Week 9: Resource Allocation and Autoscaling

NT524 — Cloud Architecture and Security

PhD. Nguyen Ngoc Tu

November 19, 2025

Resource allocation — phân bổ tài nguyên; Autoscaling — tự co giãn

Outline

Recap (1/4): Control-loop & Artifact-first Platform

- **Unifying mental model:** Platform as a reconciliation/control loop:
  - Intent → Controller → Actuators → Observability → Feedback.
- **Artifact guarantees:**
  - Golden images, signatures, SBOMs, digest pinning.
- **Platform mapping across clouds:**

Concept	OpenStack	AWS	Azure	GCP
Images / Artifacts	Glance	AMI	Managed Images	Compute Images
Compute / VMs	Nova	EC2	VM Compute	Compute Engine
Orchestration	Heat	CloudFormation	ARM/Bicep	Deployment Manager
Monitoring / Alarms	Aodh	CloudWatch	Azure Monitor	Cloud Monitoring
Identity / Policy	Keystone	IAM	Azure AD + RBAC	IAM

Control loop – Vòng điều khiển; Artifact immutability – Tính bất biến của ảnh; Reconciliation – Đồng bộ hoá; Orchestration(Heat, CloudFormation,...) – control-plane components, declarative orchestration engine

Recap (2/4): Sizing Playbook

1. Sizing Playbook (Deterministic Baseline)

- Benchmark per-instance throughput at SLO latency → capacity C (req/s).
- Compute required fleet size:

$$N_{\text{min}} = \left\lceil \frac{\lambda_{\text{peak}}}{C \cdot u_t \cdot s} \right\rceil$$

  where:
  - λ_{peak}: forecasted peak load
  - u_t: target utilization (e.g., 60–70%)
  - s: safety factor (e.g., 1.15–1.30)
- Add capacity buffers:
  - **Warm pool**: pre-started instances for fast reaction.
  - **Surge buffer**: extra nodes for sudden load spikes.

Sizing playbook – Quy trình tính kích thước; SLO – Mục tiêu mức dịch vụ

Recap (3/4): Scaling Signals & Stability Controls

2. Good Scaling Signals (Leading Indicators)

- **Latency (p90/p99)**: earliest congestion signal.
- **Queue depth**: backlog in message/task/request queues.
- **RPS / utilization** with smoothing (EWMA).
- Optional: error rate, CPU steal, GC time, memory pressure.

3. Stability Controls (Anti-flap Mechanisms)

- **Cooldown**: minimum interval between scale actions.
- **Hysteresis**: different thresholds for scale-out vs scale-in.
- **Evaluation windows**: require signal stability (n-of-m).
- **Step scaling**: add/remove capacity proportional to deficit.

Scaling signal – Tín hiệu mở rộng; Hysteresis – Vùng trễ; p90/p99 (percentile latency): độ trễ ở phân vị 90%/ 99% (p90: 10% trễ hơn mức p90; p99: 1% trễ hơn mức p99)

Recap (4/4): Operational Gaps & Next Steps

1. Operational Gaps Identified

- **Telemetry trust:** metrics and logs not fully validated → risk of wrong scaling/alerts.
- **Cross-plane runbooks:** missing standardized procedures connecting compute, network, storage, and orchestration layers.
- **Hardware-aware rules limited:** GPU, NUMA, NIC-specific operational constraints not consistently applied.
- **Artifact governance gaps:** image signing, SBOM verification, digest pinning not enforced across environments.

2. Recommended Next Steps

- **Telemetry improvements:**
  - Introduce **metric provenance** (source identification, chain of custody).
  - Deploy **synthetic probes** to validate telemetry in production.
- **Integrated operational lab:**
  - End-to-end workflow: build → sign → deploy → scale → consolidate.
  - Include cross-plane coordination, resource policies, and automated testing.
- **Documentation / Runbooks:** standardize operational playbooks for cross-layer actions and incident response.

Warm-up (1/2): Cross-Platform Artifact Lifecycle

Common artifact types across all clouds:

- **Compute artifacts:** VM images (Glance, AMI, SIG, GCE Images), snapshots, OVAs.
- **Container artifacts:** OCI images, multi-arch builds, signed digests (Cosign).
- **Network artifacts:** subnet descriptors, SG/NSG/ACL policy bundles, LB configs.
- **Storage artifacts:** volume images, snapshots, object versioning, replication rules.
- **Configuration bundles:** cloud-init, Heat templates, CFN/Bicep/Terraform modules.
- **Secrets & identity:** Barbican keys, IAM roles, MSI/ServiceAccounts, KMS CMKs.

A universal lifecycle: build → scan → sign → publish → deploy → verify → retire

- **Build:** Packer, Dockerfiles, image pipelines, CI runners.
- **Scan:** CVE scans, SBOM generation, policy checks, compliance gates.
- **Sign:** image signatures, digest pinning, provenance attestations.
- **Publish:** Glance/AMI/SIG/GCR; OCI registries; Helm charts.
- **Deploy:** Heat/CFN/Bicep/Terraform; orchestrators; autoscaling systems.
- **Verify:** health checks, telemetry, canary/policy validation.

Warm-up (2/2): Activities on OpenStack

1. **VM Image Lifecycle (Glance)**

- Import a custom image → validate properties → lock with image ID pinning.
- Build a *golden image* that reduces cold-start time for autoscaling groups.

2. **Network Artifact Lifecycle (Neutron)**

- Define networks/subnets/routers as declarative artifacts (YAML or Heat).
- Attach SG/ACL policies that autoscaling groups will inherit.

3. **Storage Artifact Lifecycle (Cinder/Swift)**

- Create volumes + snapshots; benchmark IOPS for sizing decisions.
- Explore Cinder QoS and how storage throttling affects SLO signals.

4. **Container Artifact Lifecycle (Manual or Magnum/K8s)**

- Build and push OCI-compliant images; verify signatures; deploy inside VMs.

5. **Deployment Artifacts (Heat)**

- Compose Heat stacks for ASG, LB, alarms, networks.
- Tie VM image IDs, SG policies, and metrics into a single declarative pipeline.

Learning Objectives

By the end of Week 9, you should be able to:

- Explain how **resource allocation** policies (right-sizing, reservation, oversubscription) impact SLOs, cost and security.
- Compare **vertical** and **horizontal** scaling strategies, including their operational limits in OpenStack and public clouds.
- Design **autoscaling policies** using metrics, thresholds and control-loop parameters (cooldown, hysteresis, headroom).
- Apply **resource optimization** techniques (scheduling, placement, consolidation) to a realistic OpenStack deployment.
- Critically evaluate scaling patterns and cost trade-offs for different workload classes (APIs, batch, data, ML).

SLO (Service Level Objective) — mục tiêu mức dịch vụ; right-sizing — định cỡ phù hợp; oversubscription — cấp phát vượt; hysteresis — trễ đàn hồi

Telemetry & Metric Trust (1/5): Why Telemetry Matters

- **Telemetry = the control-plane truth source.**
  - Autoscaling, consolidation, placement, anomaly detection all depend on metrics.
- **Bad telemetry → bad decisions:**
  - Missing metrics → scaling stops or becomes unsafe.
  - Delayed metrics → overreaction or late reaction.
  - Corrupted metrics → false scale-out storms.
- **Goal:** ensure metrics are **accurate, timely, trusted, and complete**.

Telemetry – Giám sát số liệu; Control-plane – Mặt phẳng điều khiển; Metric corruption – Số liệu bị sai lệch

Telemetry & Metric Trust (2/5): Ingest Pipeline Architecture

- **End-to-end pipeline:**
  - Exporters (VM/Container agents)
  - $\rightarrow$ Collectors (Prometheus / Monasca / Ceilometer)
  - $\rightarrow$ Short-term store (alarms, autoscaler)
  - $\rightarrow$ Long-term store (analytics, dashboards)
- **Failure hotspots:**
  - Exporter crash
  - Collector overload
  - Network loss between agents
  - Delayed write to metrics DB
- **Design implication:** Autoscaler must assume telemetry can fail.

Metric exporter – Bộ thu thập số liệu; Collector – Trình thu gom; Pipeline – Chuỗi xử lý

Telemetry & Metric Trust (3/5): Metric Quality Dimensions

- **1. Accuracy** – no distortion, correct sampling behavior.
- **2. Freshness** – low latency between measurement and ingestion.
- **3. Completeness** – no silent drops or missing intervals.
- **4. Consistency** – stable meaning across updates.
- **5. Provenance** – measurement source is identifiable & trusted.

Practical rules

- Set **metric TTL** (e.g., 2–3 minutes) $\rightarrow$ stale metrics invalid.
- Mark data gaps explicitly $\rightarrow$ autoscaler must fallback.
- Define "minimum metric set" required before making decisions.

Metric freshness – Độ tươi của số liệu; Metric provenance – Nguồn gốc số liệu; Consistency – Tính nhất quán

Telemetry & Metric Trust (4/5): Synthetic Probes & Cross-checks

- **Synthetic probes:**
  - Active checks (HTTP ping, TLS handshake, db query).
  - Used when internal metrics may be missing.
- **Cross-checks:**
  - RPS vs queue length
  - Latency vs CPU saturation
  - Error rate vs upstream dependency
- **Trust rule:**
  - Autoscaler uses **“majority vote”**: internal metric + synthetic probe must agree.

Synthetic probe – Kiểm tra tổng hợp; Cross-check – Đối chiếu chéo; Dependency – Phụ thuộc hệ thống

Telemetry & Metric Trust (5/5): Fail-safe Autoscaling Rules

- **When metrics degrade:**
  - No scale-out unless synthetic probes confirm overload.
  - Allow safe scale-in only when multiple signals agree.
  - Switch to "conservative mode" until validity restored.
- **Fail-safe modes:**
  - Freeze desired capacity temporarily.
  - Use warm-pool only (avoid creating new workloads).
  - Alert operators if metric gaps exceed threshold.
- **Outcome:** Prevent scale storms, flapping, & false-positive reactions.

Fail-safe mode – Chế độ an toàn; Scale storm – Bão mở rộng tài nguyên; Metric gap – Khoảng trống số liệu

Why Resource Allocation & Autoscaling Matter

Context: Week 8 defined the platform as a **reconciliation(synchronization)** control loop. This section operationalizes the compute slice of that loop: how to provision, trade off, and autoscale *safely*.

Key drivers (concise):

- **Demand variability:** diurnal, marketing bursts, nightly batch, ML jobs.
- **Cost pressure:** idle resources $\rightarrow$ direct spend; spot/preemptible options $\leftrightarrow$ complexity.
- **SLOs & risk:** p95/p99 latency, error budgets, RTO/RPO for failures.
- **Multi-tenancy & security:** isolation, fairness, quotas, noisy neighbors.

This section goal: give operators a small decision algebra + runbooks so scale decisions are **predictable, auditable and reversible**.

Demand variability – Biến động nhu cầu; Cost pressure – Áp lực chi phí; SLO – Mục tiêu chất lượng dịch vụ; Multi-tenancy – Đa thuê

Case Study: Compute Control Loop — Abstract Elements

1. Inputs (Intent / Desired State)

- SLO target (latency, throughput)
- Min / Max VM count, budget constraints
- Placement rules (affinity/anti-affinity, availability zones, hardware features)

2. Signals (Observability / Metrics)

- Application: RPS, queue depth, error rate, p90/p95/p99 latency
- System: CPU, memory, disk I/O, network throughput, VM health
- Custom metrics: app-specific KPIs, GPU utilization, NUMA locality, SR-IOV stats

3. Controller (Decision / Policy Layer)

- Scaling policy type: reactive, scheduled, predictive
- Stability heuristics: cooldown intervals, hysteresis, step scaling, evaluation windows
- Optimization: cost-aware, resource packing, risk mitigation (noisy neighbor)

SLO – Mục tiêu mức dịch vụ; RPS – Requests per second; Hysteresis – Vùng trễ; Cooldown – Khoảng nghỉ giữa các hành động

Case Study: Compute Control Loop — OpenStack Mapping

4. Actuators / Actions (OpenStack Services)

- `Nova`: create / resize / delete VMs
- `Heat`: autoscaling groups, orchestration templates
- `Senlin`: cluster lifecycle, rolling updates, group policies
- `Octavia`: load balancer provisioning
- `Cinder`: block volume attach/detach

5. Concrete Resource Mapping

- `image (Glance)`: digest pinning for deterministic boot
- `flavor / extra_specs`: CPU policy, NUMA, hugepages, SR-IOV
- `Heat + Aodh`: operator-friendly autoscale + alarm triggers
- `Senlin`: stateful cluster policies for instance groups
- `Placement / Scheduler hints`: anti-affinity, hardware constraints, availability zones

---

Nova – Máy chủ ảo; Heat – Orchestration; Aodh – Dịch vụ báo động; Glance – Quản lý kho ảnh; Senlin – Quản lý cluster stateful

Right-sizing — formula and measurement

Objective: select smallest flavor meeting SLO at expected load while leaving safe headroom.

Process:

1. Benchmark candidate flavor and measure sustainable throughput C (req/s) at target p95 latency.
2. Choose target utilization u_t (0.6–0.75) and safety factor s (1.1–1.4).
3. Compute baseline instances:

$$N_{\min} = \left\lceil \frac{\lambda_{peak}}{C \cdot u_t \cdot s} \right\rceil$$

4. Add warm pool W (10%–20%) and surge buffer B (10%–30%) as needed.

Throughput – Lưu lượng; Warm pool – Nhóm máy ảo khởi tạo sẵn; Surge buffer – Dự trữ tăng cường; SLO – Mục tiêu mức dịch vụ

Right-sizing — worked numeric example

Example:

- $\lambda_{peak} = 1400$ req/s
- $C = 200$ req/s at target p95
- $u_t = 0.7 \;\Rightarrow\; 200 \times 0.7 = 140$
- $s = 1.3 \;\Rightarrow\; 140 \times 1.3 = 182.0$
- Baseline instances: $1400 \div 182.0 \approx 7.6923 \Rightarrow N_{\text{min}} = \lceil 7.6923 \rceil = 8$
- Warm pool $W = 10\% \times 8 = 0.8 \to 1$
- Surge buffer $B = 20\% \times 8 = 1.6 \to 2$
- **Desired capacity** $\approx 8 + 1 + 2 = 11$

Throughput – Lưu lượng; Warm pool – Nhóm máy ảo khởi tạo sẵn; Surge buffer – Dự trữ tăng cường; p95 – Độ trễ 95%

Reservations, Limits & Oversubscription — Operator rules

Definitions (short):

- Reservation = guaranteed allocation (reserved hosts, pinning).
- Limit = quota cap (project-level max vCPU/RAM/IOPS).
- Oversubscription = assign virtual resources beyond physical inventory (vCPU:PCPU > 1).

Operator rules-of-thumb:

- **Critical workloads:** use dedicated pools / low overcommit (vCPU:PCPU $\leq$ 1–1.5), CPU pinning, NUMA alignment.
- **Batch / dev:** higher overcommit (2–6$\times$) acceptable if preemptible or interruptible.
- **Memory:** avoid aggressive oversub on memory for latency-sensitive services — use ballooning only for low-priority VMs.
- **IO:** quantify IOPS per workload and enforce IOPS quotas / throttles; use local NVMe for high IOPS stateful workloads.

Monitoring & alarms:

- Raise alert when host CPU steal, memory swap, or I/O queue length exceed thresholds for sustained windows.
- Autoscaler should avoid scaling into a host pool with $>$ 80% host saturation (bảo hòa).

Tenancy, Flavors, Placement — Practical patterns

Flavor taxonomy:

- **general-purpose** — balanced vCPU / RAM.
- **compute-optimized** — higher vCPU, CPU-pin options.
- **memory-optimized** — large RAM, NUMA-aware.
- **io-optimized** — local NVMe, tuned I/O scheduler.
- **gpu-accelerated** — GPU partitioning, MIG mapping.

Placement & isolation primitives:

- Host aggregates + availability zones → map to isolation tiers (prod/dev, regulated workloads).
- Placement policies & scheduler hints → enforce affinity/anti-affinity.
- Quotas per project → cap blast radius of autoscale.

Autoscale implication:

- Scaling attempts must validate flavor availability and tenant quotas before creating instances.
- Provide separate AutoScalingGroups per flavor-class / isolation tier to avoid cross-class failures.

Decision Tree — Scale-up vs Scale-out

Decision tree (simple):

1. Is the service **stateless**? → prefer scale-out (replicas) behind LB.
2. If stateful or single-threaded bottleneck? → consider scale-up (bigger flavor).
3. Is placement constrained (GPU/NUMA/SRIOV)? → prefer reserved pools or scheduled scaling windows.
4. If cold-start penalty high (heavy boot): use warm-pool or pre-warmed images.

Scale-out – Tăng số lượng bản sao; Scale-up – Tăng kích thước máy ảo; LB – Load balancer; Cold-start – Thời gian khởi tạo

Runbooks — Scale-up vs Scale-out

Scale-out runbook:

Scale-out (stateless)

1. Verify flavor + quota available (API checks).
2. Check image digest & warm-pool availability.
3. Trigger Heat ASG scale action; verify Octavia pool health checks register new instance.
4. Monitor p95 latency and error-rate for 5 minutes; rollback if error spike $>$ threshold.

Scale-up runbook (stateful):

Scale-up (stateful)

1. Quiesce writes, snapshot DB (Cinder snapshot).
2. Live-migrate to larger host or resize instance (Nova resize + confirm).
3. Re-run health checks and promote back to service.

Practical checklist & failure modes to test

Pre-deployment checklist:

- Pin Glance image by immutable ID; bake minimal boot actions into image.
- Create dedicated flavor classes and pre-provision warm pools for critical services.
- Define headroom u_t, safety factor s, warm pool
- Implement metric provenance + probe cross-check (see Telemetry section).
- Add quota & placement validation in autoscaler pre-check.

Failure-mode tests (must-run):

- Simulate image registry slow / stale → verify warm pool fallback.
- Simulate collector partition → autoscaler enters conservative mode.
- Force host saturation → verify live-migration / consolidation coordination.
- Trigger quota exhaustion → autoscaler receives failure & alerts operator.

Warm pool – Nhóm khởi tạo sẵn; Headroom – Dự trữ; Provenance – Nguồn gốc; Quota exhaustion – Hết hạn ngạch

Vertical Scaling (Scale-Up) — Deeper Analysis

Vertical scaling = increase (or decrease) the capacity of *one* instance. **Characteristics (expanded):**

- Resize to a larger flavor: more vCPU, RAM, IOPS, cache, NUMA nodes.
- Requires **interruptive operations**: stop/restart, or live-migration if workloads permit.
- Best for **stateful** or tightly coupled services: DBs, JVM monoliths, brokers, legacy apps.
- Improves **per-thread performance** when code is not horizontally parallelizable.

Limits & degradation modes:

- Hard ceiling defined by the largest host $\rightarrow$ cannot exceed physical NUMA topology.
- Larger VMs increase **failure blast radius**.
- Diminishing returns: lock contention, shared cache saturation, cross-NUMA penalties.

---

vertical scaling – mở rộng theo chiều dọc; NUMA – bộ nhớ không đồng nhất; blast radius – bán kính ảnh hưởng; lock contention – tranh chấp khoá

Horizontal Scaling (Scale-Out) — Expanded View

Horizontal scaling = add/remove *replicas* behind an LB or partitioned topology.

Patterns (more precise):

- **Stateless tiers**: REST APIs, web frontends, async processing.
- **Partitioned workloads**: sharded DBs, distributed queues, worker pools.
- **Read-heavy architectures**: caches, search clusters, replicated read-only stores.

Advantages:

- Improves **availability** via multi-AZ distribution.
- Scales linearly until hitting shared bottlenecks (DB, network, metadata services).
- Fits perfectly into autoscaling groups (Heat/Senlin + Octavia).

Costs & constraints:

- More replicas → higher steady-state cost unless scale-in is aggressive and safe.
- Requires session decoupling, idempotency, and externalizing state (Redis, DB, object storage).

stateless – không trạng thái; partitioning – phân vùng; idempotency – tính bất biến theo số lần gọi; replica – bản sao

Vertical vs Horizontal — Decision Framework

When Vertical is appropriate:

- Single-node bottlenecks: JVM monoliths, OLTP databases, brokers.
- Latency-sensitive workloads requiring strong locality (NUMA awareness).
- Operations forbidden to replicate due to business logic or licensing.

When Horizontal wins:

- Stateless microservices w/ clear request boundaries.
- Traffic spikes, unpredictable bursts, diurnal load patterns.
- Need to distribute across racks/AZs for HA.

Hybrid strategy (the modern default):

- **Right-size first (vertical)** to find the best per-instance cost/performance.
- **Scale-out second** to ensure elasticity and resilience.
- Stateful layer: vertical scale + protective read replicas + caching.

---

microservice – dịch vụ vi mô; OLTP – xử lý giao dịch trực tuyến; elasticity – tính co giãn; bottleneck – nút nghẽn

Autoscaling Fundamentals — from Signals to Safe Actions

Choosing the right metrics (deepened):

- **Infrastructure**: CPU, memory pressure, run queue, NIC pps, disk latency.
- **Application**: queue depth, request latency, RPS, concurrency.
- **Business**: orders/min, active rooms, messages/sec.

A "good" signal must be:

- **SLO-correlated**: rising signal predicts user impact.
- **Smooth**: not too noisy or spiky (avoid CPU-only triggers).
- **Fresh**: low telemetry delay ($\leq$ 30–60s end-to-end).

Threshold logic (properly engineered):

- Use target headroom ($u_t = 60\% - 70\%$) to absorb spikes.
- Implement separate up/down thresholds to avoid flapping.
- Trigger scale-in only after sustained low load (3–5 intervals).

---

queue depth – độ sâu hàng đợi; concurrency – số kết nối đồng thời; headroom – phần dự trữ; flapping – dao động mở rộng/thu hẹp liên tục

Autoscaling Policies — Deep Revision & Practical Rules

Reactive policies (OpenStack default):

- Based on real-time thresholds (CPU, latency, queue).
- Pros: simple, transparent.
- Cons: **lag** and potential oscillation if telemetry noisy.

Scheduled policies:

- Use predictable weekly/daily load curves.
- Reduces sudden scale events; protects from telemetry gaps.

Predictive policies (future direction):

- Forecast-based: ARIMA, Holt-Winters, ML models.
- Must enforce: min/max bounds, budget caps, anomaly detection.

Hybrid recommended:

- Reactive for sudden load changes.
- Scheduled for known patterns.
- Predictive as icing when telemetry + data quality are guaranteed.

---

reactive scaling – mở rộng phản ứng; scheduled scaling – mở rộng theo lịch; predictive scaling – mở rộng dự báo; anomaly – bất thường

Stability Engineering — Cooldown, Hysteresis, Dampening

Why stability matters: Autoscaling without damping creates **thrashing**, cost spikes, and SLO violations.

Controls:

- **Cooldown**: lockout window between actions (90–180s).
- **Hysteresis**: scale-out at 70%, scale-in at 40%.
- **Lookback windows**: require 3–5 consecutive intervals.
- **Warm-pool integration**: reduce cold-start latency.

Operator practices:

- Validate with synthetic probes before and after scale events.
- Never allow scale-in while error-rate rising.
- Block scaling into aggregates >80% host saturation.

---

thrashing – giao động quá mức; dampening – giảm dao động; warm-pool – nhóm máy ảo khởi tạo sẵn; saturation – bão hòa

Scheduling Objectives in Modern Clouds

Classical view:

- Place VMs to satisfy capacity constraints (CPU, RAM, disk).
- Optimize for a single objective (e.g., utilization or availability).

State-of-the-art view: multi-objective scheduling

- **Performance**: latency, throughput, queueing delay.
- **Cost**: number of active hosts, licensing, cross-AZ traffic.
- **Energy & carbon**: total power, carbon intensity of each site.
- **Security & isolation**: side-channel risk, tenant separation.
- **Data locality**: distance to storage, caches, message queues.
- **Fairness**: no tenant dominates all shared resources.

Real schedulers combine several of these targets via **weights**, priorities, and **admission policies**.

---

objective — mục tiêu tối ưu; data locality — tính cục bộ dữ liệu; carbon intensity — cường độ cacbon; admission policy — chính sách tiếp nhận tải

Scheduling Strategies: Bin-Packing vs Spread

Bin-packing:

- Fill hosts as much as possible before using new ones.
- Pros: fewer active hosts, better energy and license efficiency.
- Cons: correlated failures; more risk for noisy neighbors and contention.

Spread:

- Distribute replicas across many hosts/racks/AZs.
- Pros: resilience to host/rack failure; better tenant isolation.
- Cons: more active hosts, higher base cost and energy use.

Hybrid strategies:

- *Bin-pack within a fault domain*, then spread across domains.
- *Soft spread*: prefer separation unless utilization too low.

OpenStack tools:

- Server groups with **affinity** / **anti-affinity** rules.
- Nova scheduler filters and weighers (RAM, CPU, I/O, aggregate, custom).

---

fault domain — miền lỗi; hybrid strategy — chiến lược lai; soft spread — phân tán mềm; weigher — bộ cho điểm

Topology- and Interference-Aware Placement

Hardware topology:

- Sockets, NUMA nodes, LLC (last-level cache) sharing.
- PCIe hierarchy: GPUs, NVMe, SR-IOV NICs on specific NUMA nodes.

Interference-aware scheduling:

- Avoid co-locating two latency-critical or IO-heavy tenants.
- Separate noisy batch jobs from low-latency services.
- Use perf counters / telemetry to score *contention hotspots*.

OpenStack patterns:

- NUMA-aware flavors, CPU pinning, hugepages for critical workloads.
- Host aggregates for *"noisy-batch"* vs *"latency-sensitive"* pools.

NUMA (Non-Uniform Memory Access) — bộ nhớ truy cập không đồng đều; interference — nhiễu lẫn; contention hotspot — điểm nóng tranh chấp; CPU pinning — ghim CPU

Network- and Data-Locality-Aware Placement

Network-aware placement:

- Minimize cross-AZ and cross-region traffic for chatty services.
- Respect bandwidth limits for storage or NFV data planes.
- Prefer short paths (same rack/leaf) for low-latency tiers.

Data locality:

- Place compute close to its primary data store (Cinder, Swift, Ceph).
- Respect **regulatory** data residency (country/region constraints).
- Cache-aware placement: keep workers near hot caches or message brokers.

Implementation hooks:

- Host aggregates labeled with storage tier / network tier.
- Custom Nova filters using latency or bandwidth metadata.

---

data residency — cư trú dữ liệu; chatty service — dịch vụ trao đổi nhiều; storage tier — tầng lưu trữ; NFV — ảo hoá chức năng mạng

Security- and Compliance-Aware Placement

Security-driven constraints:

- **Anti-co-location**: do not place certain tenants together (side-channel risk).
- **Trust zones**: only run sensitive workloads on hardened, attested hosts.
- **Blast-radius control**: separate production vs dev, high vs low trust levels.

Compliance and sovereignty:

- Place data and compute in specific regions or availability zones.
- Use label-based policies to enforce industry regulations (e.g., finance, healthcare).

OpenStack mechanisms:

- Projects mapped to dedicated aggregates or cells for high-trust tenants.
- Policy-as-code (e.g., OPA, custom admission hooks) on top of Nova placement.

co-location — đồng vị trí; side-channel — kênh kề; trust zone — vùng tin cậy; sovereignty — chủ quyền dữ liệu

Fair-Sharing and Multi-Resource Fairness

Challenge:

- Tenants consume *multiple* resources (CPU, RAM, IO, network, GPU).
- Naive per-resource limits can still allow one tenant to dominate the cluster.

Multi-resource fairness (idea):

- Model each tenant's **dominant resource share** (max fraction used of any resource).
- Schedule new work to keep dominant shares balanced across tenants.

Practical approximations:

- Weight quotas by tenant priority (gold/silver/bronze).
- Use per-tenant *fair-use dashboards* and alerts.
- Throttle new VM creation when a tenant's dominant share is too high.

---

multi-resource fairness — công bằng đa tài nguyên; dominant share — phần chi phối; gold/silver/bronze — hạng ưu tiên; throttle — bóp băng thông

Consolidation, Power Management and Drain

Consolidation loop:

- Periodically identify underutilized hosts (below threshold for some window).
- Live-migrate VMs away, then put hosts into *standby* or *maintenance*.

Power-aware strategies:

- Prefer shutting down entire hosts over small frequency changes.
- Align consolidation with **carbon intensity**: keep more hosts active when grid is green.
- Consider **wear-out**: avoid constantly power-cycling the same nodes.

Interactions with autoscaling:

- Autoscaling changes VM count; consolidation changes **host** count/usage.
- Need coordination to avoid **thrashing** (e.g., cooldown for both loops, shared view of demand).

---

power management — quản lý điện năng; carbon-aware — nhận thức về cacbon; wear-out — hao mòn; cooldown — thời gian hạ nhiệt

Live Migration Costs and SLA Awareness

Live migration is not free:

- CPU and network overhead to copy memory and state.
- Short performance dips and possible jitter for latency-sensitive apps.

SLA-aware consolidation:

- Never migrate during critical windows (e.g., trading hours, batch deadlines).
- Respect per-workload **migration budgets** (e.g., max moves per day).
- Group migrations to minimize repeated cache-warmup penalties.

Policy examples:

- Allow aggressive consolidation for dev/test; conservative for prod.
- Tag VMs as *"migration-sensitive"* vs *"migration-friendly"*.

---

live migration — di trú trực tiếp; SLA (Service Level Agreement) — thoả thuận mức dịch vụ; jitter — độ rung trễ; migration budget — ngân sách di trú

Quotas, Budgets and Guardrails

Quotas:

- Per-project limits on vCPU, RAM, instances, volumes, floating IPs, etc.
- Act as **hard safety bounds** for autoscaling groups and batch bursts.

Hierarchical quotas & showback:

- Organization $\rightarrow$ department $\rightarrow$ project hierarchy.
- Track consumption and *show back* cost to business units.

Budgets and budget-aware autoscaling:

- Estimate cost from flavor + runtime + license metrics.
- When budget is near exhaustion: freeze scale-out, degrade gracefully, or offload to cheaper tiers.

Runtime guardrails:

- Max instance count per scaling group and per tenant.
- Policy-as-code: deny scaling if violating security/compliance or budget policies.

---

hierarchical quota — hạn ngạch phân cấp; showback — hiển thị chi phí nội bộ; budget-aware autoscaling — tự co giãn theo ngân sách; graceful degradation — suy giảm có kiểm soát

Autoscaling Control-Loops & Cross-Cloud Mapping (1/2)

Concept	**OpenStack**	**AWS**	**Azure**	**GCP**
VM Fleet Manager	Heat AutoScalingGroup / Senlin Cluster	EC2 Auto Scaling Group (ASG)	VM Scale Set (VMSS)	Managed Instance Group (MIG)
Image / Artifact	Glance Image (ID-pinned)	AMI (ID-pinned)	Shared Image Gallery (SIG)	GCE Images / Image Families
Load Balancer	Octavia (L4/L7)	ALB / NLB	Azure Load Balancer / Application Gateway	Cloud Load Balancing (L4/L7)
Telemetry / Metrics	Ceilometer + Gnocchi / Monasca	CloudWatch Metrics	Azure Monitor	Cloud Monitoring / Cloud Logging
Alarm Engine	Aodh	CloudWatch Alarms	Autoscale Rules (Azure Monitor)	Cloud Monitoring Alert Policies
IaC Integration	Heat Orchestration Templates	CloudFormation / CDK	ARM / Bicep	Deployment Manager / Terraform

Autoscaling Control-Loops & Cross-Cloud Mapping (2/2)

Concept	OpenStack	AWS	Azure	GCP
Scaling Policies	Heat ScalingPolicy in/out	Target Tracking, Step, Scheduled	Metric, Scheduled, Predictive (Preview)	Target-based, Step, Schedule, Predictive
Warm Pool / Pre-Provisioning	Manual warm instances; staging pools via Heat	ASG Warm Pools (native)	VMSS pre-provisioned instances	MIG pre-provisioned instances
Health Checks	Octavia health monitors	ALB/NLB Target Health	Load Balancer / App Gateway probes	Backend health checks (HTTP/TCP)
Placement / Zones	AZs, host aggregates, server groups	Multi-AZ; Spot placement strategies	Availability Zones; Proximity Placement Groups	Zones and Regional MIGs
Security Model	Security Groups, Role-based policies, Barbican	Security Groups, IAM Roles	Network Security Groups (NSG), Managed Identities	Firewall Rules, IAM Service Accounts

Autoscaling Control-Loops: Motivation

Autoscaling is a **closed control loop** that changes system capacity to preserve SLOs under varying load.

Every autoscaling system must answer 5 design questions:

1. What **SLO signal** defines "stress" or "overload"?
2. How do we **evaluate** that signal (window, statistic, stability gates)?
3. How should we **scale** (step, ratio, warm pool, headroom)?
4. How do we protect against **telemetry failure**?
5. How do we guarantee **predictable boot behavior** (image, config, trust)?

This cookbook provides a battle-tested approach used in production cloud platforms.

---

control loop — vòng điều khiển; SLO — mục tiêu mức dịch vụ; telemetry — phép đo hệ thống; warm pool — cụm khởi sẵn

1. Choose the Primary SLO Signal

Primary rule: scale only on metrics that correlate with user experience.

Good SLO-driven signals:

- **p95 latency** (API, microservices)
- **Queue length / queue age** (batch workers, ETL, AI inference)
- **Request rate / concurrency** (gateways, HTTP frontends)
- **Custom business metrics** (orders/min, sessions, jobs waiting)

Weak signals on their own:

- CPU utilization (depends on workload mix)
- Memory usage (poor predictor of load)
- Network bytes (often correlates poorly with saturation)

Best practice: **Primary SLO signal + Secondary infrastructure guard**.

---

queue length — độ dài hàng đợi; latency — độ trễ; guard — bộ bảo vệ bổ sung

2. Define Evaluation Logic: Window, Statistic, Hysteresis

The autoscaler must reduce noise and prevent oscillation.

Evaluation window (EW):

- Use EW $=$ 3–5$\times$ the signal sampling interval.
- Example: 60s samples $\rightarrow$ 3–5 minute window.

Statistic:

- Prefer **percentiles** (p95, p99) over averages.
- For queues: **max(queue age)** or **mean(wait time)**.

Hysteresis:

- Scale-out threshold high (e.g., latency $>$ 150ms)
- Scale-in threshold low (e.g., latency $<$ 80ms)
- Avoids thrashing between thresholds.

window — cửa sổ đánh giá; hysteresis — trễ đàn hồi; oscillation — dao động

3. Capacity Formula: Sizing Minimums, Warm Pool, Headroom

Step 1 — Per-instance capacity:

$$C = \text{sustainable req/s per VM at p95 SLO boundary}$$

Step 2 — Minimum instance count:

$$N_{\text{min}} = \left\lceil \frac{\lambda_{\text{peak}}}{C \cdot u_t \cdot s} \right\rceil$$

where:

- λ_{peak}: peak request rate
- u_t: target utilization (0.6–0.75)
- s: safety factor (1.2–1.4)

Step 3 — Buffers:

- **Headroom**: 20–30% above normal load
- **Warm pool**: 1–2 pre-booted instances (10–20% of N_{min})
- **Surge buffer**: +20% for sudden traffic bursts

headroom — phần dự trữ; warm pool — nhóm khởi sẵn; surge buffer — bộ đệm dồn tải

4. Define Scaling Actions: Step, Rate, or Gradient

Scaling modes:

- **Step-based**: add/subtract a fixed number (e.g., +1, –1)
- **Proportional**: add capacity proportional to overload
- **Gradient-based**: continuous controller (rare; complex SRE teams)

Recommended pattern for cloud VMs:

- Step-out: +1 or +2 VMs
- Step-in: –1 VM after long low-load window
- Cooldown: 2–5 minutes for out; 5–15 minutes for in

Avoid aggressive scale-in; always prefer conservative contraction.

---

step-based — theo nấc; proportional — theo tỷ lệ; gradient-based — theo độ dốc

5. Telemetry Failure: Safe Mode & Cross-Checks

Telemetry is rarely 100% healthy. Autoscaling must degrade safely.

Failure detection:

- Missing metrics for > 2 intervals
- Flatlined metrics (stuck at constant)
- Delayed timestamps (clock skew, collector lag)

Safe-mode rules:

- **Freeze**: do not scale-in when metrics uncertain
- **Allow scale-out only** if synthetic probe confirms overload
- Switch evaluation to **probe latency**, **LB error rate**, or **queue age**

Cross-check triad:

- Primary metric (SLO)
- Synthetic probe
- Infrastructure guard (CPU steal, IO queue depth)

---

safe mode — chế độ an toàn; synthetic probe — bài kiểm tra nhân tạo; flatline — đường phẳng sai lệch

6. Artifact Pinning & Cold-Start Control

Artifact reproducibility determines how predictable the autoscaler is.

Best practice: image + config must be *pinned*:

- Pin Nova image by **image ID** (not by name)
- Pin container/OCI image by **immutable digest**
- Use Git commit hash for user-data config bundles

Cold-start mitigation:

- Prebake agents, runtime libraries, logs, security baselines → reduce boot time
- Keep user-data scripts **minimal** to avoid blocking during scale-out
- Warm pool (pre-booted nodes) eliminates cold-start delays entirely

Without artifact pinning, autoscaling is non-deterministic and unsafe.

---

artifact pinning — cố định hiện vật; immutable digest — mã băm bất biến; cold-start — khởi động lạnh

7. Full Control-Loop Recipe

(1) Signals

- Primary SLO: p95 latency, queue age, etc.
- Secondary guards: CPU steal, error rate, packet loss.

(2) Evaluation

- Window, statistic, hysteresis.
- Cooldown durations for scale-out/in.

(3) Actuators

- Step size ($\pm$1–2)
- Warm pool size
- Max/min capacity

(4) Telemetry failover

- Freeze scale-in, require probe confirmation for scale-out.

(5) Artifact pinning

- Image ID, digest, Git commit pinned.

These 5 elements make the control-loop **predictable, safe, and auditable**.

---

auditable — có thể kiểm toán; actuator — cơ cấu tác động; freeze — đóng băng tạm thời

OpenStack Building Blocks for Autoscaling

Core components:

- **Nova**: flavors, server groups, host aggregates, availability zones.
- **Heat**: orchestration templates, stacks, *AutoScalingGroup* or *ResourceGroup*.
- **Telemetry stack**: Ceilometer / Gnocchi (metrics) and **Aodh** (alarms).
- **Senlin**: cluster engine and scaling policies for advanced use cases (optional).
- **Octavia**: load balancers in front of horizontally scaled tiers.

Typical Heat-based pattern:

- Heat template defines an *AutoScalingGroup* of Nova servers.
- Heat **ScalingPolicy** resources describe how much to scale in/out.
- Aodh alarms watch metrics and call the policies' *alarm_url* to trigger scaling.
- Octavia pool registers members as instances are created/destroyed.

Alternative Senlin pattern:

- Senlin cluster manages a pool of servers; Heat or direct API defines the cluster.
- Aodh alarm actions call a Senlin *receiver* (scale-in/scale-out action).

Telemetry — thu thập số đo; ScalingPolicy — chính sách co giãn; receiver — điểm tiếp nhận hành động;

Heat Template Pattern (1/2): AutoScaling Group

Objective: express autoscaling as Infrastructure as Code (IaC).

```
heat_template_version: 2016-10-14
resources:
  web_group:
    type: OS::Heat::AutoScalingGroup
    properties:
      cooldown: 120
      desired_capacity: 3
      max_size: 20
      min_size: 2
      resource:
        type: OS::Nova::Server
        properties:
          flavor: m1.small
          image: web-golden-image
          networks: [{ network: web-net }]
```

AutoScalingGroup — nhóm máy tự động co giãn; desired_capacity — số máy mặc định; min/max_size — giới hạn số máy; cooldown — khoảng nghỉ giữa hành động co giãn

Heat Template Pattern (2/2): Scaling Policies & Alarms

```
scaleup_policy:
  type: OS::Heat::ScalingPolicy
  properties:
    auto_scaling_group_id: { get_resource: web_group }
    adjustment_type: change_in_capacity
    scaling_adjustment: 1
    cooldown: 120

scaledown_policy:
  type: OS::Heat::ScalingPolicy
  properties:
    auto_scaling_group_id: { get_resource: web_group }
    adjustment_type: change_in_capacity
    scaling_adjustment: -1
    cooldown: 300
cpu_high:
  type: OS::Aodh::Alarm
  properties:
    description: Scale out if avg CPU > 70% for 3 min
    meter_name: cpu_util
    statistic: avg
```

Design Considerations for OpenStack Autoscaling

Images and boot time:

- Use **golden images** per tier to minimize cold-start delay.
- Pre-configure agents, logging, security baselines; keep cloud-init/user-data small.
- Consider **pre-warmed** instances (warm pool) for very spiky workloads.

Metrics pipeline and alarms:

- Ensure Ceilometer/Gnocchi or Monasca collects metrics at sufficient granularity.
- Prefer **application-level** metrics (RPS, queue depth, latency) where possible.
- Validate alarm behavior in a test project; exercise both scale-out and scale-in paths.

Network and security:

- Security groups and ACLs must automatically apply to new instances (via Heat templates).
- Load balancer health checks must detect failed instances quickly and remove them.
- Ensure logs and metrics from new instances are shipped to central observability stack.

---

cloud-init — khởi tạo đám mây; pre-warmed instance — máy ảo khởi sẵn; health check — kiểm tra sức khoẻ; observability stack — bộ công cụ quan sát hệ thống

AWS Building Blocks for Autoscaling

Core components:

- **EC2 Auto Scaling Group (ASG)**: manages VM count, placement, and lifecycle.
- **Launch Template / Launch Configuration**: defines AMI, instance type, networking.
- **Elastic Load Balancing (ALB/NLB)**: distributes traffic; integrates with ASG health checks.
- **CloudWatch**: metric collection (CPU, RPS, latency, custom metrics).
- **CloudWatch Alarms**: threshold-based triggers for scaling policies.
- **Auto Scaling Policies**: target tracking, step scaling, scheduled scaling.
- **Warm Pools**: pre-initialized EC2 instances ready for immediate scale-out.

Typical ASG-based pattern:

- Launch Template defines how to create instances (AMI, type, SGs).
- ASG manages desired/min/max capacity.
- CloudWatch collects metrics; Alarms trigger scaling policies.
- ALB/NLB registers instances automatically via target groups.

Auto Scaling Group — nhóm tự co giãn; AMI — ảnh máy Amazon; target group — nhóm đích; CloudWatch — giám sát AWS

AWS Target-Tracking: Recommended Autoscaling Pattern

Objective: express autoscaling declaratively via target metrics.

```
{
  "AutoScalingGroupName": "web-asg",
  "MinSize": 2,
  "MaxSize": 20,
  "DesiredCapacity": 3,
  "TargetGroupARNs": ["arn:aws:...:targetgroup/web"],
  "MixedInstancesPolicy": {
    "LaunchTemplate": {
      "LaunchTemplateSpecification": {
        "LaunchTemplateName": "web-template",
        "Version": "$Latest"
      }
    }
  }
}

aws application-autoscaling put-scaling-policy \
  --policy-name cpu-target \
  --service-namespace ec2 \
```

AWS Autoscaling Policy Types

1. **Target Tracking (recommended)**
   - "Keep metric at X": e.g., CPU at 60–70%.
   - Equivalent to a PID-like controller without user complexity.
   - Best for web/API tiers.
2. **Step Scaling**
   - Trigger different scale amounts for different metric ranges.
   - Good for queue-based or bursty workloads.
3. **Scheduled Scaling**
   - Scale based on predictable patterns (office hours, campaigns).
   - Complements target tracking during stable periods.

---

step scaling — co giãn theo nấc; scheduled scaling — co giãn theo lịch; predefined metric — số đo dựng sẵn

AWS Metrics and Alarms: CloudWatch Pipeline

Metric sources:

- EC2 built-in metrics (CPU, network I/O, status checks).
- ALB metrics: request count, target response time, 4xx/5xx rate.
- SQS metrics: queue length, message age (excellent autoscaling signals).
- Custom app metrics via CloudWatch Agent or Embedded Metric Format (EMF).

CloudWatch Alarms:

- Threshold-based: CPU $> 70\%$ for 3×60s $\rightarrow$ scale out.
- Can target ASG policies directly.
- Combine multiple alarms (latency $+$ error rate) for safer decisions.

CloudWatch Agent — tác tử giám sát; queue age — tuổi hàng đợi; EMF — định dạng số đo nhúng

AWS Warm Pools and Cold-Start Mitigation

Cold-start cost:

- EC2 boot time (20–90s)
- App initialization (JVM warm-up, loading models, DB migrations)
- Security agent startup, logging pipeline registration

AWS Warm Pools:

- Pre-initialized instances kept in *Stopped* or *Running* state.
- Instant capacity on scale-out.
- Helps eliminate boot storms during traffic spikes.

Best practice:

- Use warm pools for latency-critical services.
- Prebake AMI with all dependencies; avoid heavy user-data boot scripts.

---

warm pool — cụm máy khởi sẵn; boot storm — bão khởi động; prebake — dựng trước

Network, Placement and Security in AWS Autoscaling

Networking:

- Instances join the ASG's subnets; use Multi-AZ for resilience.
- ALB Target Groups auto-register new instances.
- Health checks remove unhealthy nodes from rotation.

Placement:

- Spread Across AZs (default) for availability.
- Capacity-optimized strategies for Spot instances (avoid interruptions).
- MixedInstancesPolicy for heterogeneous instance types.

Security:

- Security Groups auto-attach to new instances.
- IAM roles via instance profiles; rotate credentials automatically.
- AMIs must be signed and version-pinned for scaling determinism.

Multi-AZ — đa vùng sẵn sàng; Spot — phiên bản giá rẻ; instance profile — hồ sơ phiên bản

Design Considerations for AWS Autoscaling

1. **AMI quality & boot time**

- Keep AMI images minimal but fully preconfigured.
- Pin AMIs by ID; avoid name-based drifting.

2. **Metric design**

- Prefer ALB latency / SQS queue age / app RPS over CPU.
- Use CloudWatch high-resolution metrics (1s–10s) only when needed.

3. **Scaling safety**

- Scale-out fast; scale-in cautiously.
- Protect against telemetry gaps: use ALB health + status checks.

4. **Multi-tenancy & quotas**

- Use Service Quotas and AWS Budgets for guardrails.
- Prevent runaway scale-out during sudden failures or DDoS.

---

high-resolution metric — số đo độ phân giải cao; telemetry gap — mất dữ liệu giám sát; Service Quotas — hạn ngạch AWS

Azure Building Blocks for Autoscaling

Core components:

- **Virtual Machine Scale Sets (VMSS)**: Azure's native autoscaling fleet.
- **Images**: Shared Image Gallery (SIG) or custom VM images.
- **Azure Monitor**: unified metrics platform for VMs, LB, queues, apps.
- **Autoscale Rules**: metric-based or schedule-based scaling policies.
- **Azure Load Balancer (L4)** and **Application Gateway (L7)**: distribute traffic and provide health probes.
- **Azure Autoscale Engine**: performs scale-in/out for VMSS based on rules.
- **Proximity Placement Groups (PPG)**: low-latency co-location for HPC/NFV.

Typical VMSS-based pattern:

- VMSS defines image, size, networking, zones.
- Azure Autoscale monitors metrics from Azure Monitor.
- Autoscale engine adjusts VM count.
- LB or Application Gateway registers/deregisters VM instances.

Scale Set — bộ máy ảo co giãn; Shared Image Gallery — bộ sưu tập ảnh chia sẻ; health probe — thăm dò sức khoẻ

Azure VMSS Autoscaling Pattern (Bicep Example)

Objective: express autoscaling as Infrastructure as Code using Bicep.

```
resource vmss 'Microsoft.Compute/virtualMachineScaleSets@2023-03-01' = {
  name: 'web-vmss'
  location: resourceGroup().location
  sku: {
    name: 'Standard_DS2_v2'
    capacity: 3
  }
  properties: {
    upgradePolicy: { mode: 'Rolling' }
    virtualMachineProfile: {
      storageProfile: {
        imageReference: {
          id: '/subscriptions/.../galleries/webSIG/images/webImage/versions/1.0.0'
        }
      }
      networkProfile: {
        networkInterfaceConfigurations: [
          {
            name: 'webnic'
```

Azure Autoscaling Policy Types

1. Metric-Based Autoscale (most common)

- Triggered by Azure Monitor metrics.
- CPU, memory (via guest agent), disk queue depth.
- For app services: request rate, latency, error percentage.

2. Scheduled Autoscale

- Scale for business hours, promotions, or predictable cycles.

3. Custom Metrics Autoscale

- Use Application Insights, EventHub, or custom push metrics.
- Ideal for queue length, job backlog, Kafka lag, or business KPIs.

4. Predictive Autoscale (Preview / Selective Regions)

- Uses ML to predict demand 30 minutes ahead.
- Good for periodic workloads (diurnal API load, nightly ETL).

---

predictive autoscale — tự co giãn dự báo; custom metric — số đo tùy chỉnh; KPI — chỉ số hiệu suất

Azure Monitor Metrics and Diagnostics Pipeline

Metric sources:

- VMSS instance metrics (CPU, disk queue, NIC throughput).
- Application Gateway: p95 latency, HTTP 5xx rate.
- Azure Load Balancer: health probe success/fail.
- Storage queues: message count, queue age.
- App Insights: request rate, dependency failures, trace-based metrics.

Autoscale evaluation:

- Rolling window: 1–5 minutes depending on the metric.
- Statistics: Average, Minimum, and in some services Percentile.
- Combine multiple rules (e.g., CPU $> 70\%$ **AND** latency > 150ms).

diagnostics — chẩn đoán; dependency failure — lỗi phụ thuộc; health probe — thăm dò sức khoẻ

Warm Pools, Instance Preparation, and Cold-Start Control

Azure VMSS Instance Preparation:

- Use Shared Image Gallery to distribute pre-baked images globally.
- Preinstall agents, libraries, telemetry, and security baselines.
- Keep cloud-init/custom script extension lightweight.

Azure "pre-provisioning":

- VMSS supports zone-redundant pre-provisioning.
- Instances created ahead of time reduce scale-out latency.

Best practices:

- Use a SIG + version pinning to guarantee deterministic scaling.
- Validate that apps join the LB/APGW within seconds.
- Use health probe grace period to avoid false instance failures.

---

pre-provisioning — chuẩn bị trước; script extension — tiện ích thi hành; grace period — thời gian gia ân

Network, Placement, and Security in Azure Autoscaling

Networking:

- VMSS deployed across multiple Availability Zones (AZs) for HA.
- Application Gateway or Azure Load Balancer auto-register new instances.
- Health probes remove unhealthy VMs immediately.

Placement:

- **Proximity Placement Groups (PPG)** reduce latency for HPC/NFV.
- **Fault domains** ensure replicas are separated within a datacenter.
- Capacity-aware placement for Spot VMs to minimize interruptions.

Security:

- Network Security Groups (NSGs) auto-apply to new instances.
- Managed Identities (MSI) replace static credentials.
- Azure Policy enforces image pinning, region restrictions, or VM size rules.

---

fault domain — miền lỗi nội DC; PPG — nhóm vị trí gần; Managed Identity — danh tính quản lý

Design Considerations for Azure Autoscaling

1. Image Quality & SIG Versioning

- Always pin SIG image versions (avoid drifting).
- Keep images lightweight but complete; avoid heavy extensions.

2. Metric Selection

- Prefer App Insights latency, Storage Queue depth, APGW metrics.
- CPU alone is rarely sufficient.

3. Scaling Safety

- Use separate scale-out (fast) and scale-in (slow) policies.
- Add a termination protection window during scale-in.
- Use multiple rules to avoid false positives.

4. Governance & Cost Control

- Use Azure Policy to restrict regions, VM sizes, or BYO images.
- Budget alerts + Cost Analyzer for autoscaling guardrails.
- Limit Spot VM percentage for mission-critical tiers.

SIG versioning — phiên bản ảnh SIG; termination protection — chống xoá nhầm

Common Scaling Patterns by Workload Class

Interactive APIs / Microservices:

- Horizontal scaling behind load balancers; target p95/p99 latency and error budget.
- HPA-like behavior based on **RPS** and latency, with CPU as secondary guard.
- Aggressive scale-out, conservative scale-in to avoid user-visible flapping.

Batch / Data Processing:

- Queue length, message age, or job count as **primary** signals.
- Spot/preemptible capacity where failure/retry is acceptable.
- Often combine **scheduled** scale-up with metric-based scale-down.

ML Training / GPU Jobs:

- Pool of GPU workers pulling from a job queue; capacity typically **scheduled** by window.
- Strict quotas, fair-sharing, and gang-scheduling to avoid interference.
- Autoscaling is usually coarse-grained (per training wave), not continuous.

NFV / Data Plane:

- Focus on packets per second (pps), latency and jitter, loss and tail drops.
- SR-IOV, DPDK, XDP; scaling constrained by NIC, NUMA and CPU architecture.
- Preference for **scale-up within a node** before scale-out, to preserve locality.

From Benchmarks to Capacity Numbers

Step 1: Benchmark per-instance capacity at SLO

- Measure sustainable throughput C (req/s) per instance at target p95 latency.
- Use a **steady-state** load test, not just short spikes.

Step 2: Use peak demand to compute minimum instances

$$N_{\text{min}} = \left\lceil \frac{\lambda_{\text{peak}}}{C \cdot u_t \cdot s} \right\rceil$$

- λ_{peak}: peak request rate (req/s) for the design horizon.
- u_t: target utilization (e.g., 0.6–0.75 to keep headroom).
- s: safety factor (e.g., 1.1–1.4 for variance and unknowns).

Step 3: Add buffers

- Warm pool $W \approx 10\% - 20\%$ of N_{min}, rounded to an integer.
- Surge buffer $B \approx 20\%$ of N_{min} for sudden spikes and failover.

These values become the **min/max** and warm capacity for your autoscaling group, regardless of platform (OpenStack, AWS, Azure, GCP).

Cost Comparison: Static vs Autoscaled

Static provisioning:

- Size for λ_{peak} and keep that capacity always on.
- Simple and predictable; easy to combine with **reserved** capacity discounts.
- Often a large amount of idle capacity at off-peak hours.

Autoscaled provisioning:

- Capacity follows the load curve (with some lag and dampening).
- Total cost depends on traffic shape and scale-in aggressiveness.
- More operational complexity: alarms, policies, observability and guardrails.

Hybrid pattern (industry default):

- Maintain a **baseline** static capacity sized for typical load and failure scenarios.
- Use autoscaling for **bursts** above baseline and to exploit cheaper capacity (spot/preemptible).
- Combine with budgets/quotas to prevent runaway cost during incidents or DDoS.

---

reserved capacity — dung lượng đặt trước; dampening — làm dịu dao động; runaway cost — chi phí vượt kiểm soát

Week 9 Wrap-Up — Resource Allocation and Autoscaling

Theme: Architect autoscaling as a *control loop* that balances SLOs, cost, resilience, and security — across any cloud platform.

Key takeaways:

- **Resource allocation is a policy decision**, not a configuration step: sizing, placement, isolation tiers, oversubscription and quotas encode business intent and operational constraints.
- Vertical and horizontal scaling have distinct **capabilities, limits, and operational risks**; most production systems adopt a hybrid strategy.
- Autoscaling requires **trusted signals** (SLO-driven), stable evaluation (window, hysteresis, cooldown), and safe behavior during telemetry failure.
- Across clouds (OpenStack, AWS, Azure, GCP), autoscaling follows the same **control-loop pattern**: metrics → alarms → scaling policies → warm pool → LB integration.
- Benchmarks feed the sizing formulas; **buffers** (warm pool, surge) and **guardrails** (quotas, budgets, policies) ensure safety under burst or failure.

Next: Hybrid Cloud Deployment — extending placement, scaling and governance across multiple providers and regions.

Unified Mental Model — Autoscaling (1/2)

A production-grade autoscaler is defined by five pillars:

1. Signals (What to measure)

- Primary SLO: p95 latency, queue age, request rate
- Secondary guards: CPU steal, error rate, packet loss
- Probe-based verification: active health and synthetic tests

2. Evaluation (How to decide)

- Windowing, percentiles, hysteresis, cooldown
- Multiple signals must *agree* before action

3. Actuation (How to scale)

- Step or proportional scaling
- Warm pool, surge buffer, min/max capacity
- Predictable, image-pinned instance lifecycle

---

actuation — hành động; hysteresis — vùng trễ; warm pool — nhóm máy sẵn sàng

Unified Mental Model — Autoscaling (2/2)

4. Resilience (How to stay safe)

- Fail-safe scale-in freeze during telemetry gaps
- Slow scale-in, fast scale-out
- LB health checks + graceful drain

5. Governance (How to stay in budget and compliant)

- Quotas, budgets, regional policies
- Image/instance restrictions (pinned IDs, approved catalogs)
- Policy-as-code for security and cost enforcement

These pillars are *cloud-agnostic* — the same design applies to OpenStack, AWS ASG, Azure VMSS, and GCP MIGs.

---

freeze — đóng băng; governance — quản trị; LB — Load balancer; policy-as-code — chính sách dưới dạng mã