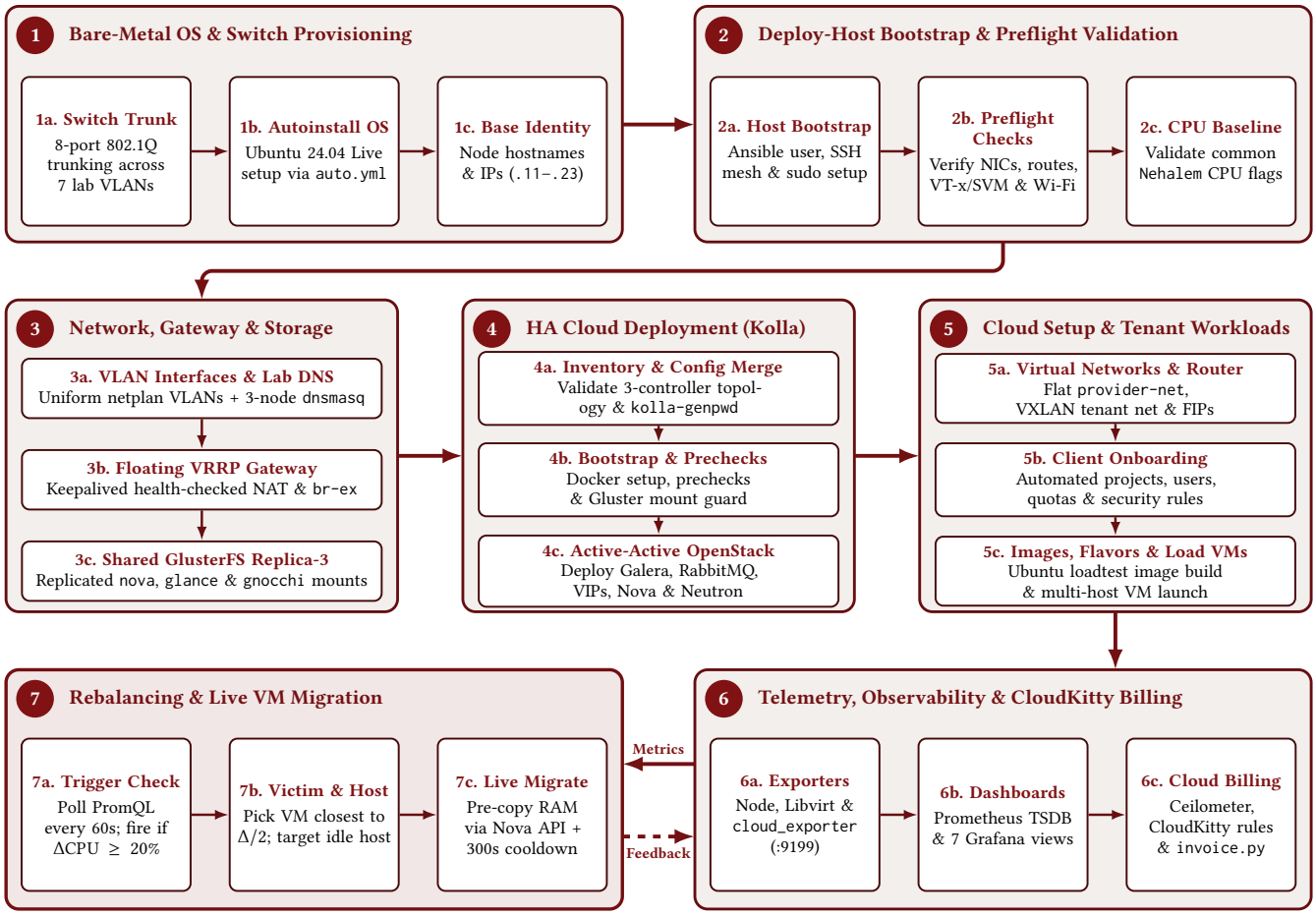

Figure 1. Cloud lab setup and infrastructure deployment workflow.

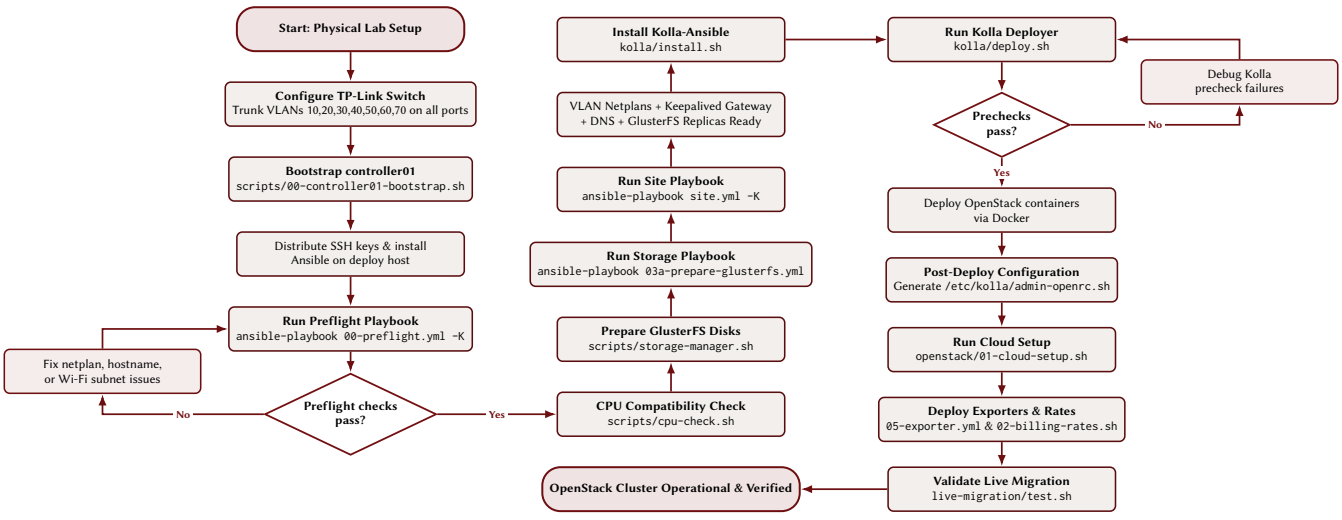

Figure 2. Cloud lab setup and command execution flowchart.

2.3.1 VM Provisioning and Storage. The selected compute node coordinates the resources required to create the VM. It interacts with several OpenStack services. It retrieves the requested operating system image from Glance (Step 9).

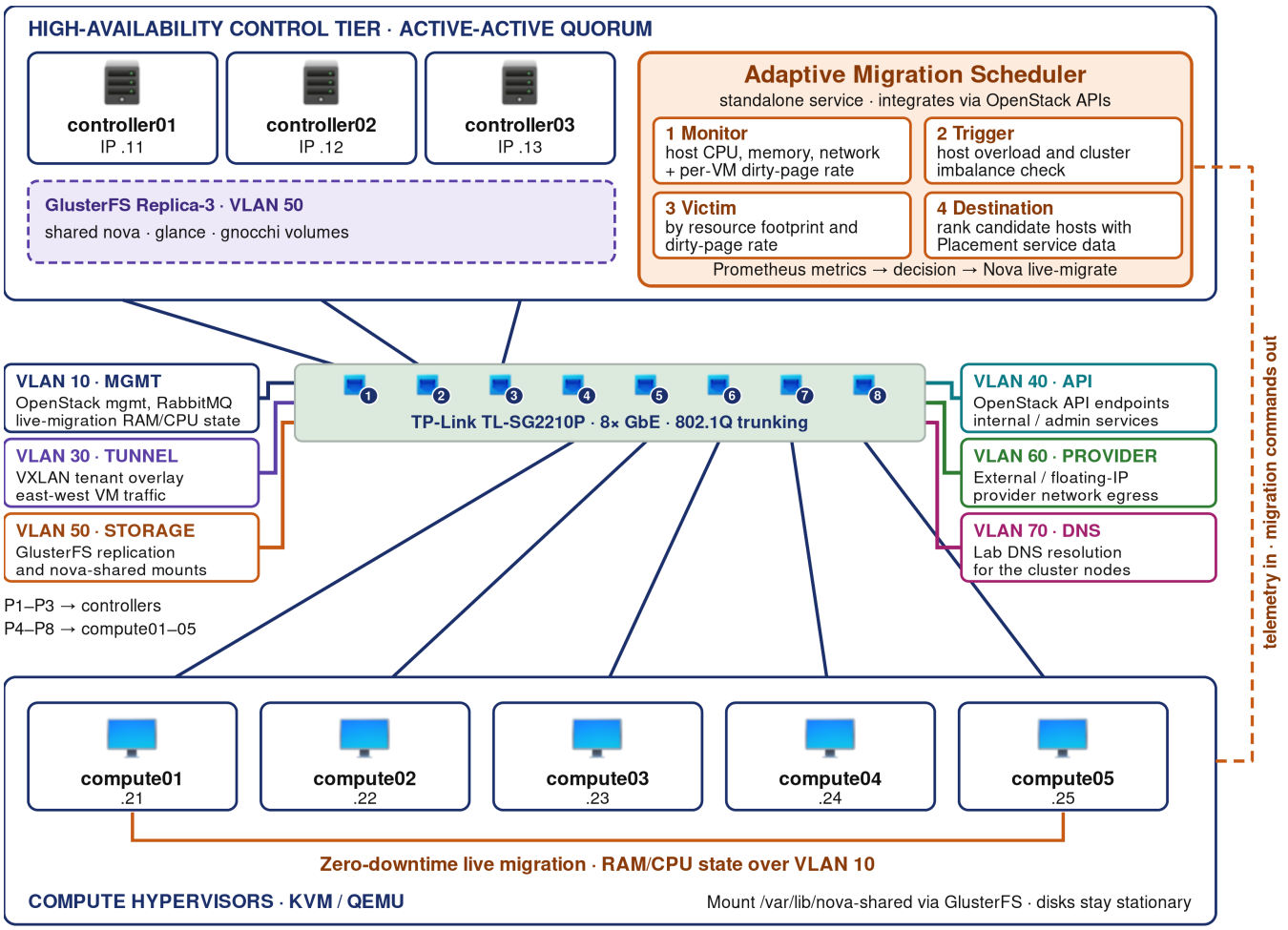

Figure 3. High-level system architecture.

It asks Neutron and Open vSwitch to create the virtual network port (Step 10) and receives the network configuration (Step 11). GlusterFS [5] provides shared storage for the VM disk files (Step 12). Next, `nova-compute` instructs libvirt to spawn the instance (Step 13). Libvirt starts the virtual machine on KVM and QEMU (Step 14). Once the instance starts, its status becomes ACTIVE (Step 15) and the Nova API returns the instance ID and status to Horizon or the CLI (Step 16). GlusterFS uses three replicas. It provides the shared storage path `/var/lib/nova-shared`. The other compute nodes also mount this shared volume. Multiple compute nodes can access the same VM disk files. This avoids transferring the complete disk image during shared storage live migration.

2.3.2 Monitoring and Billing. The custom `cloud_exporter.py` collects OpenStack information. This includes server and migration data. It exposes metrics to Prometheus. Host and VM CPU metrics are also collected. Node exporter and libvirt exporter are used. Prometheus [6] is exposed on port 9091. It stores metrics for querying. Grafana [7] provides monitoring dashboards. Ceilometer and Gnocchi support telemetry collection. CloudKitty provides billing functionality. These components supply the monitoring information used by CloudShift.

2.4 CloudShift Automated Load Balancing and Live Migration

CloudShift monitors compute node CPU utilisation. It starts VM migration when resource imbalance exceeds the configured conditions. The Nova Scheduler handles initial VM placement. CloudShift manages the distribution of already running VMs. Figure 5 shows the workflow sequence.

The CloudShift scheduler `scheduler.py` queries Prometheus for CPU utilisation metrics using PromQL (CloudShift Step 1 in Figure 4). The monitoring interval is 60 seconds. Figure 5 shows the detailed sequence. It identifies the busiest and least utilised hosts. It calculates the difference in their CPU utilisation. Migration is considered when this difference exceeds the configured threshold. The cooldown period must have expired. No migration must

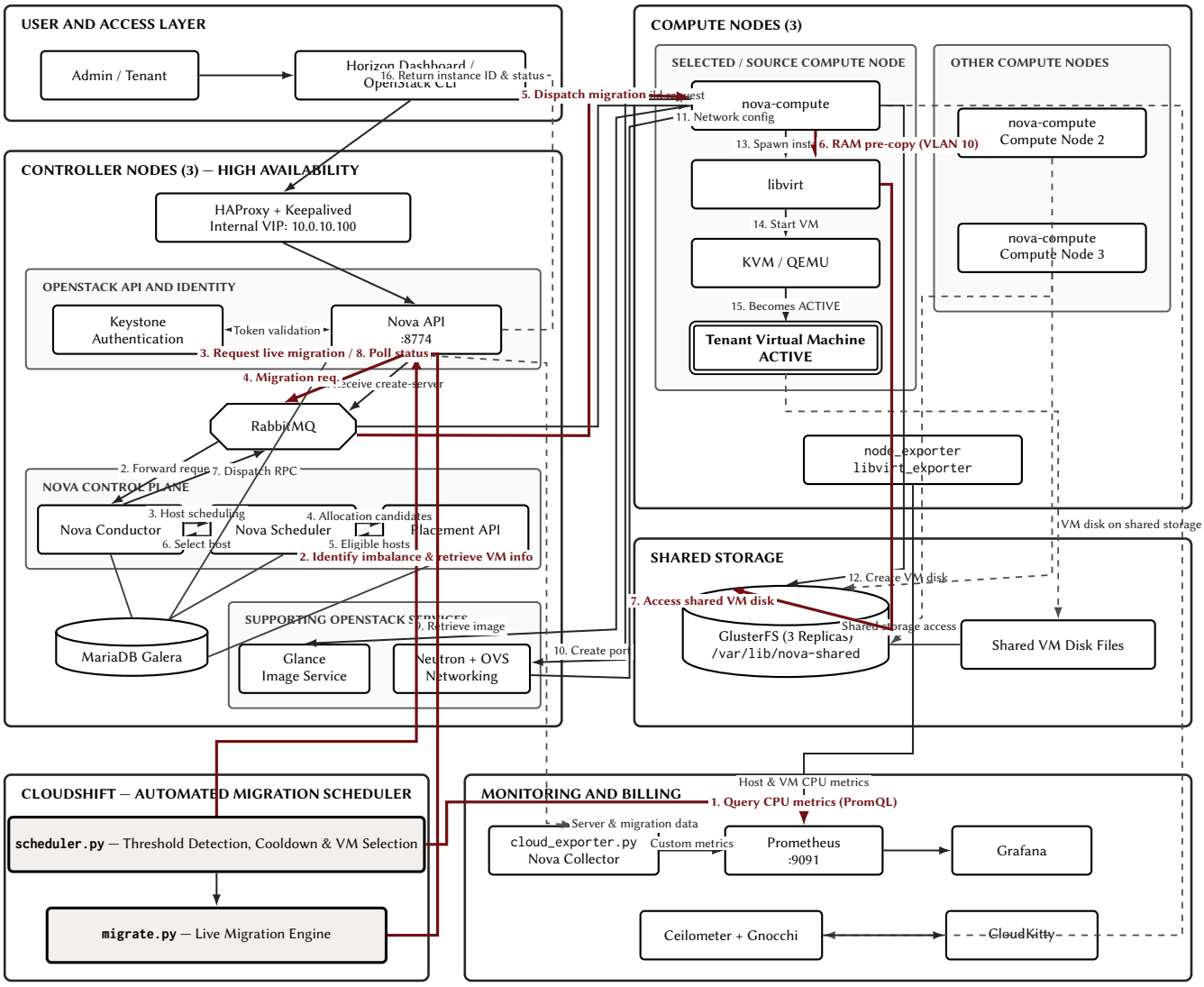

Figure 4. OpenStack logical architecture, 16-step VM provisioning workflow and 8-step CloudShift live migration flow.

be in progress. CloudShift retrieves the active VMs on the busiest host through the Nova API (CloudShift Step 2). It obtains their individual CPU consumption metrics. The `pick_vm()` mechanism evaluates the candidates. It selects a suitable VM whose migration can reduce the imbalance. CloudShift identifies a suitable destination host. It passes the migration request to `migrate.py`. Before migration, the engine validates that the VM is ACTIVE. The source and destination hosts must differ. The destination must be enabled and operational. It then submits a live migration request to the Nova API with `block_migration=False` [1] (CloudShift Step 3). The Nova API forwards the migration request to RabbitMQ (CloudShift Step 4), which dispatches the operation to `nova-compute` (CloudShift Step 5). The source hypervisor transfers the VM memory pages to the destination over Management VLAN 10 through libvirt (CloudShift Step 6). The VM continues running during this. The destination accesses the existing disk files through shared GlusterFS storage (CloudShift Step 7). It does not copy the entire disk. Meanwhile, `migrate.py` polls the Nova API until migration finishes (CloudShift Step 8). A brief interruption may occur during the final switchover.

2.5 Design Considerations

Several design considerations guide the implementation of CloudShift. OpenStack manages VM provisioning. CloudShift handles automated load balancing. It does not replace the Nova Scheduler. Shared storage through GlusterFS lets compute nodes access the same VM disk files. This reduces migration overhead. Migration validation checks the VM

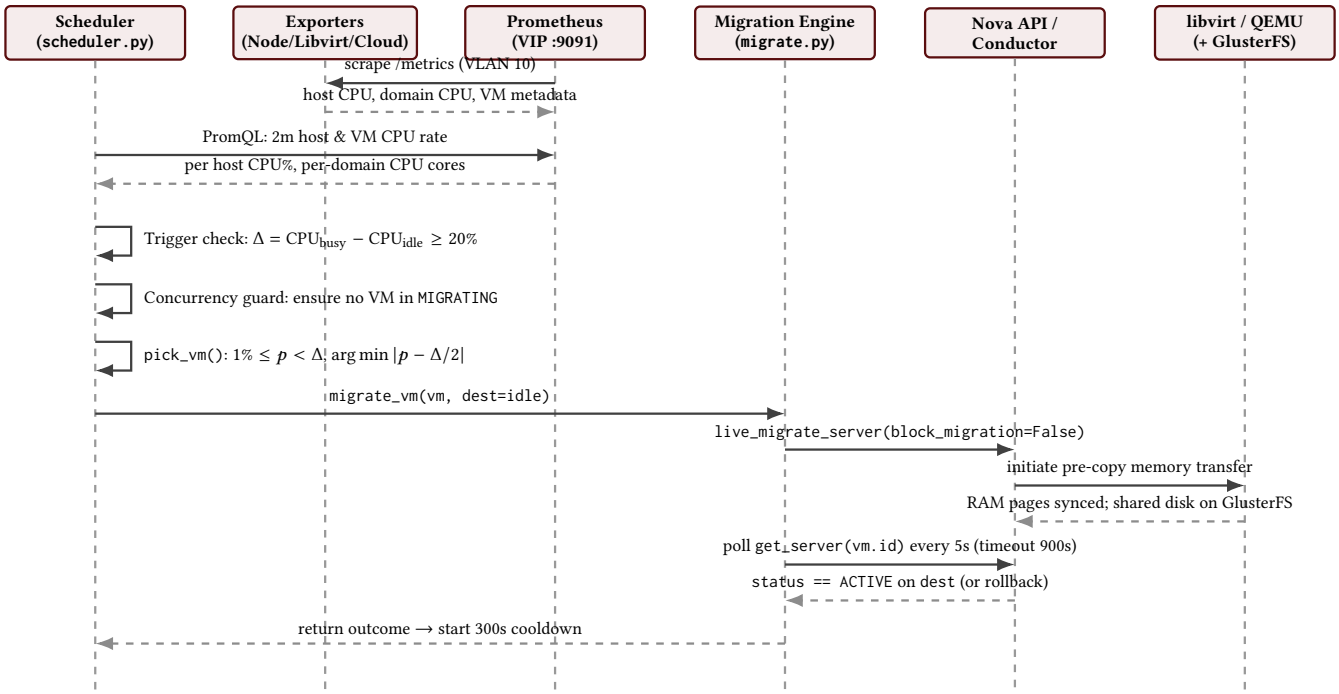

Figure 5. CloudShift live migration workflow sequence.

status and destination availability before initiating migration. The cooldown mechanism prevents repeated migrations within short intervals. Monitoring integration uses Prometheus to collect CPU utilisation metrics. Grafana provides visual monitoring. Migration verification confirms successful migration. It checks both the VM status and its current compute host. Deployment automation through Ansible and Kolla Ansible improves deployment consistency. It reduces manual configuration.

Overall, CloudShift integrates automated infrastructure deployment, OpenStack resource management, shared storage and monitoring with an additional live migration mechanism to improve workload distribution across compute nodes.

3 Hardware and Software Platform

3.1 Hardware

The lab uses three controller servers and three compute servers. A TP-Link TL-SG2210P eight port Gigabit switch connects them. The three compute nodes are heterogeneous in CPU capacity. Cloud03 compute01 provides 16 vCPUs. Cloud03 compute02 provides 8 vCPUs. Cloud03 compute03 provides 4 vCPUs. Each node has 16 to 32 GB RAM. Each has 1 GbE NICs for management and storage traffic. Controllers have dedicated disks for GlusterFS bricks. Compute nodes mount shared storage over the network. Wi-Fi adapters on gateway controllers provide external connectivity. The common Nehalem CPU baseline ensures live migration compatibility across nodes. vCPU overcommitment is set to 3 to 1. Management and Ansible access use the 192.168.0.x physical network.

3.2 Software

Table 2 lists the software stack, grouped by function, with the reason each tool is used.

Ubuntu Server LTS. Provides the stable base operating system for all six nodes, with long term support and cloud images. It supplies package management, SSH access and the runtime for Ansible and the Python services. Ubuntu cloud images are used because they are small, pre-configured for cloud init, and boot quickly on KVM. Its LTS release gives a predictable, well supported base for the whole lab.

KVM/QEMU and libvirt. Provide hardware-accelerated virtualisation and live migration of VMs. libvirt manages domains and cooperates with shared GlusterFS storage so only memory is copied during migration. QEMU emulates the guest hardware while KVM offloads execution to the CPU. Together they give the Nova compute service a reliable, migration-capable hypervisor.

Open vSwitch. Implements VLAN and VXLAN networking for management and tenant traffic, enabling multi tenant isolation and trunked switch ports across the lab. It integrates with Neutron to provide virtual switches on each compute node. OVS was chosen for its maturity and OpenStack support.

Node type	Count	CPU / RAM	Storage / Network
Controller	3	≥8 vCPUs, 16–32 GB	Dedicated disk for /srv/gluster, 1 GbE, Wi-Fi gateway
Compute	3	16 / 8 / 4 vCPUs, 16–32 GB	GlusterFS network mount, 1 GbE
Switch	1	TP-Link TL-SG2210P	8-port Gigabit Ethernet

Table 1. Hardware platform.

OpenStack 2026.1. The IaaS platform supplying Nova (compute), Neutron (network), Glance (images) and Keystone (identity), plus Ceilometer, Gnocchi and CloudKitty for telemetry and billing. OpenStack was chosen because it is open source, modular and exposes the APIs CloudShift needs for scheduling and migration. It also provides the multi tenant model and Horizon dashboard used throughout the lab.

Kolla-Ansible. Containerises and deploys OpenStack across the three controllers with high availability, validating the inventory and merging it with Kolla's stock inventory. It standardises the deployment and makes it reproducible. Kolla-Ansible was chosen to avoid manual service installation.

Docker. Container runtime that runs every OpenStack control-plane service as a container. It isolates services and simplifies upgrades and restarts. Docker is the runtime required by Kolla.

GlusterFS. Provides three replica shared storage for VM disks and images, enabling live migration without block migration. It presents a single shared path to all compute nodes. GlusterFS was chosen for its simple replica setup on commodity disks.

MariaDB Galera. Clustered database holding OpenStack state with quorum and failover across the three controllers. Galera gives synchronous replication and automatic failover. It backs every OpenStack service that needs persistent state.

RabbitMQ. Messaging bus coordinating asynchronous requests between OpenStack services. It decouples producers and consumers and supports high availability. RabbitMQ is the standard message broker for OpenStack.

Valkey. Distributed coordination and locking for telemetry services. It replaces Redis for coordination tasks. Valkey is used by Ceilometer and related services.

HAProxy and Keepalived. Provide VIP failover and load balancing for the high availability control plane. HAProxy distributes requests across controllers, while Keepalived moves the VIP on failure. Together they give a single entry point for clients.

Prometheus. Time-series database and scraper; writes incoming samples to a write ahead log (WAL) on secondary memory before compaction, giving durable, queryable metrics with low overhead. Prometheus is TSDB-based and persists samples to disk via the WAL so data survives restarts. It scrapes node, libvirt and cloud exporters and evaluates PromQL queries used by the scheduler.

Grafana. Visualisation layer for cluster, VM, migration, project and billing dashboards, provisioned from JSON. It queries Prometheus and renders time series panels. Grafana was chosen for its flexible dashboards and provisioning support.

Ceilometer and Gnocchi. Telemetry collection and time series storage feeding usage data to CloudKitty. Ceilometer collects metrics while Gnocchi stores them. They provide the usage records behind billing.

CloudKitty. Usage rating and billing engine producing per tenant invoices. It applies rating rules to Ceilometer/Gnocchi data. CloudKitty was chosen for OpenStack-native billing.

Ansible. Automates provisioning and deployment of all nodes and services from a single inventory. Playbooks and configuration files are written in YAML rather than JSON or other formats, because YAML is more easily human-readable, supports comments, and is declarative, mapping cleanly to nested data structures. Ansible was chosen for agentless SSH based automation.

Python 3. Implements the scheduler and custom exporter. Python was selected because its OpenStack SDK and Prometheus client libraries provide ready-made API bindings, its readable syntax speeds up development of the control logic, and it integrates easily with YAML configuration. Python also has strong support for HTTP APIs and data processing, and is the language of the OpenStack client libraries, so no additional language runtime is needed.

Git. Version control for playbooks, templates and configuration. It tracks changes and enables collaboration. Git keeps the infrastructure code reproducible.

QCOW2 images. VM images are stored in QCOW2 format only, because it supports copy on write, snapshots and sparse allocation while remaining compatible with Glance and QEMU. QCOW2 keeps image storage efficient on shared GlusterFS. Only QCOW2 is used to keep the image pipeline simple and consistent.

4 Sample Source Code

The following listings show the essential implementation snippets.

```
def migrate_vm(conn, vm, dest, dry_run=False):
    vm = conn.compute.get_server(vm.id)
    src = vm.compute_host
```

Group	Tool	Why used
Operating System	Ubuntu Server LTS	Host operating system for all nodes and cloud images.
Virtualisation	KVM/QEMU, libvirt	Virtualisation and live migration of VMs with shared storage.
	Open vSwitch	VLAN and VXLAN networking for management and tenant traffic.
Cloud	OpenStack 2026.1	IaaS platform for compute, network, images, and identity.
Deployment	Kolla-Ansible	Containerised deployment and high availability across controllers.
	Docker	Container runtime for OpenStack services.
Storage	GlusterFS	Three-replica shared storage for VM disks and images.
Data Services	MariaDB Galera	Clustered database for OpenStack state.
	RabbitMQ	Messaging between services.
	Valkey	Coordination for telemetry services.
	HAProxy, Keepalived	VIP failover for high availability.
Observability	Prometheus	Metrics collection and time series storage.
	Grafana	Dashboards for cluster and billing.
Telemetry and Billing	Ceilometer, Gnocchi	Telemetry collection and storage.
	CloudKitty	Usage rating and billing.
Automation and Tools	Ansible	Automation of provisioning and deployment.
	Python 3	Scheduler and exporter implementation.
	Git	Version control for configuration.

Table 2. Software platform and purpose.

```
    if vm.status != "ACTIVE" or src == dest:
        return False
    if dest not in compute_hosts(conn):
        return False
    if dry_run:
        return True
    conn.compute.live_migrate_server(vm, host=dest, block_migration=
False)
    start = time.time()
    time.sleep(3)
    while time.time() - start < TIMEOUT:
        now = conn.compute.get_server(vm.id)
        if now.status == "ERROR":
            return False
        if now.status == "ACTIVE" and now.compute_host == dest:
            return True
        if now.status == "ACTIVE" and now.compute_host == src and time.
time() - start > 30:
            return False
        time.sleep(5)
    return False
```

Listing 1. `live migration/migrate.py`: live migration and completion wait.

Listing 1 waits for the completion of one live migration being performed. First, the VM is re-fetched. It checks if it is ACTIVE. It checks that it is not on the destination. It checks that the destination is an enabled compute host. During dry-run mode, it returns without migrating. Otherwise it calls Nova live_migrate_server with shared storage. The server status is polled until it becomes ACTIVE on the destination, detects a rollback, or times out.

```
def pick_vm(vms, diff):
    candidates = [(s, pct) for s, pct in vms if 1 <= pct < diff]
    return min(candidates, key=lambda c: abs(c[1] - diff / 2))[0] if
candidates else None
```

Listing 2. `live migration/scheduler.py`: victim selection.

Listing 2 selects the victim VM from the busiest host. The candidates with CPU share of at least 1 percent but less than the imbalance are kept. It picks the VM whose share is closest to half the imbalance so both hosts end up balanced. VMs below 1 percent are pointless and hence skipped. VMs larger than the imbalance are skipped to avoid flipping the skew.

```
def check(conn):
    usable = set(migrate.compute_hosts(conn))
    cpu = {h: v for h, v in per_host(
        '100 * (1 - avg by (instance) (rate(node_cpu_seconds_total{mode
="idle"}[2m])))').items() if h in usable}
    if len(cpu) < 2:
        return False
    busy, idle = max(cpu, key=cpu.get), min(cpu, key=cpu.get)
    diff = cpu[busy] - cpu[idle]
    if diff < cfg["threshold"]:
        return False
    by_host = migrate.vms_by_host(conn)
    if any(s.status == "MIGRATING" for vms in by_host.values() for s in
 vms):
        return False
    cores = per_host('count by (instance) (node_cpu_seconds_total{mode
="idle"})')[busy]
    used = {s["metric"]["domain"]: float(s["value"][1]) for s in
prometheus(
        'sum by (domain) (rate(
libvirt_domain_info_cpu_time_seconds_total[2m]))')}
    vms = [(s, 100 * used.get(s.instance_name, 0) / cores) for s in
by_host[busy] if s.status == "ACTIVE"]
    vm = pick_vm(vms, diff)
    if vm is None:
        return False
    if cfg["dry_run"]:
        return False
    return migrate.migrate_vm(conn, vm, idle)
```

Listing 3. `live migration/scheduler.py`: one scheduler pass.

Listing 3 carries out one scheduler pass. From Prometheus, it collects the per host CPU. It finds the busiest and idlest hosts. If their difference goes above the configured threshold, it proceeds. It also guards against concurrent migrations. It builds per VM CPU shares. It assigns victim selection to pick_vm. It will either log a dry-run decision or trigger migrate_vm.

```yaml
prometheus_url: http://10.0.10.100:9091
hosts:
  cloud03-compute01: 10.0.10.21
  cloud03-compute02: 10.0.10.22
  cloud03-compute03: 10.0.10.23
threshold: 20
interval: 60
cooldown: 300
dry_run: true
```

Listing 4. `live migration/config.yaml`: decision parameters.

Listing 4 configures the scheduler. It points at Prometheus. It maps each compute host to its IP address. It sets the balance parameters. These are the threshold, the polling interval, the reset after migration cooldown, and the dry-run option. These values are loaded on boot and on each pass by the scheduler.

```python
def collect_nova(conn):
    m = Metrics()
    projects = {p.id: p.name for p in conn.identity.projects()}
    per_project = {}
    for s in conn.compute.servers(details=True, all_projects=True):
        project = projects.get(s.project_id, s.project_id)
        flavor = s.flavor
        vcpus, ram = int(flavor.vcpus or 0), int(flavor.ram or 0)
        base = {"domain": s.instance_name or "", "name": s.name,
                "project": project, "host": s.compute_host or ""}
        m.add("cloud_vm_info", "gauge", "One series per VM",
              dict(base, id=s.id, status=s.status,
                   flavor=flavor.original_name or flavor.name or ""),
1)
        m.add("cloud_vm_vcpus", "gauge", "vCPUs of the VM's flavor",
base, vcpus)
        m.add("cloud_vm_ram_mb", "gauge", "RAM (MB) of the VM's flavor"
, base, ram)
        p = per_project.setdefault(project, {"instances": 0, "vcpus":
0, "ram": 0})
        p["instances"] += 1
        if s.status == "ACTIVE":
            p["vcpus"] += vcpus
            p["ram"] += ram
    for project, p in per_project.items():
        m.add("cloud_project_instances", "gauge", "VMs per project (any
 state)",
              {"project": project}, p["instances"])
        m.add("cloud_project_vcpus", "gauge", "vCPUs of ACTIVE VMs per
project",
              {"project": project}, p["vcpus"])
        m.add("cloud_project_ram_mb", "gauge", "RAM (MB) of ACTIVE VMs
per project",
              {"project": project}, p["ram"])
    counts = {}
    for mig in conn.compute.migrations():
        key = (mig.migration_type, mig.status, mig.source_compute, mig.
dest_compute)
        counts[key] = counts.get(key, 0) + 1
    for (kind, status, src, dst), n in counts.items():
        m.add("cloud_migrations_total", "counter",
              "Nova migrations by type/status/source/destination",
              {"type": kind, "status": status, "source": src, "dest":
dst}, n)
    return m.render()
```

Listing 5. `monitoring/cloud_exporter.py`: Nova collector.

Listing 5 gathers from Nova per VM, per project, and migration metrics. It enumerates for all servers the vCPUs, RAM, flavor, status and host. It sums for each project the per instance totals for running instances. It counts the migrations by status, type, source and destination for the cloud_migrations_total counter.

```yaml
nova_instance_datadir_volume: "/var/lib/nova"
nova_compute_extra_volumes:
  - "/var/lib/nova-shared/instances:/var/lib/nova/instances:shared"
nova_extra_volumes:
  - "/var/lib/nova-shared/instances:/var/lib/nova/instances:shared"
glance_file_datadir_volume: "/var/lib/glance-shared"
gnocchi_metric_datadir_volume: "/var/lib/gnocchi-shared"
enable_cinder: "no"
```

Listing 6. `kolla/globals.yml`: shared-storage mounts.

Listing 6 adds the shared GlusterFS storage to the OpenStack services. The instances shared directory provides VMs a shared location. They can live migrate between the compute nodes without block migration. Glance and Gnocchi share their data directories. Cinder is disabled due to block storage as it is provided by GlusterFS.

```
[libvirt]
cpu_mode = custom
cpu_models = Nehalem
```

Listing 7. `kolla/config/nova/nova-compute.conf`: guest CPU baseline.

Listing 7 sets the guest CPU baseline to Nehalem. A common baseline is required across all the compute nodes for live migration. The source CPU features must be supported by the destination. After editing it, Nova is reconfigured to apply the changes.

```yaml
# Single source of truth for the node list. Edit the lines marked
CHANGE.
all:
  vars:
    ansible_user: ansible
    ansible_become: true
  children:
    control:
      hosts:
        cloud03-controller01: {ansible_host: 192.168.0.11, host_id: 11,
 nic: eno2}
        cloud03-controller02: {ansible_host: 192.168.0.12, host_id: 12,
 nic: eno2}
        cloud03-controller03: {ansible_host: 192.168.0.13, host_id: 13,
 nic: eno2}
    compute:
      hosts:
        # cloud03-controlcompute:                       # controller 03
is also a compute node
        cloud03-compute01: {ansible_host: 192.168.0.21, host_id: 21,
nic: enx00e04c645b2f}
```

```
        cloud03-compute02: {ansible_host: 192.168.0.22, host_id: 22, nic: enx000ec6b1d736}
        cloud03-compute03: {ansible_host: 192.168.0.23, host_id: 23, nic: eno1}
        #cloud03-compute04: {ansible_host: 192.168.0.24, host_id: 24, nic: enp3s0}
        #cloud03-compute05: {ansible_host: 192.168.0.25, host_id: 25, nic: enp3s0}
    gateway:
      hosts:
        cloud03-controller01: {wifi_if: wlo1, gw_priority: 150}   # CHANGE wifi_if (ip -br link)
        cloud03-controller02: {wifi_if: wlo1, gw_priority: 120}   # no Wi-Fi adapter on a controller?
        cloud03-controller03: {wifi_if: wlo1, gw_priority: 100} # -> delete its line (see docs/HA.md)
```

Listing 8. `ansible/inventory.yml`

Listing 8 is the definitive source of the node list. Every IP address is driven by host_id. The hostname and the inventory name must be equal. Compute nodes and controller nodes are separately categorized. Only the controllers with Wi-Fi are part of the gateway group.

```
---
lab_domain: cloud03.lab

vlan: {mgmt: 10, storage: 20, tunnel: 30, api: 40, egress: 50, provider: 60, dns: 70}

# First three octets of each /24. Host address = <prefix>.<host_id>   (see inventory.yml)
net:
  mgmt: "10.0.10"
  storage: "10.0.20"
  tunnel: "10.11.10"
  api: "10.14.40"          # controllers only. (Was 192.168.1: that is your home Wi-Fi range.)
  egress: "10.12.0"
  dns: "10.0.70"

internal_vip: 10.0.10.100  # Kolla/HAProxy VIP on VLAN 10
external_vip: 10.14.40.100 # Kolla/HAProxy VIP on VLAN 40 (Horizon, Grafana, APIs)

# ---- Internet gateway: floating addresses (VRRP) held by ONE healthy controller that has Wi-Fi ----
gateway_egress_ip: 10.12.0.1        # default route of every node without its own Wi-Fi
provider_cidr: 172.24.4.0/24        # Neutron provider / floating-IP network (VLAN 60)
provider_gateway: 172.24.4.1        # gateway of that network, put on br-ex of the VRRP master
gateway_vrid: 61                    # Kolla own keepalived uses 51
gateway_vrrp_password: cloud3gw     # exactly 8 characters (VRRP truncates longer ones)
upstream_dns: [1.1.1.1, 8.8.8.8]
physical_net: 192.168.0.0/24        # the untagged VLAN 1 network (Ansible/SSH access)

# DNS servers = every controller, on VLAN 70 (computed from the inventory)
# dns_servers computed from inventory (see all.yml)

# ---- Shared storage: GlusterFS, 3 copies, one on each controller ----
gluster_brick_root: /srv/gluster     # bricks live in <root>/<volume>/brick on each controller
gluster_arbiter: false               # true = third controller keeps only metadata (faster writes, 2 data copies)
gluster_require_dedicated_disk: true  # true = refuse to put bricks on the root filesystem
gluster_volumes:
  nova:    {mount: /var/lib/nova-shared,    clients: compute, virt: true}   # VM disks (live migration)
  glance:  {mount: /var/lib/glance-shared,  clients: control}
    # images
  gnocchi: {mount: /var/lib/gnocchi-shared, clients: control}
    # telemetry data
```

Listing 9. `ansible/group_vars/all.yml`

Listing 9 explains the following shared variables. VLAN IDs and subnets. Internal and external VIPs. The GlusterFS volume layout. The 8 character VRRP password. The inventory host IDs are used to compute the DNS servers.

```
# Managed by Ansible. Adds VLAN interfaces on top of the NIC already defined in your
# existing netplan file (the 192.168.0.x address stays there, untouched).
network:
  version: 2
  ethernets:
    {{ nic }}:
      addresses: [192.168.0.{{ host_id }}/24]
  vlans:
    mgmt:
      id: {{ vlan.mgmt }}
      link: {{ nic }}
      addresses: [{{ net.mgmt }}.{{ host_id }}/24]
    storage:
      id: {{ vlan.storage }}
      link: {{ nic }}
      addresses: [{{ net.storage }}.{{ host_id }}/24]
    tunnel:
      id: {{ vlan.tunnel }}
      link: {{ nic }}
      addresses: [{{ net.tunnel }}.{{ host_id }}/24]
{% if inventory_hostname in groups['control'] %}
    api:
      id: {{ vlan.api }}
      link: {{ nic }}
      addresses: [{{ net.api }}.{{ host_id }}/24]
{% endif %}
    egress:
      id: {{ vlan.egress }}
      link: {{ nic }}
      addresses: [{{ net.egress }}.{{ host_id }}/24]
{% if inventory_hostname not in groups['gateway'] %}
      routes:
        - to: default
          via: {{ gateway_egress_ip }}      # the floating gateway address
      nameservers:
        search: [{{ lab_domain }}]
        addresses: [{{ (dns_servers + [upstream_dns[0]]) | join(', ') }}]
{% endif %}
    provider:                       # Neutron external interface: must be UP and must NOT have an IP
      id: {{ vlan.provider }}
      link: {{ nic }}
      link-local: []
    dns:
      id: {{ vlan.dns }}
      link: {{ nic }}
      addresses: [{{ net.dns }}.{{ host_id }}/24]
```

Listing 10. `ansible/templates/netplan.yaml.j2`

Listing 10 is the Jinja2 template that is attached on top of the existing NIC. It does not remove the 192.168.0.x address or its default route. The per node addresses are rendered from the host_id. The API VLAN and default route are added conditionally.

```
# Managed by Ansible. Floating Internet gateway for the lab (separate from Kolla's keepalived container).
# The controller with the highest healthy priority holds {{ gateway_egress_ip }} on VLAN 50; its
```

```
# notify script also puts {{ provider_gateway }} on br-ex (the VM floating-IP network gateway).
global_defs {
    router_id {{ inventory_hostname }}
    enable_script_security
    script_user root
}

vrrp_script chk_uplink {
    script "/usr/local/sbin/gw-check.sh"
    interval 5
    timeout 4
    fall 2
    rise 2
    weight -60            # a node whose Wi-Fi has no Internet loses 60 priority and hands the gateway over
}

vrrp_instance LAB_GATEWAY {
    state BACKUP
    interface egress
    virtual_router_id {{ gateway_vrid }}
    priority {{ gw_priority }}
    advert_int 1
    garp_master_delay 1
    garp_master_repeat 5
    garp_lower_prio_delay 10
    garp_lower_prio_repeat 5
    garp_master_refresh 30
    garp_master_refresh_repeat 2
    authentication {
        auth_type PASS
        auth_pass {{ gateway_vrrp_password }}
    }
{% if groups['gateway'] | length > 1 %}
    unicast_src_ip {{ net.egress }}.{{ host_id }}
    unicast_peer {
{% for h in groups['gateway'] if h != inventory_hostname %}
        {{ net.egress }}.{{ hostvars[h].host_id }}
{% endfor %}
    }
{% endif %}
    virtual_ipaddress {
        {{ gateway_egress_ip }}/24 dev egress
    }
    track_script {
        chk_uplink
    }
    notify_master "/usr/local/sbin/gw-notify.sh MASTER"
    notify_backup "/usr/local/sbin/gw-notify.sh BACKUP"
    notify_fault  "/usr/local/sbin/gw-notify.sh FAULT"
}
```

Listing 11. `ansible/templates/keepalived-gateway.conf.j2`

Listing 11 implements the configuring of the floating Internet gateway with VRRP. The controller with the highest priority health-wise holds the egress and the provider VIPs. Priority by 60 is dropped by a failed uplink. This results in the failover of the gateway.

```
# Managed by Ansible - lab DNS, runs on every controller
# Listens on this controller's VLAN 70 address, plus the two floating gateway addresses
# (only the current gateway master actually holds those, bind-dynamic copes with them coming and going).
bind-dynamic
listen-address={{ net.dns }}.{{ host_id }}
listen-address={{ gateway_egress_ip }}
listen-address={{ provider_gateway }}

domain={{ lab_domain }}
local=/{{ lab_domain }}/
domain-needed
bogus-priv
cache-size=1000

# Upstream resolvers
{% if inventory_hostname in groups['gateway'] %}
resolv-file=/run/systemd/resolve/resolv.conf
{% else %}
no-resolv
{% endif %}
{% for s in upstream_dns %}
server={{ s }}
{% endfor %}

# Cluster hosts (management VLAN), the storage-VLAN names of the controllers, and the two API VIPs
{% for h in groups['all'] %}
host-record={{ h }},{{ h }}.{{ lab_domain }},{{ net.mgmt }}.{{ hostvars[h].host_id }}
{% endfor %}
{% for h in groups['control'] %}
host-record={{ h }}-st,{{ net.storage }}.{{ hostvars[h].host_id }}
{% endfor %}
host-record=api.{{ lab_domain }},{{ internal_vip }}
host-record=public.{{ lab_domain }},{{ external_vip }}
```

Listing 12. `ansible/templates/dnsmasq.conf.j2`

Listing 12 shows the lab DNS configuration that runs on every controller. It is responsible for the resolution of cluster hostnames, storage VLAN names, the two API VIPs, and the forwarding of upstream queries from the gateway master.

5 Results and Discussion

5.1 Outcomes

The lab was deployed end to end on a testbed with 3 controller and 3 compute hosts. It was validated against the objectives. All Ansible playbooks pass syntax checks. Templates render for every host. The keepalived and dnsmasq configurations pass their built in tests. The Kolla inventory merge places all services across the three controllers. The platform was observed healthy end to end. The Galera cluster reached size 3 and ready state. All nine Nova services were up. These include compute, scheduler and conductor. They run across the three controllers and three compute nodes. No service was down (Figure 12). Prometheus scraped the endpoints. One transient target was down. Grafana dashboards were provisioned from JSON. They exposed cluster, compute, migration, project and platform views.

5.2 Observations

The cluster ran a nine VM workload across five client projects (Figure 14). Before any balancing, placement was skewed. `cloud03-compute02` held 5 VMs. That is 14 vCPUs and 14.5 GiB RAM. `cloud03-compute01` and `cloud03-compute03` held 2 VMs each. That is 5 vCPUs each. The CPU spread between the busiest and idlest node was 78.1% (Figure 10). The compute nodes are heterogeneous in capacity. Compute01 has 16 vCPUs. Compute02 has 8 vCPUs. Compute03 has 4 vCPUs. Raw VM counts understate the load on the smaller nodes. Cloud03 compute02 and cloud03 compute03 are effectively more saturated than their two to five VM counts suggest. Under the load test workload, cloud03 controller01 was pinned near 98 to 100% CPU. The load average

reached up to 79. The other controllers stayed near 23 to 28% sustained CPU. They peaked at 82 to 84%. This reflects the control plane role split and container footprint (Figure 6, Figure 13). Memory pressure was highest on cloud03 compute02. It used 14.5 of 15.3 GiB. Cloud03 compute03 used 5.5 of 5.6 GiB. Horizon's hypervisor view shows this (Figure 16).

Network segmentation was visible per VLAN. API, DNS, egress and GlusterFS storage traffic were tracked separately. GlusterFS replication peaks were near 9.8 MB/s. A large provisioning spike occurred around 15:30 (Figure 7). The custom exporter stayed fresh. The last value was about 34 s since last success. Constant error counts were 129 for Nova and 1.55k for billing (Figure 13). Per project CPU efficiency varied widely. Client2 project1 reached 61%. Client1 project1 stayed near 35%. Another project transiently exceeded 100% (Figure 14, Figure 15). Horizon confirmed the tenant topology. There were nine active instances across the three hosts. There were 21 networks. One external provider network plus tenant /24s. Cumulative usage accounting was also shown (Figure 17, Figure 18, Figure 19).

5.3 Results

The scheduler completed eight live migrations with zero failures. The peak rate was about four migrations per hour between 01:30 and 03:15 (Figure 21). Placement was redistributed from a 2/5/2 split to 8/4/2. This was across cloud03 compute01, cloud03 compute02 and cloud03 compute03. The cluster grew to 14 active VMs (Figure 21). The CPU spread between the busiest and idlest node moved from 78.1% before to 77.8% after. Table 3 summarises the quantitative outcome.

Metric	Value
Migrations completed	8
Migrations failed	0
Peak migration rate	4.03 / hour
Compute hosts up	3
CPU spread (before)	78.1%
CPU spread (after)	77.8%
Active VMs (final)	14

Table 3. Quantitative results.

5.4 Analysis

The results validate API driven live migration on shared GlusterFS storage with a common Nehalem CPU baseline. Eight migrations completed without failure. The Nova API and dashboards tracked the moves in real time. However, the marginal change in CPU spread (78.1% to 77.8%) indicates that the current victim selection and placement logic reduces imbalance only slightly. Moving a few VMs between hosts that are later re loaded by the workload does not materially change the spread. The scheduler needs more aggressive or lookahead based placement to converge. The distribution also reflects the vCPU overcommit (3:1) and the heterogeneous node sizes (16, 8 and 4 vCPUs). A count based 2/5/2 split overloads the smaller nodes. Victim selection by CPU share alone does not account for per node capacity. This is why the spread barely improved. Observed trade offs remain. GlusterFS write amplification during image uploads and migrations. The CPU baseline limitation imposed by the oldest guest model. CloudKitty billing lag. The lab provides a reproducible environment. Results are limited to a single cluster and educational scope. The scheduler was first validated in dry run mode.

5.5 Future Work

Live migration is not yet robust. Migrations can fail midway when the destination runs low on memory or CPU. Network congestion on the management VLAN can stall memory pre copy and extend downtime. GlusterFS write amplification and I/O contention slow the process. The Nehalem CPU baseline limits guest features on newer hardware. The scheduler considers only CPU. It ignores memory, network and disk pressure. Victim selection by CPU share does not account for per node capacity. The cooldown and single migration lock slow rebalancing on larger clusters.

To improve, we will add multi metric imbalance scoring and capacity aware placement. We will allow concurrent migrations with a configurable limit. We will use predictive models based on workload trends. We will upgrade storage to faster replicated backends. We will raise the CPU baseline once all nodes support a newer model. We will make billing near real time. We will add policy based rules such as time windows and priority tenants.

6 Conclusion

CloudShift demonstrates a scheduler that is workload-aware for live migration of VMs on an OpenStack testbed that is highly available and built entirely with infrastructure-as-code. The multi-node lab setup, with 3 controllers and 3 computes that share a GlusterFS storage, were deployed using Kolla-Ansible, validated end to end and instrumented with a custom exporter, Grafana, and Prometheus, along with per project usage accounting through CloudKitty. This work showed the feasibility of closed-loop rebalancing over Nova and Placement APIs on commodity hard-ware that has shared storage, and provides an environment that is reproducible for future experiments dealing with scheduling, telemetry and cloud management. Future work centers on improvement of victim selection and placement that converges cluster imbalances faster and upon addressing.

7 References

[1] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," in *Proc. 2nd USENIX NSDI*, 2005. [Online]. Available: https://www.usenix.org/legacy/events/nsdi05/tech/full_papers/clark/clark.pdf

[2] A. Beloglazov and R. Buyya, “Optimal Online Deterministic Algorithms and Adaptive Heuristics for Energy and Performance Efficient Dynamic Consolidation of Virtual Machines in Cloud Data Centers,” *Concurrency and Computation: Practice and Experience*, vol. 24, no. 13, pp. 1397–1420, 2012. [Online]. Available: https://doi.org/10.1002/cpe.1867

[3] OpenStack Foundation, “OpenStack Documentation,” 2026. [Online]. Available: https://docs.openstack.org

[4] OpenStack Foundation, “Kolla-Ansible Documentation,” 2026. [Online]. Available: https://docs.openstack.org/kolla-ansible

[5] Red Hat, Inc., “GlusterFS Documentation,” 2026. [Online]. Available: https://docs.gluster.org

[6] Prometheus Authors, “Prometheus Documentation,” 2026. [Online]. Available: https://prometheus.io/docs

[7] Grafana Labs, “Grafana Documentation,” 2026. [Online]. Available: https://grafana.com/docs

Appendix: Screenshots

The following figures show the deployment, monitoring, and migration dashboards and the provisioning workflow described in Section 5.

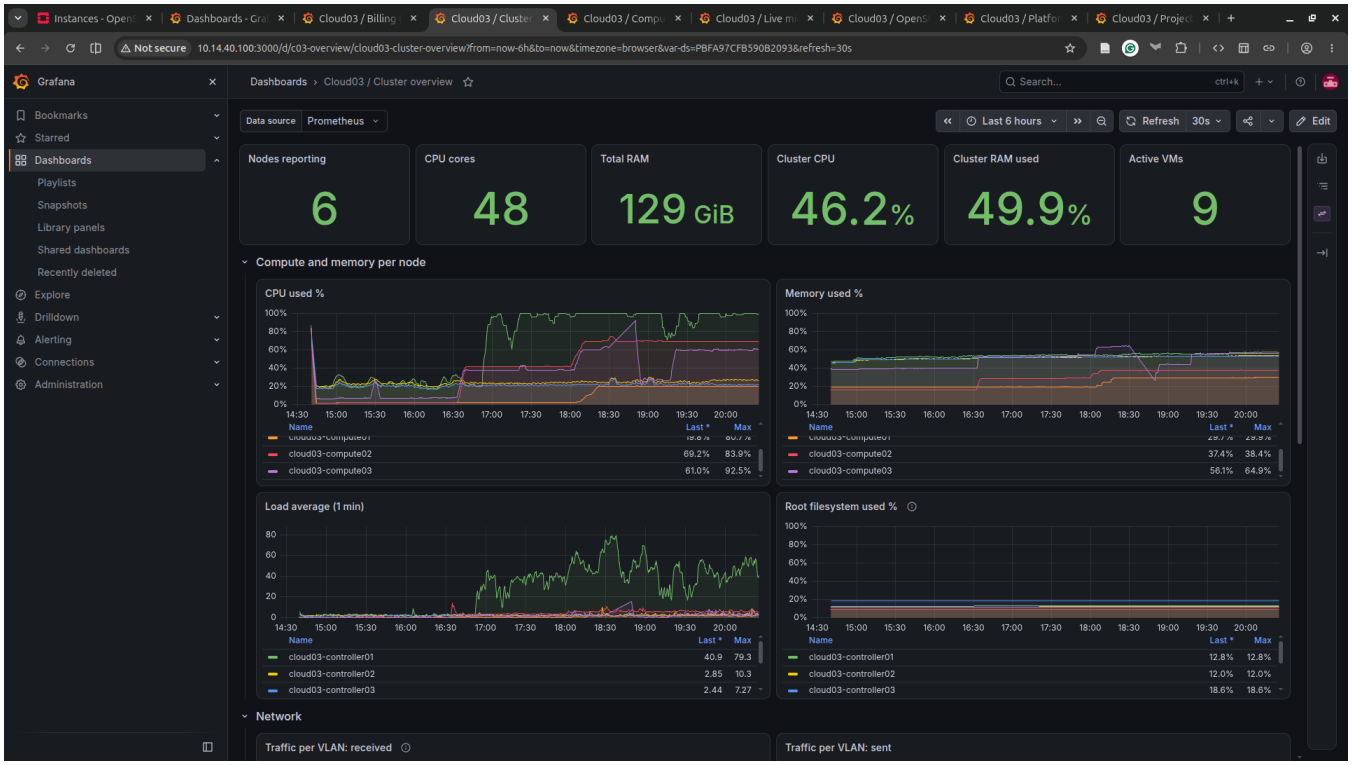

Figure 6. Cluster overview: nodes, CPU, RAM, per node CPU/memory, load, and root filesystem usage.



Figure 7. Cluster overview: per-VLAN traffic, GlusterFS storage throughput, egress, and disk throughput.

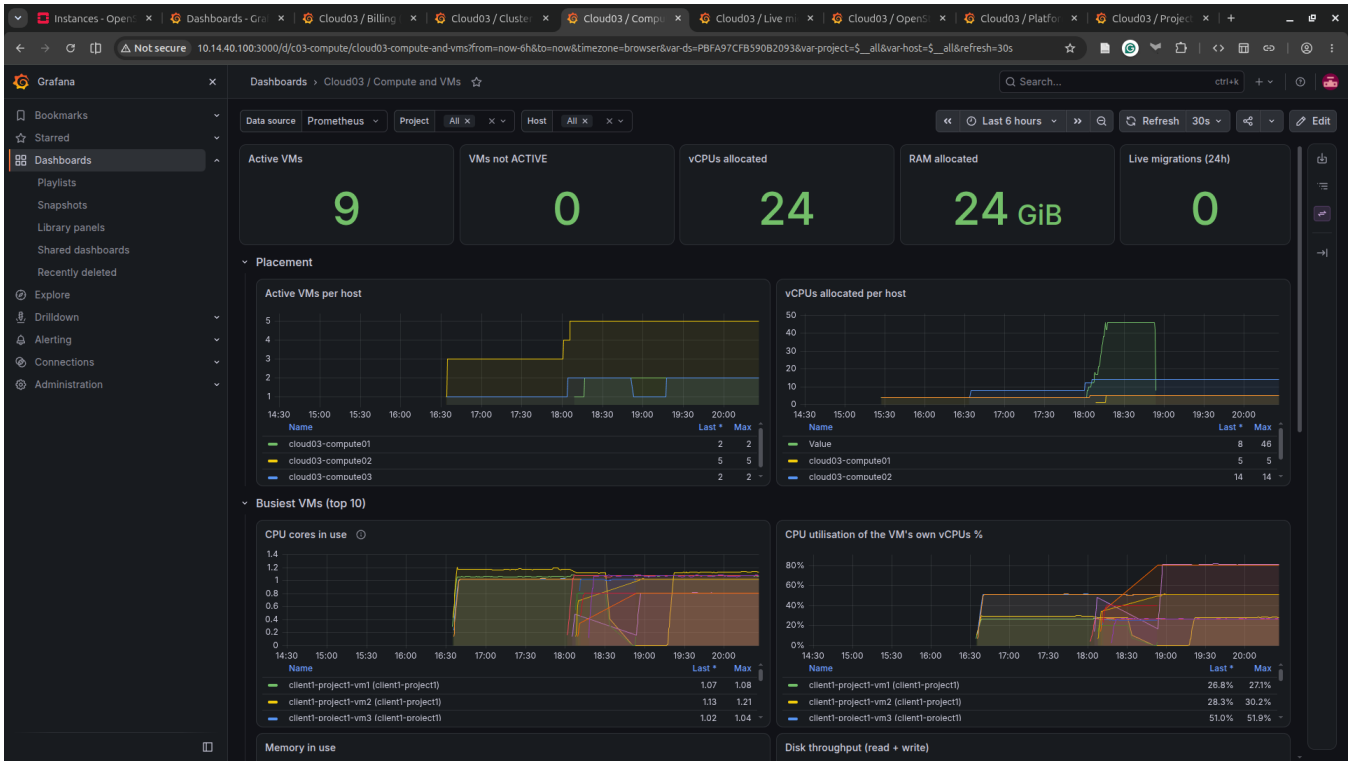

Figure 8. Compute and VMs: active VMs, vCPUs and RAM allocated, placement per host, and busiest VMs.

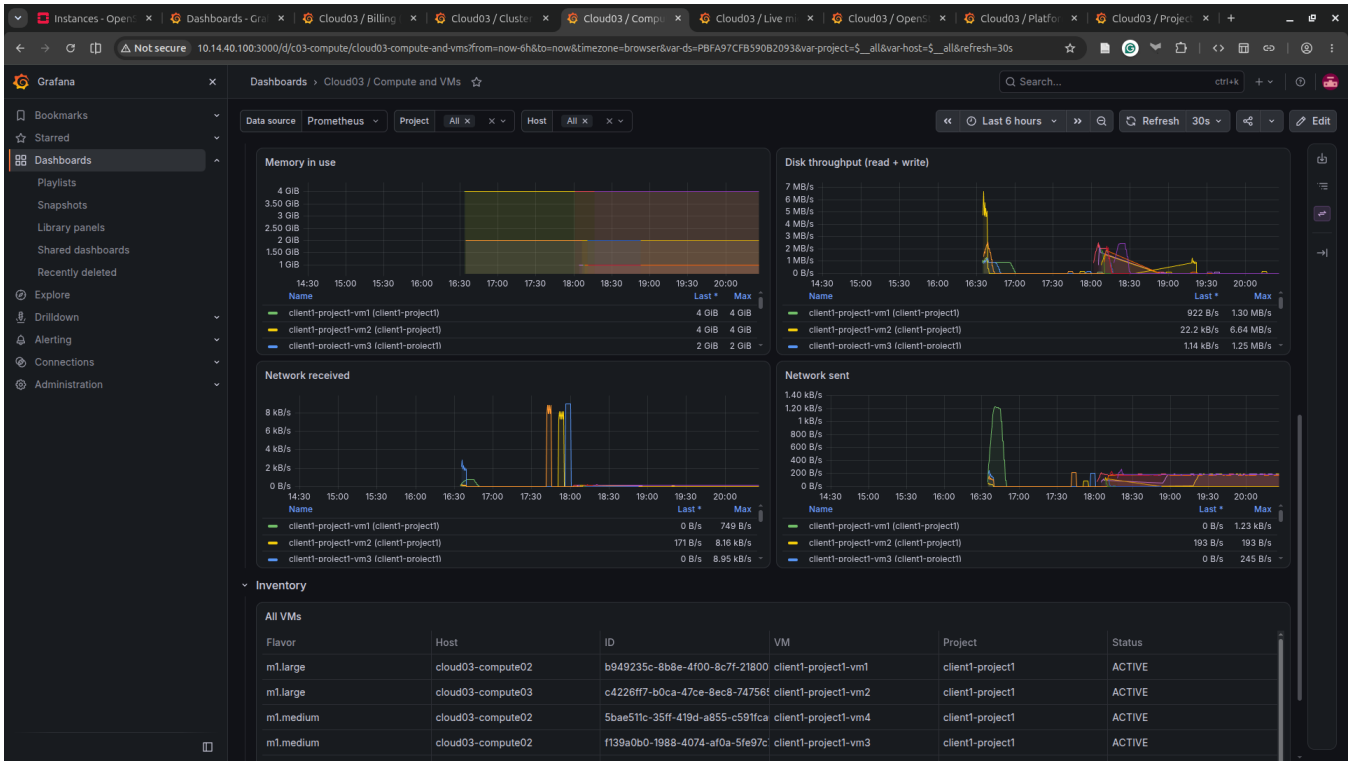

Figure 9. Compute and VMs: per VM memory, disk, network, and the live VM inventory table.

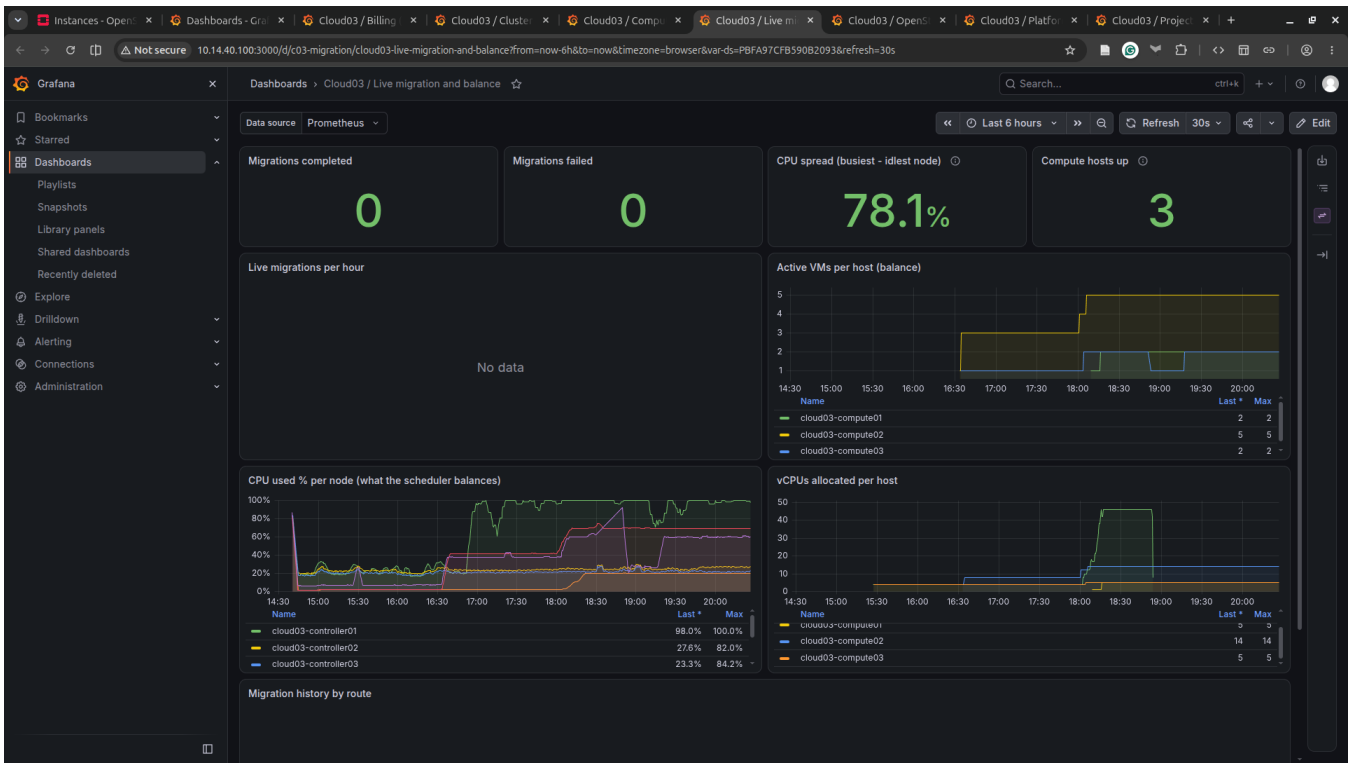

Figure 10. Live migration and balance dashboard before migrations: zero migrations, 78.1% CPU spread.

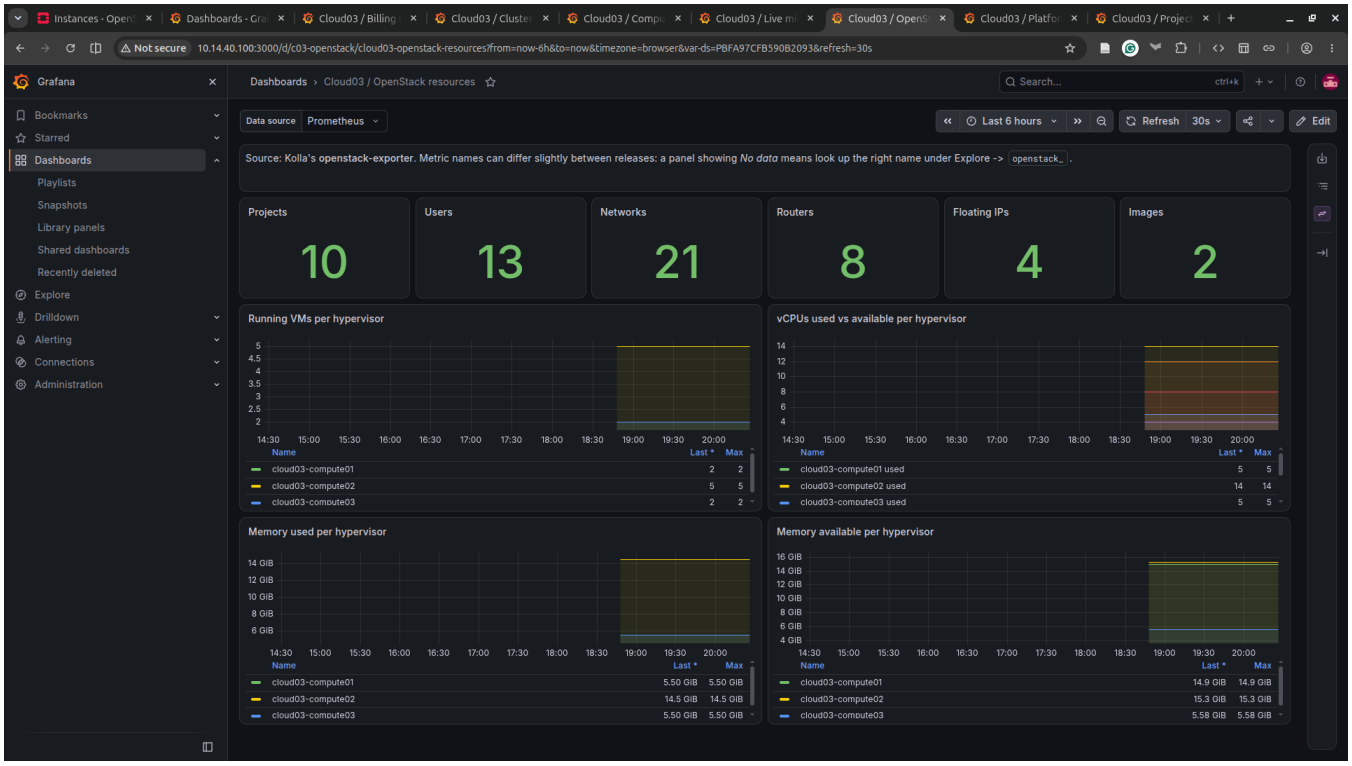

Figure 11. OpenStack resources: projects, users, networks, routers, floating IPs, images, and per-hypervisor usage.

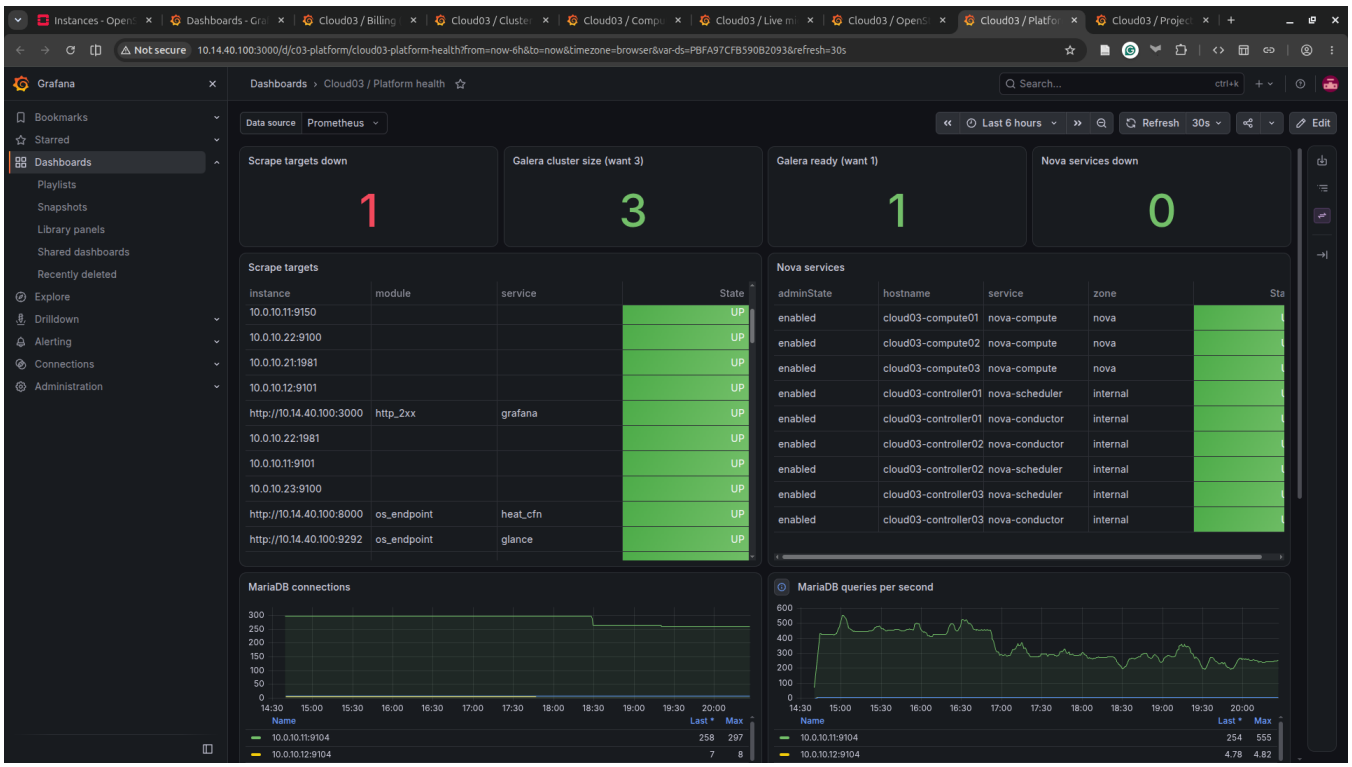

Figure 12. Platform health: scrape targets, Nova services, Galera cluster, and MariaDB connections/queries.

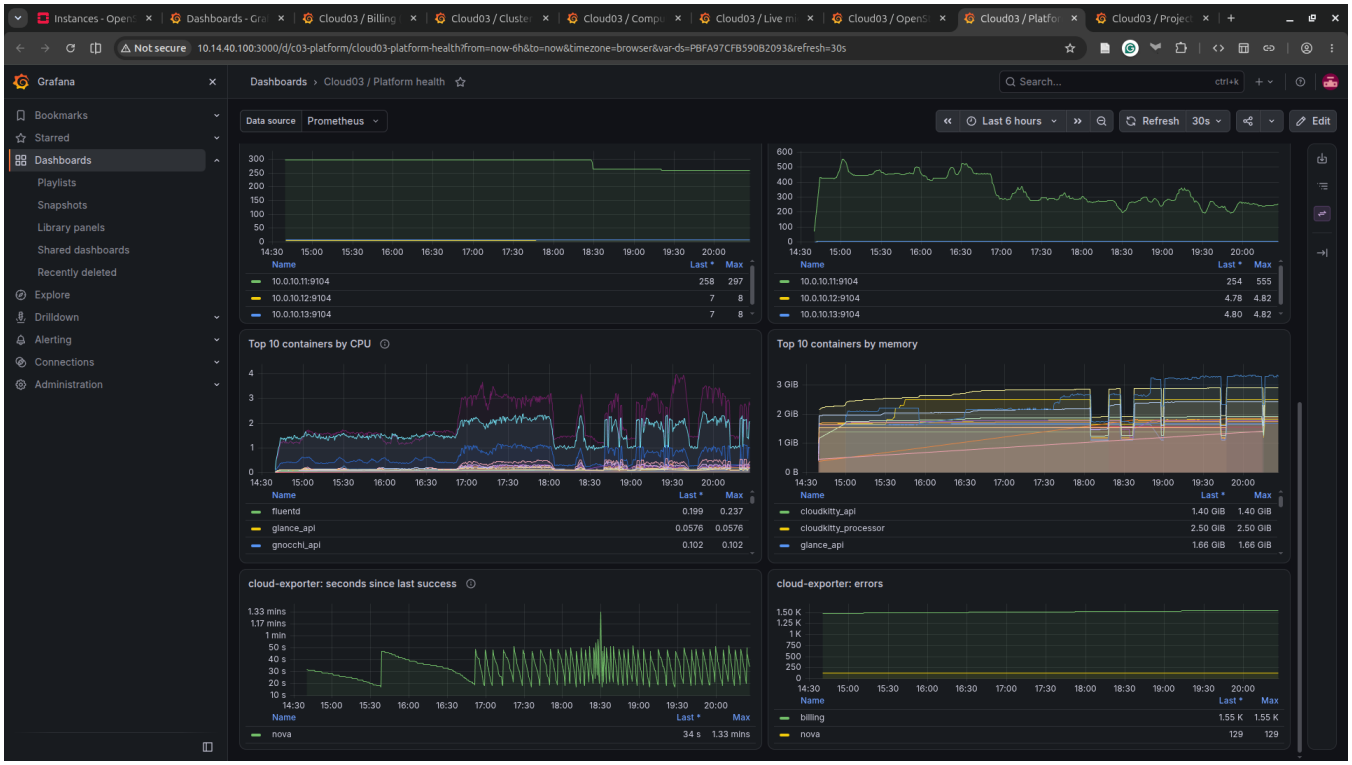

Figure 13. Platform health: top containers by CPU/memory and cloud-exporter scrape freshness and errors.

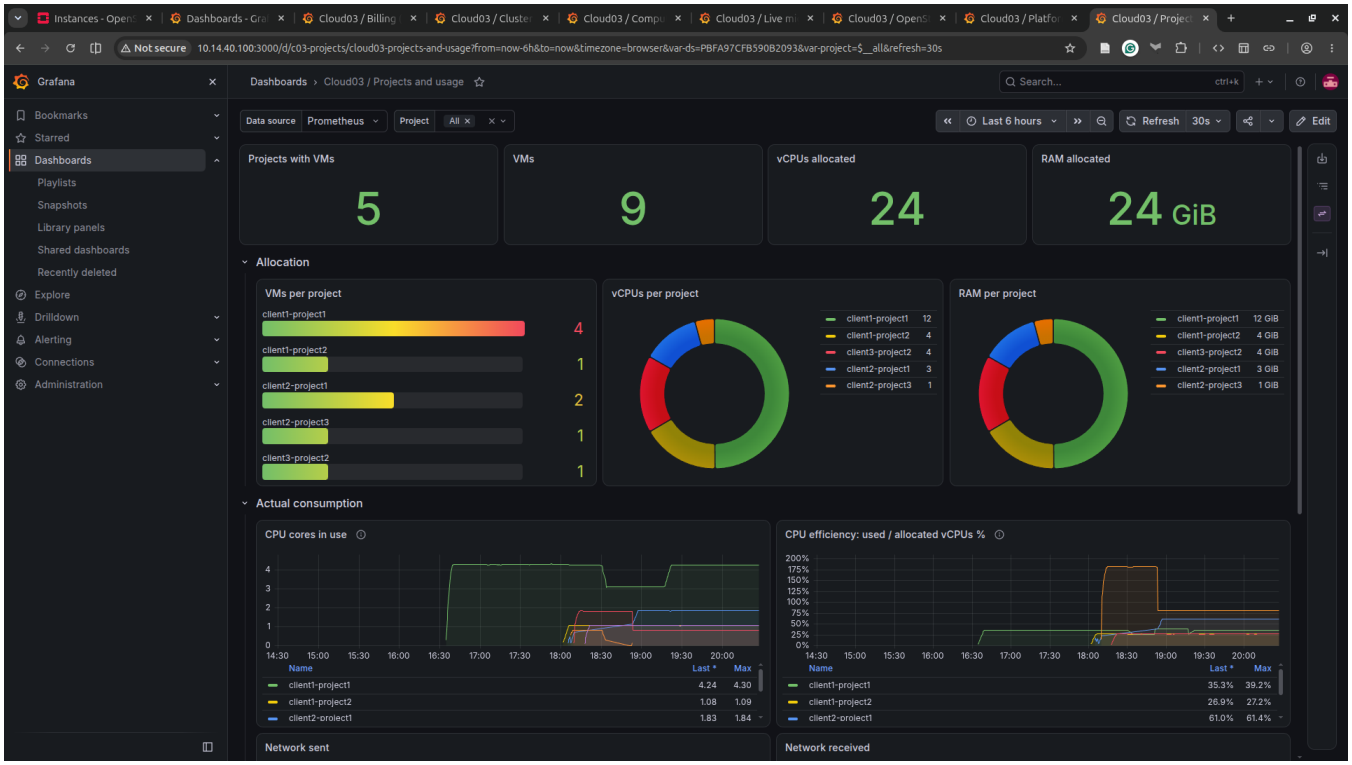

Figure 14. Projects and usage: allocation per project and CPU efficiency.

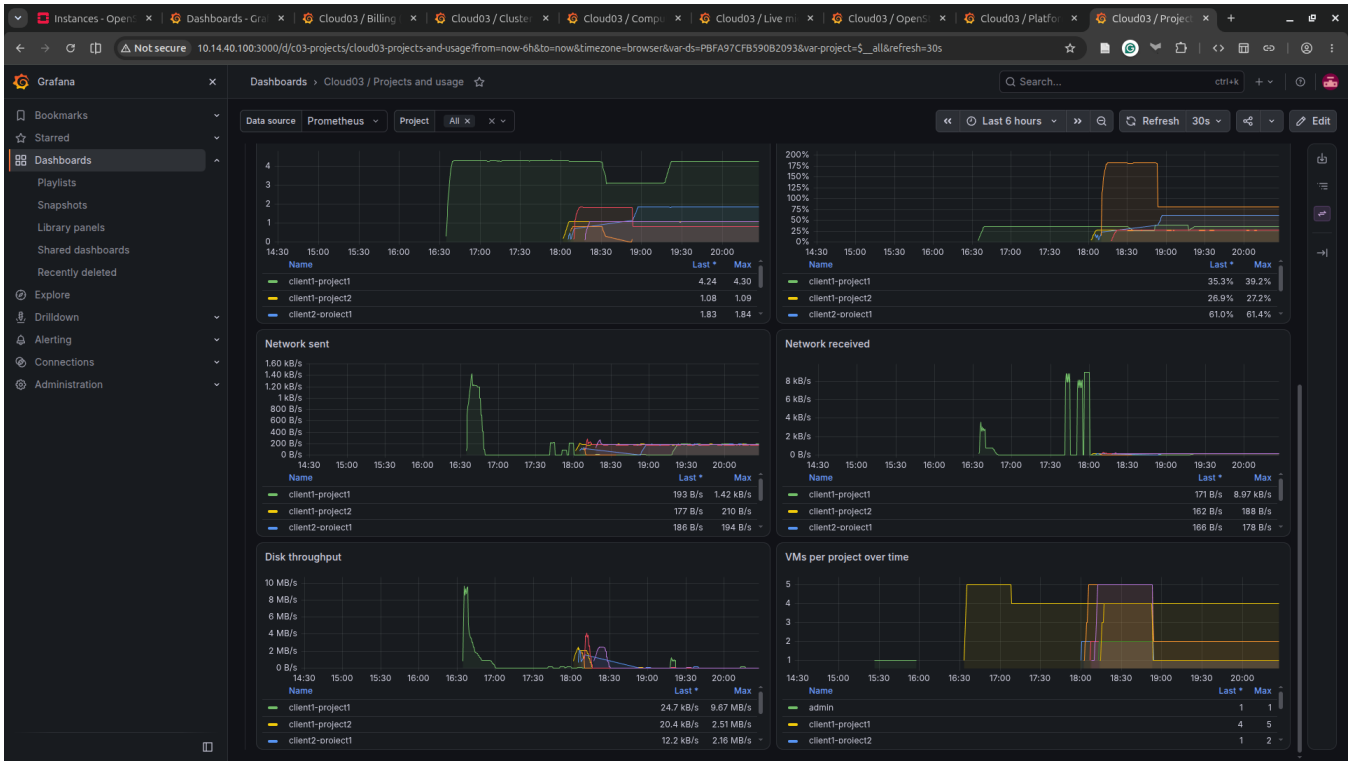

Figure 15. Projects and usage: per project network, disk, and VM count over time.

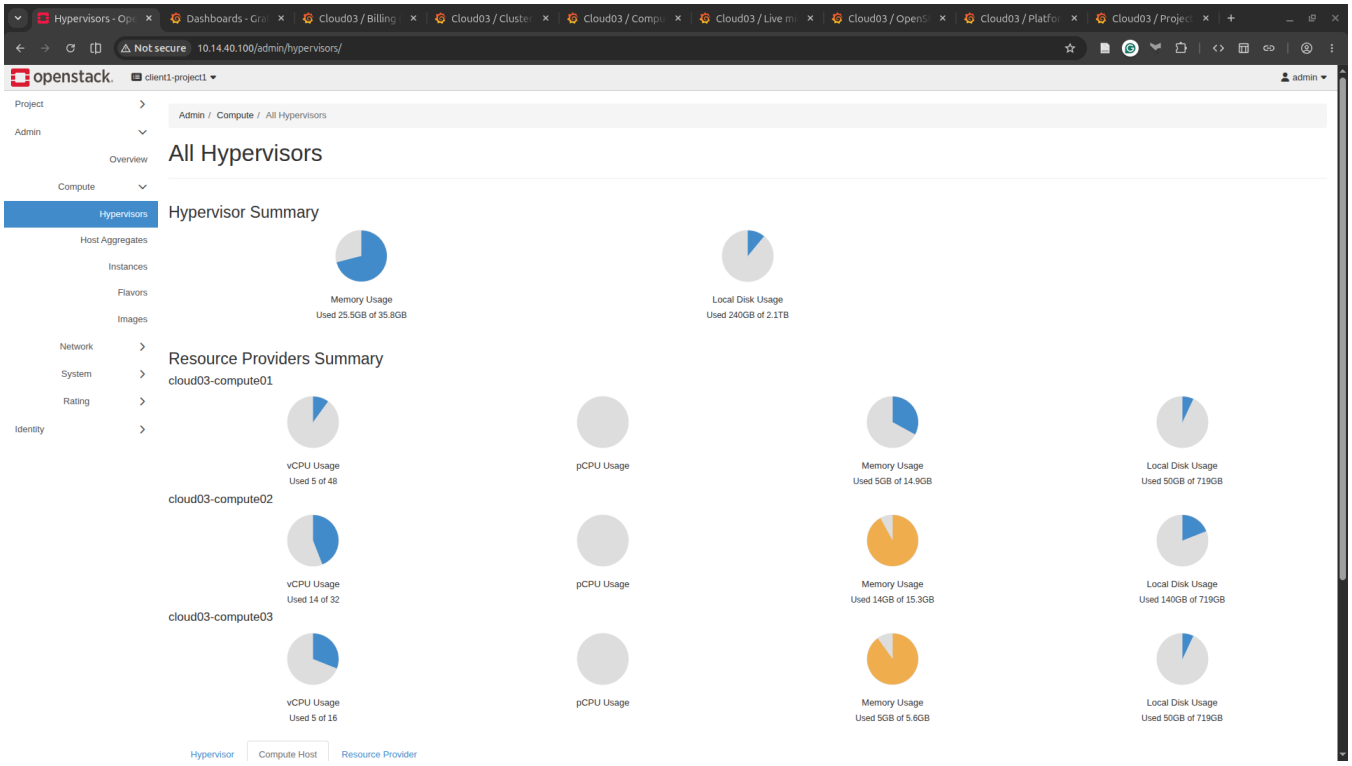

Figure 16. Horizon hypervisor summary and resource-provider view of the three compute nodes.

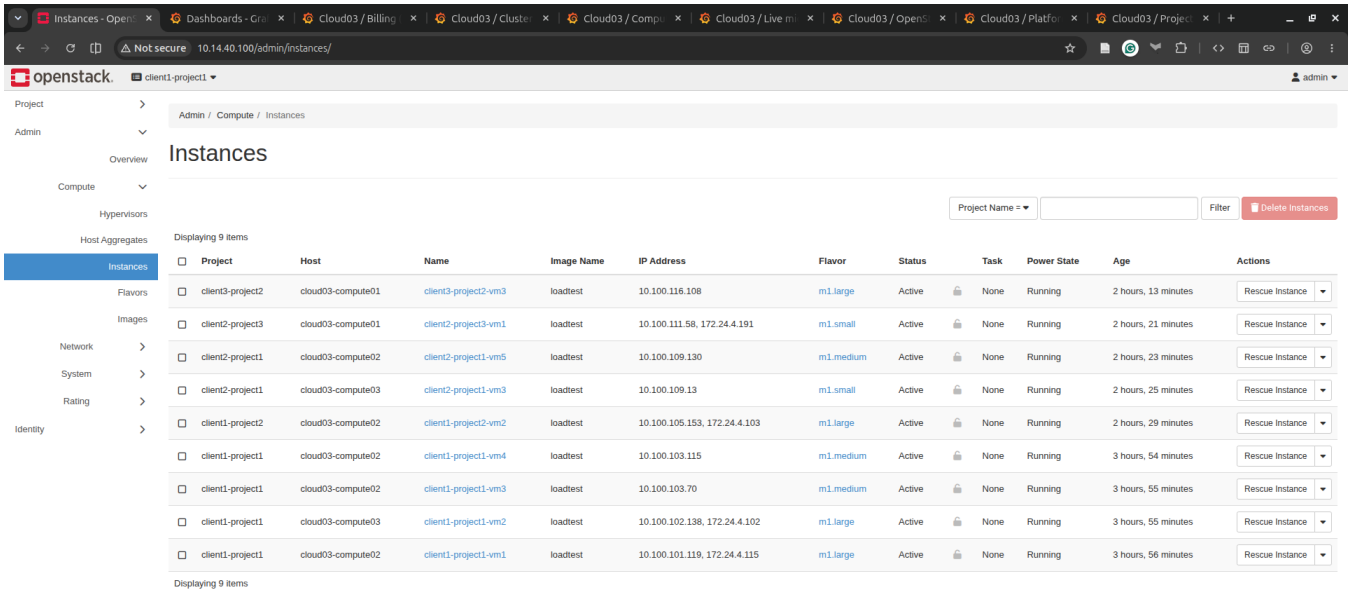

Figure 17. Horizon instances view: nine active tenant VMs across the three compute hosts.

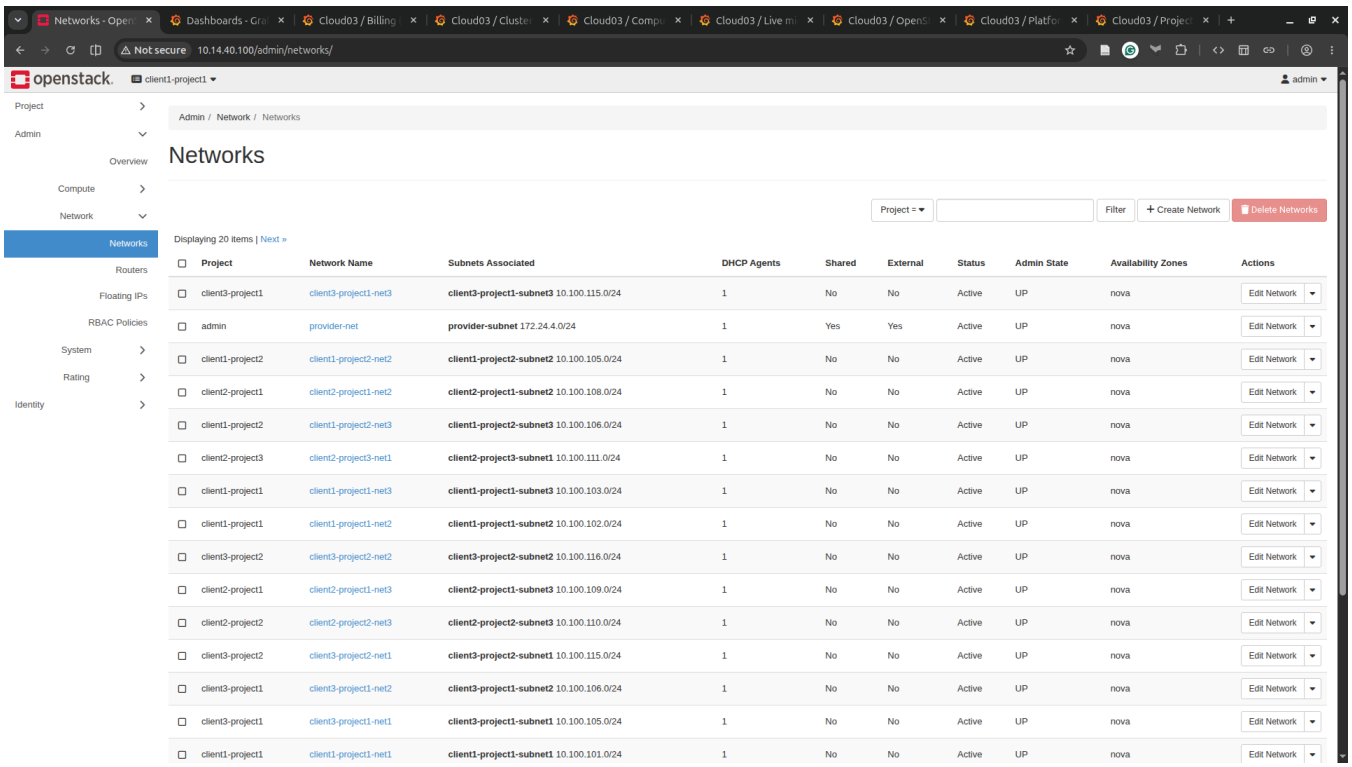

Figure 18. Horizon networks view: external provider network and isolated tenant networks.

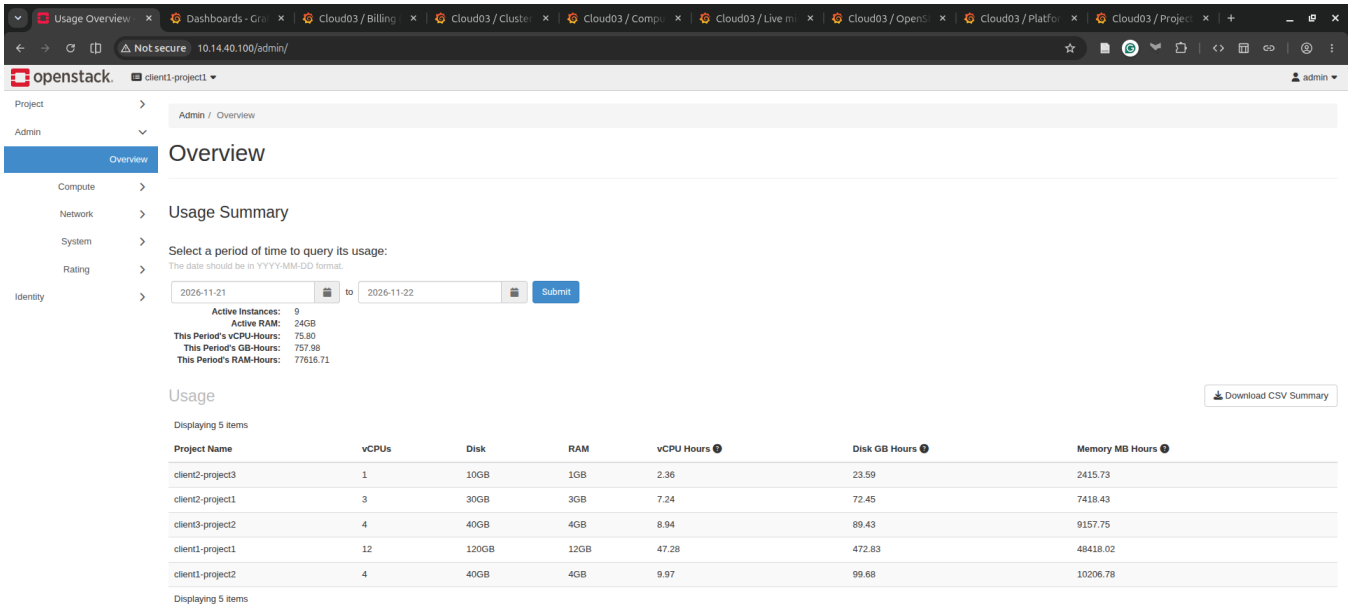

Figure 19. Horizon admin overview: period usage summary and per project vCPU/RAM/disk hours.

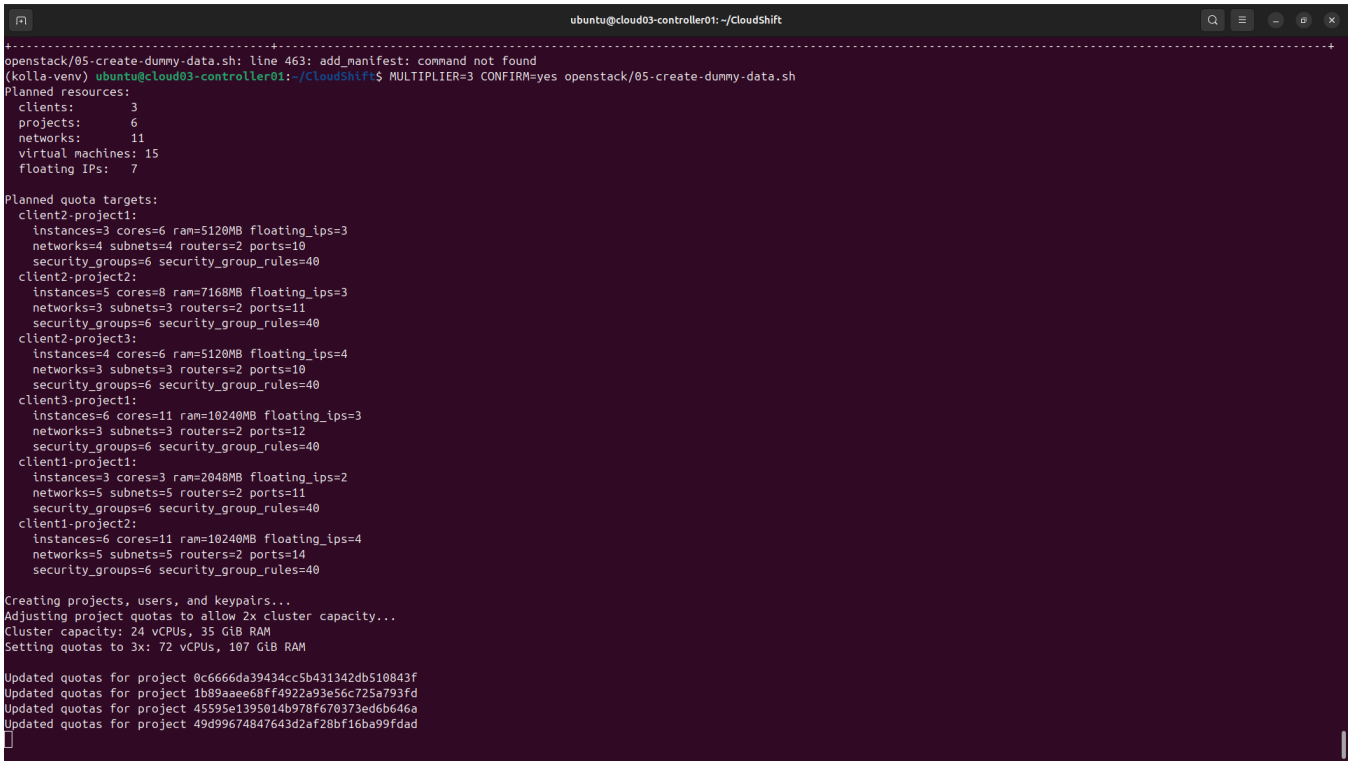

Figure 20. Provisioning workflow: dummy-data script scaling quotas to 3x cluster capacity.

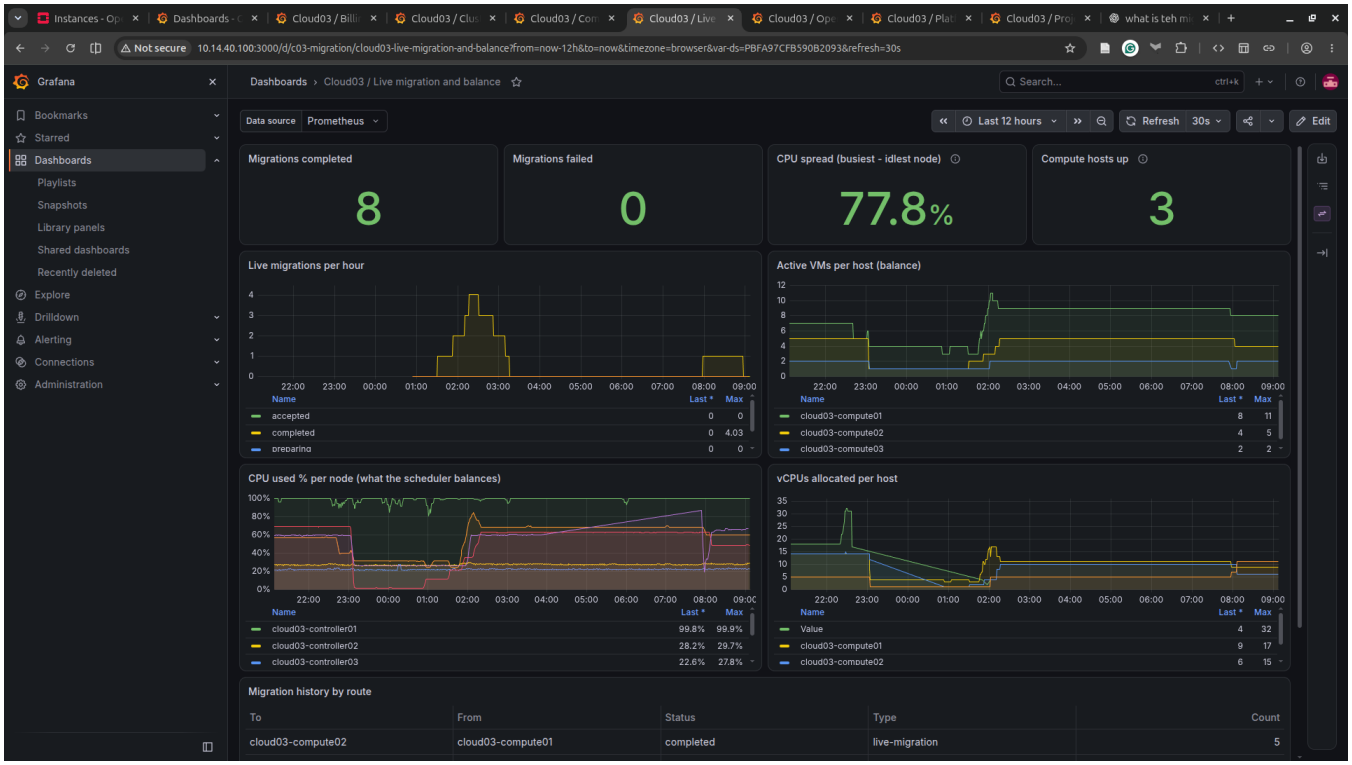

Figure 21. Live migration and balance after eight completed migrations, with route history.