Loss of Execution Truth in Layered Cloud Orchestration

A Case Study of OpenStack and Kubernetes Under CPU Oversubscription

Alexey A. Nekludoff

AstraVerge Research

E-mail: an@astraverge.org
ORCID: 0009-0002-7724-5762

January 2026

Report type: Zenodo Report (technical case study)
Disclaimer: The case is anonymized; identifiers are illustrative and do not disclose internal infrastructure details.

Abstract

Layered infrastructure stacks routinely combine virtual-machine orchestration (e.g., OpenStack) with container orchestration (e.g., Kubernetes) deployed *inside* virtual machines. While administratively convenient, this architecture creates multiple independent control planes competing over a shared execution substrate. This report analyzes a production incident in which CPU oversubscription and cluster-wide saturation led to loss of controllability: virtual machines remained *logically* `ACTIVE` while being *physically* non-responsive; live migration became impossible; and recovery required a forced hypervisor reboot. We formalize this failure mode as *loss of execution truth*—a divergence between control-plane state and execution-plane reality—and model its amplification through layered oversubscription. We further show that this divergence is fundamentally observational and introduce a COE/GOP-based perspective for detecting execution incoherence across independent system localities. Together, these results provide a unified explanation for why automation and self-healing mechanisms fail under saturation in modern layered cloud platforms.

Keywords: OpenStack; Nova; Kubernetes-in-VM; layered orchestration; overcommit; oversubscription; SRE; control plane; execution plane; observability; live migration; CPU starvation; observational coherence; execution incoherence; COE; GOP.

Contents

1 Introduction **1**
- 1.1 Orchestration vs. execution . . . 1
- 1.2 Why layered control is fragile . . . 1
- 1.3 Contributions . . . 2

2 System architecture **2**
- 2.1 Control planes and execution planes . . . 2
- 2.2 OpenStack layer (VM orchestration) . . . 2
- 2.3 Kubernetes layer (pod orchestration inside VMs) . . . 2
- 2.4 Resource model and oversubscription . . . 3

3 Incident description **4**
- 3.1 Environment overview . . . 4
- 3.2 Timeline . . . 4
- 3.3 Observed symptoms . . . 5

4 Failure analysis **5**
- 4.1 VM `ACTIVE` $\neq$ VM executing . . . 5
- 4.2 Why live migration failed under CPU starvation . . . 6
- 4.3 Cluster-wide saturation as a systemic precondition . . . 6

5 Loss of execution truth **6**
- 5.1 Definition . . . 6
- 5.2 Invariants of execution truth . . . 7
- 5.3 Mechanism . . . 8
- 5.4 Observability failure modes . . . 9

6 Implications for cloud architecture **9**
- 6.1 Layered orchestration and its limits . . . 9
- 6.2 Kubernetes on virtual machines . . . 9
- 6.3 Overcommit as a reliability control, not a financial tool . . . 10
- 6.4 Operational recommendations . . . 10
- 6.5 Process maturity vs. execution understanding . . . 10

7 COE/GOP perspective **11**

8 Conclusions **11**

A Minimal glossary **13**

B Operationalization of Execution Truth **13**
- B.1 Observational metrics . . . 13
- B.2 Indicative thresholds . . . 13
- B.3 Divergence function . . . 13
- B.4 Reproducible load testing . . . 14
- B.5 Recovery and fencing playbook . . . 14
- B.6 Logging and tracing requirements . . . 14

1 Introduction

Modern cloud infrastructures are built around the idea of orchestration: the ability to allocate, schedule, and rearrange computational resources dynamically in response to demand. Platforms such as OpenStack and Kubernetes provide powerful abstractions that allow operators and application teams to treat large collections of servers as elastic pools of virtual machines and containers. These abstractions enable rapid provisioning, multi-tenant isolation, and high degrees of automation.

However, behind these abstractions lies a physical execution substrate that remains finite, discrete, and subject to fundamental constraints. CPU cores, memory bandwidth, and I/O channels cannot be oversubscribed indefinitely without consequences. When orchestration layers are stacked, as in environments where Kubernetes is deployed on top of OpenStack-provisioned virtual machines, the gap between abstract resource models and physical execution can grow large enough to produce qualitatively new failure modes.

This report examines one such failure: a production incident in which CPU oversubscription led to a complete loss of operational control, forcing a hard reboot of a virtualization host despite the presence of multiple orchestration layers designed to provide resilience and self-healing.

1.1 Orchestration vs. execution

It is useful to distinguish between two conceptually different parts of a cloud system. The *control plane* consists of APIs, schedulers, databases, and automation components that decide what should run where and in what quantity. OpenStack Nova, Kubernetes schedulers, and their associated management services belong to this plane.

The *execution plane*, in contrast, consists of the components that actually execute workloads: hypervisors, virtual machine processes, operating system kernels, and ultimately the physical CPUs and memory controllers. This plane is governed by the physics of computation: finite cores, finite caches, finite memory bandwidth, and finite I/O capacity.

Under normal operating conditions, the control plane provides a useful abstraction over the execution plane. When resources are plentiful, scheduling decisions made by orchestration systems are faithfully realized by the hardware. Under heavy load, however, the execution plane may no longer be able to honor the assumptions embedded in control-plane models. When this happens, control-plane state can remain internally consistent while losing its ability to predict or constrain what actually happens on the hardware.

1.2 Why layered control is fragile

Layered cloud architectures introduce multiple independent control planes, each with its own resource model and scheduling logic. In a typical OpenStack–Kubernetes deployment, OpenStack schedules virtual machines based on vCPU and memory allocations, while Kubernetes schedules containers based on CPU and memory requests and limits within those virtual machines. Both layers, however, ultimately compete for the same physical resources.

When utilization is low, this layering appears benign: spare capacity absorbs inconsistencies between virtual and physical resource models. As utilization increases, especially under aggressive overcommit, the discrepancy between what the control planes believe to be available and what the hardware can actually deliver grows. Because each layer operates on its own abstract view of capacity, no single component has a complete or authoritative picture of physical constraints.

The hypothesis underlying this report is that stacked schedulers combined with cluster-wide oversubscription inevitably lead to a divergence between control-plane state and execution-plane reality. This divergence, which we term *loss of execution truth*, manifests not merely as performance degradation but as a breakdown of the system's ability to enact its own recovery and management policies.

1.3 Contributions

This report contributes:

- A compact incident-driven characterization of *loss of execution truth* in a layered OpenStack–Kubernetes setting.
- An analysis of why standard remediation actions, notably live migration, fail under conditions of severe CPU starvation.
- A set of architecture-level implications and mitigations that do not rely on vendor-specific assumptions.

2 System architecture

2.1 Control planes and execution planes

The environment under study is best understood as a composition of distinct control planes operating over a shared execution substrate. A *control plane* consists of software components that represent, schedule, and manage computational resources through models and state machines. An *execution plane* consists of the components that actually execute workloads and enforce physical constraints.

In the analyzed system, two major control planes coexist. The first is OpenStack, which manages the lifecycle and placement of virtual machines. The second is Kubernetes, which manages containers and services running inside those virtual machines. Both control planes issue commands and maintain state based on abstract resource models, yet neither has direct authority over the physical CPUs and memory.

The execution plane, by contrast, consists of the hypervisor (KVM/QEMU), the Linux kernel on the virtualization hosts, and the physical hardware itself. It is here that virtual CPUs are mapped to physical cores and where scheduling decisions are ultimately enforced. A central observation of this work is that failures arise when the control planes continue to operate on their abstract models while the execution plane is no longer able to honor them.

2.2 OpenStack layer (VM orchestration)

OpenStack provides an infrastructure-as-a-service layer for managing pools of compute, storage, and network resources. In the studied environment, its Nova component is responsible for scheduling virtual machines onto physical compute hosts, tracking their lifecycle states, and coordinating management operations such as start, stop, and live migration [5].

From the perspective of OpenStack, each compute host exposes a quantity of virtual CPUs and memory that can be allocated to instances. These quantities may be overcommitted, allowing the total virtual capacity assigned to virtual machines to exceed the physical capacity of the host. OpenStack relies on periodic heartbeats and hypervisor-reported metrics to maintain a model of host health and instance status.

Crucially, OpenStack's authority extends only to the level of virtual machines. It does not observe or control the containerized workloads running inside those machines, nor does it directly measure the instantaneous scheduling latency or execution progress of virtual CPUs on physical cores.

2.3 Kubernetes layer (pod orchestration inside VMs)

Kubernetes is deployed inside the virtual machines provided by OpenStack and operates as a platform for containerized applications. Within each virtual machine, Kubernetes schedules pods based on declared CPU and memory requests, limits, and quality-of-service classes [3].

This deployment model is commonly chosen for organizational reasons: virtual machines provide tenant isolation, accounting boundaries, and security domains, while Kubernetes provides application teams

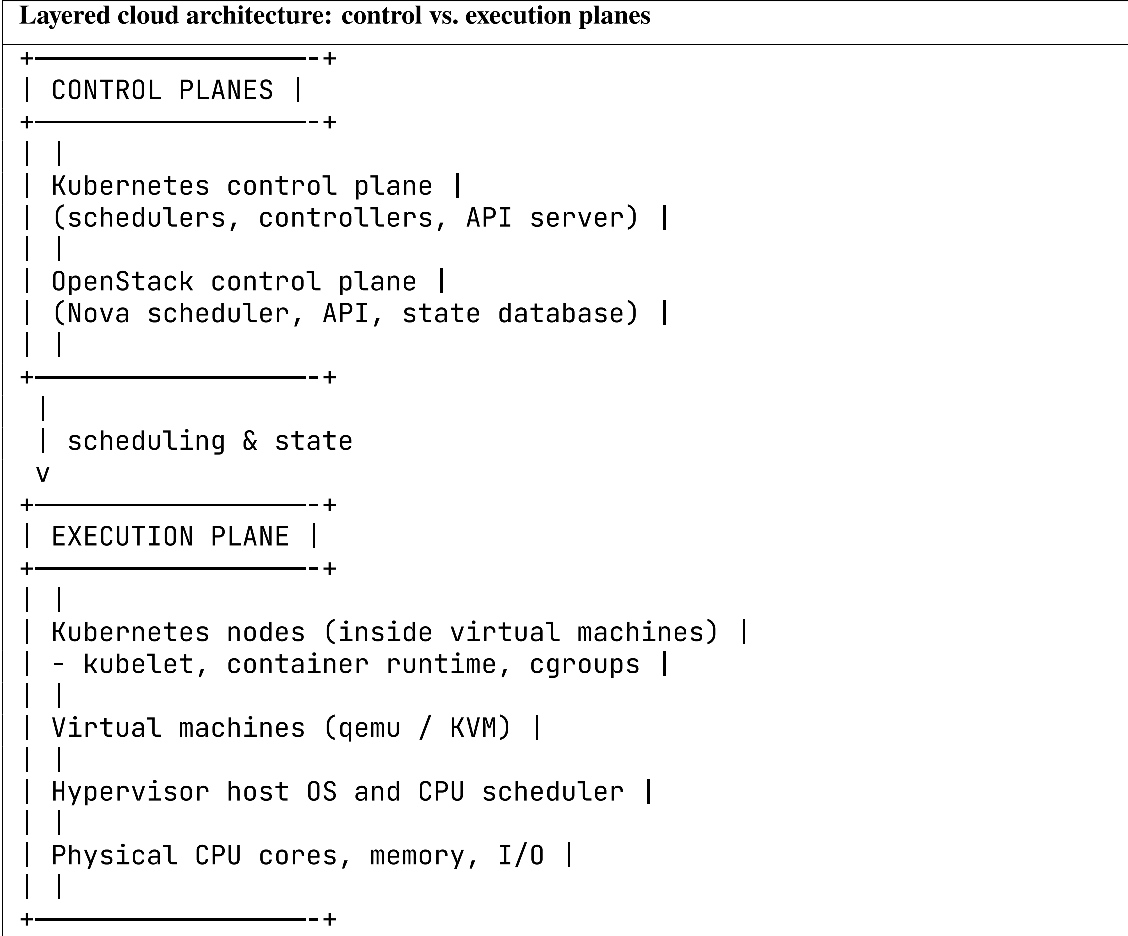

Figure 1: Separation between control planes and the execution plane in a layered OpenStack–Kubernetes environment. Control planes reason about abstract resource models, while the execution plane enforces physical constraints. Loss of execution truth occurs when these layers diverge under load.

with a flexible and self-service platform. From Kubernetes' point of view, each virtual machine is a node with a fixed amount of CPU and memory resources.

However, Kubernetes has no direct visibility into the physical state of the hypervisor or the host CPU scheduler. It assumes that the virtual resources reported by the guest operating system correspond to physically available capacity, an assumption that becomes increasingly fragile as overcommit increases at the virtualization layer.

2.4 Resource model and oversubscription

Both OpenStack and Kubernetes rely on abstract resource models. Let C denote the number of physical CPU cores available on a compute host, and let V denote the total number of virtual CPUs assigned to all virtual machines scheduled on that host. The overcommit or oversubscription ratio is then

$$r = \frac{V}{C}.$$

An overcommit ratio $r > 1$ implies that more virtual CPU capacity has been promised than can be delivered simultaneously by the hardware. While moderate oversubscription can be efficient when workloads are bursty or idle, high values of r make sustained peak load unavoidable and increase the probability of CPU starvation.

In a layered environment, Kubernetes performs its own scheduling within each virtual machine based

on virtual CPU counts that are themselves derived from V. As a result, effective contention for physical cores is multiplied across layers, amplifying the risk that control-plane assumptions about available capacity will diverge from execution-plane reality.

When Kubernetes is deployed inside virtual machines, a second level of oversubscription is introduced. Let v be the number of vCPUs assigned to a given VM and P the total CPU capacity requested or limited by the pods scheduled within that VM. The inner oversubscription ratio is

$$r_2 = \frac{P}{v}.$$

The effective contention for physical CPU time is then approximately bounded by the product of the two ratios,

$$r_{\text{eff}} \approx r_1 \cdot r_2 = \frac{V}{C} \cdot \frac{P}{v}.$$

This relation, illustrated in Fig. 2, captures how layered schedulers amplify each other's overcommit decisions, even when each layer appears locally reasonable.

Double oversubscription in layered orchestration (Nova + Kubernetes)
Step 1: OpenStack (host-level) Physical host: C physical cores OpenStack assigns VMs: total V vCPUs Overcommit ratio: $r_1 = V/C$ **Step 2: Kubernetes (VM-level)** Each VM reports v vCPUs to the guest OS Kubernetes schedules pods with total requests/limits P CPU units (conceptually mapped onto v) Inner oversubscription: $r_2 = P/v$ **Effective contention (execution plane)** Guest workloads compete for vCPU time vCPU threads compete for host CPU time Effective amplification (simplified): $r_{\text{eff}} \approx r_1 \cdot r_2$ **Failure mode under peak load** If r_{eff} is high and load is sustained: → CPU starvation of VMs and hypervisor daemons → management operations time out (e.g., live migration) → control-plane state diverges from execution reality → *loss of execution truth*

Figure 2: Two independent schedulers oversubscribe the same physical CPU budget: OpenStack at the host level and Kubernetes inside VMs. The effective contention can be seen as an amplification of oversubscription across layers, increasing the probability of CPU starvation and loss of execution truth.

3 Incident description

3.1 Environment overview

3.2 Timeline

The incident unfolded during an early-morning maintenance window of a large OpenStack-based enterprise cloud. A compute host belonging to a production OpenStack cluster experienced sustained CPU saturation, causing all virtual machines resident on that host to become unresponsive. Attempts to recover by means of live migration failed, after which the hypervisor was fully rebooted.

Table 1 summarizes the main operational events.

Time (local)	Event
01:14	Start of service degradation in the OpenStack compute cluster. CPU load rises across the cluster, with a particular compute host becoming critically oversubscribed.
01:14–02:02	All virtual machines located on the affected host become unresponsive. The OpenStack control plane continues to report them as `ACTIVE`. Live migration attempts fail.
02:02	Decision taken to reboot the virtualization host after it is determined that recovery through orchestration is not possible. Hypervisor reboot is executed.
02:02–02:25	Post-reboot phase: virtualization services come back online; virtual machines are restarted and partially redistributed across other hosts in the already overloaded cluster.
02:25	OpenStack virtualization service is declared restored. Verification of affected applications begins.
After 02:25	Some application owners are unreachable, leaving part of the business impact unverified until daytime operations.

Table 1: Timeline of the OpenStack compute failure and recovery.

3.3 Observed symptoms

The failure exhibited a characteristic pattern associated with severe oversubscription and loss of control over the execution layer:

- **Host-level CPU saturation.** The compute node hosting the affected virtual machines was overloaded to the point where normal scheduling of virtual CPU threads became unreliable.
- **Virtual machines reported as `ACTIVE` but non-responsive.** From the perspective of the OpenStack control plane, the virtual machines remained in an operational state, while in practice they were unable to process workloads or respond to management operations.
- **Live migration failure.** Attempts to evacuate or migrate virtual machines away from the overloaded host failed, indicating that the hypervisor and its management interfaces (e.g., libvirt/qemu) were no longer able to perform coordinated state transfer under extreme CPU pressure.
- **Forced hypervisor reboot as the only recovery mechanism.** Because orchestration-level actions could not be executed, recovery required a full reboot of the physical virtualization host, abruptly terminating and then restarting all resident virtual machines.
- **Partial and delayed assessment of business impact.** While at least one business application (campaign management) was confirmed to be affected, several others could not be verified at the time of recovery due to the unavailability of responsible application owners.

Together, these symptoms indicate a divergence between control-plane state and execution-plane reality, a condition we analyze in the following sections as a loss of execution truth.

4 Failure analysis

4.1 VM `ACTIVE` $\neq$ VM executing

In OpenStack, the `ACTIVE` state of a virtual machine is a control-plane assertion rather than a physical guarantee. It indicates that the Nova scheduler and its associated databases believe the instance to be running, but it does not imply that the virtual CPUs of that instance are actually being scheduled on physical cores at sufficient rates to execute instructions.

Under normal operating conditions, this abstraction is acceptable: hypervisor processes (e.g., `qemu`) remain responsive, heartbeats from compute services are timely, and management operations can be performed. However, when a compute host becomes severely oversubscribed at the CPU level, this correspondence breaks down. Virtual machine processes may receive only sporadic CPU time, or be effectively starved, while still appearing alive to higher layers.

As a result, the OpenStack control plane may continue to report instances as `ACTIVE` even when, from the perspective of applications and users, the machines are effectively non-functional. This creates a semantic gap between logical state and execution reality, in which the system claims liveness while being unable to deliver computational progress.

4.2 Why live migration failed under CPU starvation

Live migration of a virtual machine is an inherently resource-intensive operation. It requires that the source and destination hypervisors remain responsive, that sufficient CPU cycles be available to copy memory pages and synchronize device state, and that management daemons (such as `libvirt`) can exchange control messages in a timely fashion.

In the incident under study, the compute host was already operating under extreme CPU pressure. The virtual CPU threads of guest machines were competing not only with each other but also with the hypervisor's own management processes. Under these conditions, the prerequisites for live migration were violated: neither the hypervisor nor the guest instances could reliably participate in the migration protocol.

Consequently, migration requests issued by the OpenStack control plane did not result in successful evacuation of the affected virtual machines. Instead, they stalled or failed, leaving instances trapped on a host that was no longer capable of executing them. This represents a form of orchestration deadlock, in which the very lack of resources that necessitates migration also prevents it from occurring.

4.3 Cluster-wide saturation as a systemic precondition

A critical detail in the incident report is that the entire OpenStack compute cluster was described as overloaded, not merely the single host that eventually failed. This cluster-wide saturation is a key systemic precondition for the observed behavior.

When spare capacity exists elsewhere in the cluster, an overloaded host can be relieved by migrating or restarting instances on less loaded nodes. In contrast, when oversubscription is pervasive, no safe target for relocation exists. Every potential destination host is itself operating near or beyond its effective capacity.

Under such conditions, orchestration mechanisms lose their ability to re-balance load, and failures local to a single compute node propagate into cluster-level loss of control. The system enters a regime in which both preventive (scheduling) and corrective (migration, evacuation) controls are neutralized by the absence of physical headroom.

5 Loss of execution truth

5.1 Definition

We define *loss of execution truth* as a condition in which the authoritative control-plane state of a system (e.g., `ACTIVE`, `UP`, `Ready`) remains internally consistent, yet ceases to reflect or constrain the actual execution of workloads on physical hardware.

Formally, let S_c denote the control-plane state and S_e the execution-plane state. Under normal operation, S_c provides a predictive and enforceable abstraction over S_e: commands issued to the control plane result in corresponding changes in execution. Loss of execution truth occurs when this relation is broken, i.e., when $S_c \not\Rightarrow S_e$ and S_e no longer responds to S_c.

In the incident analyzed here, virtual machines remained in the `ACTIVE` state according to OpenStack, while their virtual CPUs were unable to make forward progress due to physical CPU starvation on the host. The control plane asserted liveness and controllability, while the execution plane was effectively stalled.

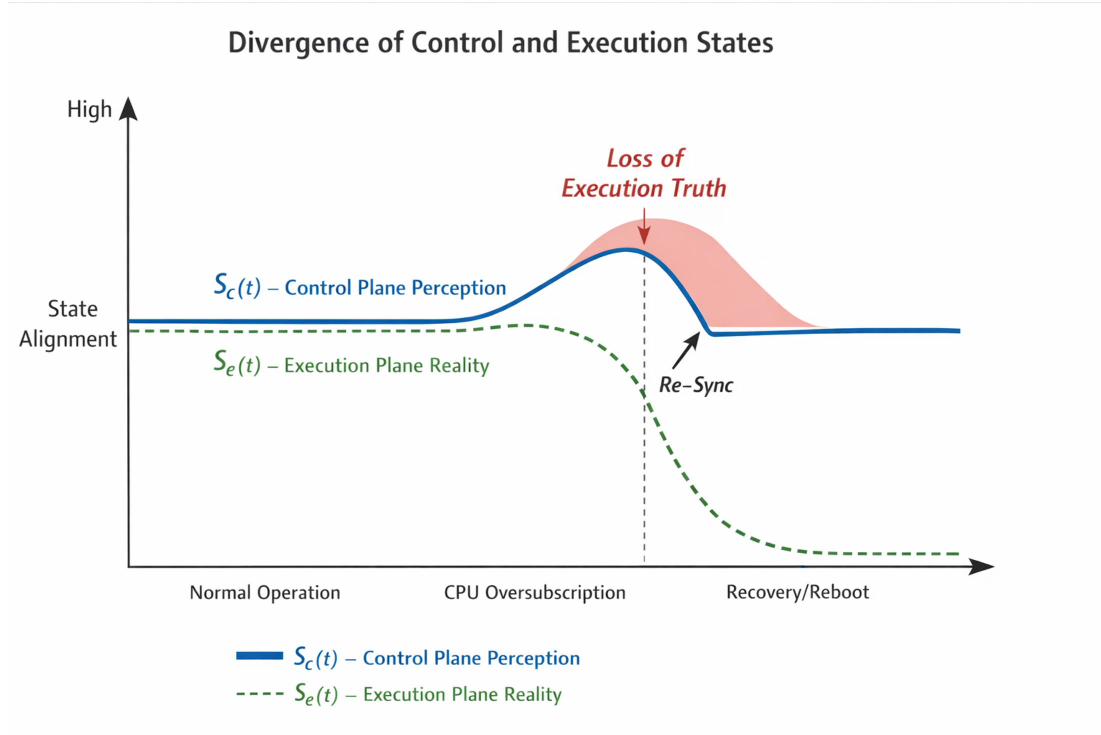

Figure 3: Divergence between control-plane state $S_c(t)$ and execution-plane state $S_e(t)$ under CPU oversubscription. During normal operation the two states are aligned; under saturation they diverge, producing a period of *loss of execution truth*, until forced recovery (e.g., reboot) re-synchronizes the control plane with physical reality.

Axioms of Execution Truth

Axiom 1 (Execution primacy). The physical execution of workloads on CPU, memory, and I/O devices is the only source of ground truth about system liveness and progress.

Axiom 2 (Control-plane representation). Control planes maintain abstract representations of system state that approximate execution, but do not constitute execution themselves.

Axiom 3 (Coherence requirement). A system is operationally reliable only while its control-plane state remains coherently aligned with its execution-plane state.

Axiom 4 (Layered divergence). When multiple independent control planes manage the same execution substrate, no layer can guarantee coherence unless it has privileged access to execution truth.

Axiom 5 (Failure condition). Loss of execution truth occurs when control-plane representations continue to evolve while execution-plane progress is stalled or degraded, rendering orchestration and automation ineffective.

5.2 Invariants of execution truth

The axioms of execution truth imply a set of invariants that must hold for any cloud system to remain operationally coherent, regardless of its specific implementation.

Invariant 1 (Single execution truth). At any moment, there exists a unique physical execution state $S_e(t)$ that determines whether workloads make forward progress. All control-plane representations are

approximations of this state.

Invariant 2 (Bounded divergence). The distance between control-plane state $S_c(t)$ and execution-plane state $S_e(t)$ must remain bounded. When this distance grows without bound, orchestration becomes decoupled from reality.

Invariant 3 (Observability coupling). Any control plane capable of taking recovery actions must have timely access to execution-plane signals relevant to those actions.

Invariant 4 (Recovery feasibility). A system must preserve sufficient execution-plane capacity to perform its own recovery operations. If oversubscription eliminates this capacity, automation becomes self-defeating.

Invariant 5 (Coherence under composition). When multiple control planes are composed, their combined representation must still satisfy Invariants 1–4 with respect to the same execution plane. Violations of this condition lead to loss of execution truth.

5.3 Mechanism

The mechanism leading to loss of execution truth in layered cloud environments can be summarized as a cascade:

1. **Oversubscription.** The total virtual CPU capacity provisioned by the orchestration layers exceeds the physical CPU capacity of the cluster.

2. **CPU starvation.** Under peak load, hypervisor processes and guest virtual machines compete for insufficient physical CPU cycles, causing both application workloads and management daemons to become sluggish or unresponsive.

3. **Management path degradation.** Control-plane components (e.g., `nova-compute`, `libvirt`, and hypervisor RPC handlers) are themselves subject to CPU scheduling delays, preventing timely execution of orchestration commands.

4. **State divergence.** The control plane continues to report instances as `ACTIVE` and hosts as `UP` based on stale or delayed heartbeats, while the execution plane can no longer guarantee progress.

5. **Orchestration deadlock.** Corrective actions such as live migration require precisely the resources that have been exhausted. As a result, the system cannot use orchestration to escape the failure mode.

The final outcome is a system that appears healthy according to its own control interfaces, yet is incapable of executing workloads or performing self-healing actions.

The amplification effect expressed by

$$r_{\text{eff}} \approx r_1 \cdot r_2$$

provides a compact explanation for the observed collapse. Even moderate oversubscription at each layer can result in extreme effective contention at the execution plane. As r_{eff} grows, both guest workloads and hypervisor management processes become subject to CPU starvation, accelerating the divergence between control plane state and execution plane reality.

In this regime, orchestration mechanisms depend on precisely the resources that have been exhausted, making loss of execution truth not an incidental failure but a mathematically predictable outcome of layered overcommit.

This divergence is illustrated in Fig. 3.

5.4 Observability failure modes

Loss of execution truth undermines observability in a fundamental way. Metrics, health checks, and status reports are generated by components that are themselves subject to the same resource starvation as the workloads they are meant to observe.

In the studied incident, this led to a situation in which:

- virtual machines were reported as `ACTIVE` despite being unresponsive;
- orchestration actions (e.g., migration) were acknowledged but not completed;
- business services exhibited failures that were only partially detected and confirmed.

Such conditions create a dangerous form of operational blindness: engineers and automation systems act on control plane data that no longer corresponds to physical reality, delaying effective intervention and increasing the scope of impact.

6 Implications for cloud architecture

The architectural implications discussed in this section follow directly from the axioms of execution truth introduced in Section 5. In particular, any cloud architecture that violates the coherence requirement between control-plane and execution-plane state (Axiom 3) or allows multiple independent control planes to compete over the same execution substrate (Axiom 4) is structurally predisposed to loss of execution truth under load.

6.1 Layered orchestration and its limits

The architecture analyzed in this case combines two independent orchestration layers: OpenStack for virtual machines and Kubernetes for containers deployed inside those virtual machines. Each layer implements its own scheduling, health checking, and recovery logic, yet both ultimately compete for the same underlying physical resources.

Under moderate load, this layered orchestration appears to function correctly, as spare capacity masks the mismatch between virtualized resource models and physical reality. However, under high utilization or bursty workloads, the lack of a unified view of physical capacity leads to uncontrolled oversubscription and, as shown in this incident, to loss of execution truth.

This suggests that layered orchestration should not be treated as a neutral architectural choice. It introduces a systemic risk that grows superlinearly with utilization and workload variability.

6.2 Kubernetes on virtual machines

Running Kubernetes inside virtual machines provides organizational benefits such as tenant isolation, accounting, and delegated administration. However, it also places Kubernetes inside a simulated resource environment whose capacity and performance characteristics are controlled by an external scheduler (OpenStack Nova).

As a result, Kubernetes cannot directly observe or react to physical CPU and memory pressure. From its perspective, a node may have "16 cores" even when the hypervisor is unable to deliver more than a fraction of that capacity in real time. This undermines core Kubernetes mechanisms such as quality-of-service classes, eviction policies, and autoscaling, all of which assume that the node's reported resources correspond to physical reality.

6.3 Overcommit as a reliability control, not a financial tool

CPU and memory overcommit are often treated primarily as economic instruments: they increase utilization and improve return on investment. The incident analyzed here demonstrates that, beyond a certain threshold, oversubscription becomes a dominant reliability risk.

In particular, when overcommit ratios are high across an entire cluster, no safe destination exists for evacuation or load shedding. The system loses not only performance margin but also its ability to perform corrective actions such as live migration or rescheduling.

Overcommit should therefore be governed as a reliability control, with explicit headroom reserved for recovery operations and for the control-plane components themselves.

6.4 Operational recommendations

Based on the analysis, we propose several vendor-neutral measures to mitigate loss of execution truth in layered cloud environments:

- **Preserve physical headroom.** Enforce cluster-level capacity limits that reserve CPU and memory for management and recovery operations.
- **Establish a single source of execution truth.** Define which layer (hypervisor, VM, or container runtime) is authoritative for physical resource availability, and ensure that higher layers treat it as such.
- **Detect and fail fast on unresponsive hosts.** Prefer aggressive fencing and host isolation over prolonged attempts at orchestration-level recovery when hypervisor responsiveness is lost.
- **Align observability with the execution plane.** Monitor physical CPU scheduling delays, hypervisor responsiveness, and management-daemon latency, not only logical VM or pod states.
- **Test recovery under saturation.** Validate live migration, evacuation, and failover procedures under deliberately induced high-load conditions, not only in nominal regimes.

6.5 Process maturity vs. execution understanding

Large-scale cloud environments are typically governed by extensive operational processes: incident response playbooks, escalation paths, service ownership models, and formal postmortem procedures. These mechanisms are essential for coordinating large teams and ensuring accountability. However, the incident analyzed in this report demonstrates a fundamental limitation of process-driven reliability.

Processes operate on the control-plane representation of a system: status dashboards, health checks, alerts, and incident tickets all reflect what orchestration layers *believe* about system state. When loss of execution truth occurs, these representations remain internally consistent while becoming disconnected from physical reality. Under such conditions, even well-designed processes guide operators toward actions that cannot be effectively executed.

In particular, escalation procedures, service-level ownership, and communication protocols cannot compensate for a lack of understanding of how execution actually occurs on physical hardware. When hypervisors and virtual CPUs are starved of physical CPU time, no amount of procedural rigor can make live migration succeed or restore responsiveness. Recovery then depends on ad hoc interventions, such as forced reboots, that bypass the orchestration stack entirely.

This leads to a general principle: *processes do not replace knowledge*. Reliable operation of layered cloud platforms requires that engineers and operators understand not only the APIs and workflows of control planes but also the execution-plane mechanisms that ultimately enforce scheduling and resource allocation. Without this knowledge, organizations risk building ever more elaborate processes around abstractions that no longer correspond to reality.

In layered cloud architectures, therefore, operational maturity must be coupled with execution-level expertise. Only by maintaining an accurate mental model of how virtual CPUs map onto physical cores, and how management daemons compete with workloads for those cores, can teams recognize the onset of loss of execution truth and intervene before control is lost.

From the perspective of the axioms of execution truth, organizational processes and escalation paths operate entirely within the control plane. When execution-plane reality diverges from control-plane representation, processes continue to function while their referent vanishes. This explains why, in layered cloud failures, procedural maturity cannot substitute for execution-level understanding.

While the preceding sections focus on architectural and organizational implications of layered cloud failures, they leave open a critical question: how can such loss of execution truth be detected before it forces disruptive intervention? Addressing this requires a shift from treating control-plane metrics as authoritative to treating cross-layer observation coherence as a first-class operational property. The following section introduces a COE/GOP-based perspective that formalizes this shift.

7 COE/GOP perspective

This section reframes the failure in terms of observational coherence across independent system localities.

The failure analyzed in this report can also be interpreted through the lens of cross-locality observation and event coherence. In layered cloud architectures, multiple subsystems independently observe and report on system state: OpenStack reports instance status, Kubernetes reports node and pod health, hypervisors expose process-level signals, and physical hosts provide CPU scheduling and I/O metrics. Each of these sources constitutes a distinct observation locality with its own timing, resolution, and failure modes.

The incident described here demonstrates what happens when these observation localities lose coherence. OpenStack reported virtual machines as `ACTIVE`, Kubernetes continued to report nodes and pods as running, while the hypervisor and the physical CPU scheduler were already unable to deliver execution. Each subsystem was internally consistent, yet globally misleading.

From a COE (Coherent Observational Epistemology) [4] perspective, this represents a breakdown of inter-locality coherence. A Global Observation Protocol (GOP) would not treat any single control-plane signal as authoritative. Instead, it would continuously align and compare independent observation streams—control-plane heartbeats, hypervisor responsiveness, CPU scheduling delays, and workload-level liveness—to detect when their implied realities diverge.

In practical terms, a GOP-inspired system would flag a state of *execution incoherence* when, for example, control-plane liveness remains positive while execution-plane signals (e.g., sustained CPU run queue length, qemu scheduling latency, or stalled guest clocks) indicate loss of forward progress. Such a condition would invalidate normal orchestration decisions and trigger an explicit transition to a degraded or emergency control mode.

Viewed in this way, loss of execution truth is not merely a technical fault but a failure of observational alignment. COE and GOP provide a conceptual framework for restoring this alignment by making disagreement between localities first-class and actionable, rather than implicitly trusting any single layer of abstraction.

8 Conclusions

This report examined a production failure in a layered OpenStack–Kubernetes environment in which severe CPU oversubscription led to loss of controllability, failed recovery actions, and a forced hypervisor reboot. The incident illustrates a fundamental failure mode of modern cloud stacks: when multiple orchestration layers compete for the same physical resources without a shared notion of execution truth, control-plane state can diverge from execution-plane reality.

By introducing the concept of *loss of execution truth* and formalizing its dynamics through layered oversubscription and control–execution divergence, this work provides a compact and general explanation for why self-healing and automation mechanisms fail precisely under conditions of cluster-wide saturation. The failure is not an anomaly but an emergent property of architectures that violate the coherence requirement between abstract resource models and physical execution.

The COE/GOP perspective further shows that this failure mode is fundamentally observational. Independent control planes and monitoring systems each report internally consistent states, yet without cross-locality alignment they produce a globally misleading picture of system health. Treating observational coherence as a first-class operational property is therefore essential for detecting and mitigating loss of execution truth before it forces disruptive recovery actions.

Finally, this case highlights a broader organizational lesson: while mature operational processes are necessary, they are not sufficient. *Processes do not replace knowledge*. When layered control planes lose their connection to the execution plane, only engineers who understand the physical mechanisms of scheduling and resource contention can recognize and resolve the failure, regardless of how well incident workflows and escalation paths are defined.

References

[1] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, Eds., *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016.

[2] B. Ibryam and R. Huß, *Kubernetes Patterns*. O'Reilly Media, 2019.

[3] "Kubernetes documentation," Kubernetes, Accessed: Jan. 15, 2026. [Online]. Available: `https://kubernetes.io/docs/`

[4] A. A. Nekludoff, *Coherent observational epistemology: Foundational principles, secondary principles, and axiomatic system*, Zenodo Report, 2025. DOI: `10.5281/zenodo.17632756` [Online]. Available: `https://doi.org/10.5281/zenodo.17632756`

[5] "Openstack nova documentation," OpenStack, Accessed: Jan. 15, 2026. [Online]. Available: `https://docs.openstack.org/nova/latest/`

A Minimal glossary

- **Control plane:** components that decide and record desired state (APIs, schedulers, databases).
- **Execution plane:** components that execute workloads (hypervisors, kernels, CPU schedulers).
- **Overcommit/oversubscription:** provisioning more virtual resources than physical capacity.

B Operationalization of Execution Truth

This appendix translates the theory of execution truth into measurable signals, reproducible tests, and operational procedures. Its purpose is to provide a concrete, testable framework for detecting and responding to divergence between control-plane state $S_c(t)$ and execution-plane state $S_e(t)$ in layered cloud environments.

B.1 Observational metrics

The following metrics are required to observe execution-plane degradation and control–execution divergence:

- **CPU steal (%)** — fraction of time vCPUs wait for physical CPU.
- **Host runqueue length** — number of runnable processes on the hypervisor.
- **qemu/libvirt response latency (ms)** — management daemon responsiveness.
- **libvirt RPC timeout rate (%)** — fraction of failed or timed-out management calls.
- **Heartbeat lag (s)** — delay between expected and received compute heartbeats.
- **kubelet scheduling delay (ms)** — time to place or evict pods inside a VM.
- **Pod eviction rate** — frequency of QoS-based evictions.
- **I/O wait and disk latency (ms)** — storage-layer execution delays.
- **Management CPU share (%)** — CPU consumed by nova, libvirt, and related daemons.

B.2 Indicative thresholds

The following thresholds provide an initial, environment-dependent guideline:

- CPU steal $> 10\%$ for 1 minute $\rightarrow$ warning; $> 25\%$ $\rightarrow$ critical.
- qemu/libvirt p95 latency > 200 ms $\rightarrow$ warning; > 500 ms $\rightarrow$ critical.
- libvirt RPC timeout rate $> 1\%$ over 5 minutes $\rightarrow$ critical.
- Heartbeat lag > 30 s $\rightarrow$ critical.
- Host runqueue length $> 2\times$ CPU cores $\rightarrow$ warning; $> 4\times$ $\rightarrow$ critical.
- kubelet scheduling delay p95 > 500 ms $\rightarrow$ warning.
- I/O p99 latency > 100 ms $\rightarrow$ warning; > 300 ms $\rightarrow$ critical.

B.3 Divergence function

Let $m_i(t) \in [0, 1]$ be normalized metrics derived from the signals above. Define the divergence function:

$$d(t) = \sum_i w_i m_i(t),$$

where w_i are weights reflecting diagnostic importance (e.g., CPU steal 0.25, qemu latency 0.2, heartbeat lag 0.2, runqueue 0.15, kubelet delay 0.1, I/O 0.1).

Indicative thresholds:

$$d_{\text{warn}} = 0.4, \qquad d_{\text{crit}} = 0.7.$$

Reaching d_{crit} indicates loss of execution truth and requires immediate isolation or fencing of the affected host.

B.4 Reproducible load testing

To validate the theory, layered oversubscription must be stress-tested:

- **CPU-only oversubscription:** set host-level $r_1 = 1.5, 2.0, 3.0$ and VM-level $r_2 = 1.0, 1.5, 2.0$ using tools such as `stress-ng`.
- **Cluster-wide saturation:** load all hosts to remove migration targets.
- **Management-path stress:** generate nova and libvirt API load under high CPU contention.
- **I/O-bound variant:** combine moderate CPU load with high disk p99 latency.

All metrics from Section B.1 and migration outcomes should be recorded to determine the minimal r_{eff} at which orchestration becomes unreliable.

B.5 Recovery and fencing playbook

Early warning ($d \geq d_{\text{warn}}$):

- Alert on-call engineers and freeze new scheduling on suspect hosts.
- Collect metrics snapshots and management logs.
- Assess available cluster headroom for planned evacuation.

Critical state ($d \geq d_{\text{crit}}$ or heartbeat lag threshold exceeded):

- Immediately fence the host in the control plane.
- Attempt controlled live migration if management daemons remain responsive.
- If migration fails, perform forced hypervisor reboot via out-of-band control.

Post-recovery:

- Verify VM integrity and application health.
- Recompute r_1, r_2, and r_{eff} for the incident.
- Adjust overcommit and headroom policies accordingly.

B.6 Logging and tracing requirements

All control-plane and hypervisor logs should include at minimum: timestamp, component, request_id, rpc_latency_ms, cpu_steal_pct, host_id, vm_id, action, and outcome. Distributed tracing should be enabled for control-plane to compute-host RPC paths to localize sources of delay and divergence.