Kubernetes Efficiency & Performance

Requests/Limits • QoS • Cluster Sizing • Spot Nodes • Cost • Benchmarking

Audience: Advanced / Intermediate (beginner-accessible)

Focus: Theory-first guidance for right-sizing, stability, and cost control

Agenda

- Resource Requests & Limits (CPU, memory tuning)
- Pod Quality of Service (QoS Classes)
- Cluster Sizing & Node Pool Optimization
- Spot/Preemptible Nodes in Cloud Environments
- Cost Monitoring Tools (Kubecost, OpenCost)
- Performance Benchmarks & Load Testing

1) Resource Requests & Limits

Concepts & Scheduling

- **Request:** Guaranteed minimum for scheduling & QoS. Scheduler bins pods to nodes using requests.
- **Limit:** Hard cap (CPU throttled; memory OOM-kills if exceeded).
- **Units:** CPU in millicores (1000m = 1 vCPU); memory in bytes (e.g., Mi, Gi).
- **Golden rule:** Request ≈ typical steady usage, Limit = safe ceiling (or unset for CPU if burst is desired).

CPU vs. Memory Semantics

Resource	Exceeding Request	Exceeding Limit	Notes
CPU	Pod may burst (if limit > request or no limit)	Throttling via CFS quota	No CPU OOM; only throttling
Memory	Allowed (until node pressure)	OOMKill container	Memory is non-compressible; accurate requests matter

Tip: If latency-sensitive and bursty, consider no CPU limit (only a request) to avoid CFS throttling; use HPA to contain growth.

Tuning Guidance

- Measure first: Use Prometheus histories (p50/p95) over peak windows.
- Set requests: p95 steady-state + headroom (e.g., +20%).
- Set limits: CPU: 2–3× request or none; Memory: 1.2–1.5× request (depends on GC/language).

Language specifics:

- JVM: Align heap + metaspace + overhead with memory limit; enable cgroup-aware ergonomics.
- Go: Tune GOGC; watch RSS vs. heap; avoid tiny limits (GC thrash).
- Node/Python: Consider worker concurrency; avoid swapping.

Overcommit & Evictions

- CPU overcommit is safe (throttles under contention).
- Memory overcommit risks evictions under pressure (QoS order applies).
- Node pressure signals: MemoryPressure, PIDPressure, DiskPressure → kubelet evicts using policies.

2) Pod Quality of Service (QoS)

Classes & Rules

QoS	How to get it	Eviction Priority	Use when
Guaranteed	Every container: request == limit for CPU & memory	Highest protection	Critical/latency-sensitive
Burstable	At least one request set; not all equal to limits	Medium	Most services
BestEffort	No requests/limits	Lowest (evicted first)	Non-critical batch/ephemeral

Implication: QoS affects eviction and fairness; choose intentionally.

3) Cluster Sizing & Node Pool Optimization

Capacity Planning (Theory)

- **Inputs:** Workload mix (cpu/mem), diurnal patterns, replicas, PDBs, failure domains (AZs).
- **Targets:** Node utilization 50–70% CPU, 60–80% memory (leave headroom for spikes & bin-packing).
- **Headroom:** Per node & per cluster; consider “N+1” or “AZ-1” failure scenarios.


Demand Forecast
Bin Packing Requests
Headroom spikes/HA
Node Count per Pool
Cluster Autoscaler Policy

- Right-size nodes: Match pod shapes to instance shapes (avoid fragmenting memory).
- Pools: Separate by workload traits (CPU-optimized, memory-optimized, GPU, spot); use taints/tolerations & affinity.

Autoscaling Interactions

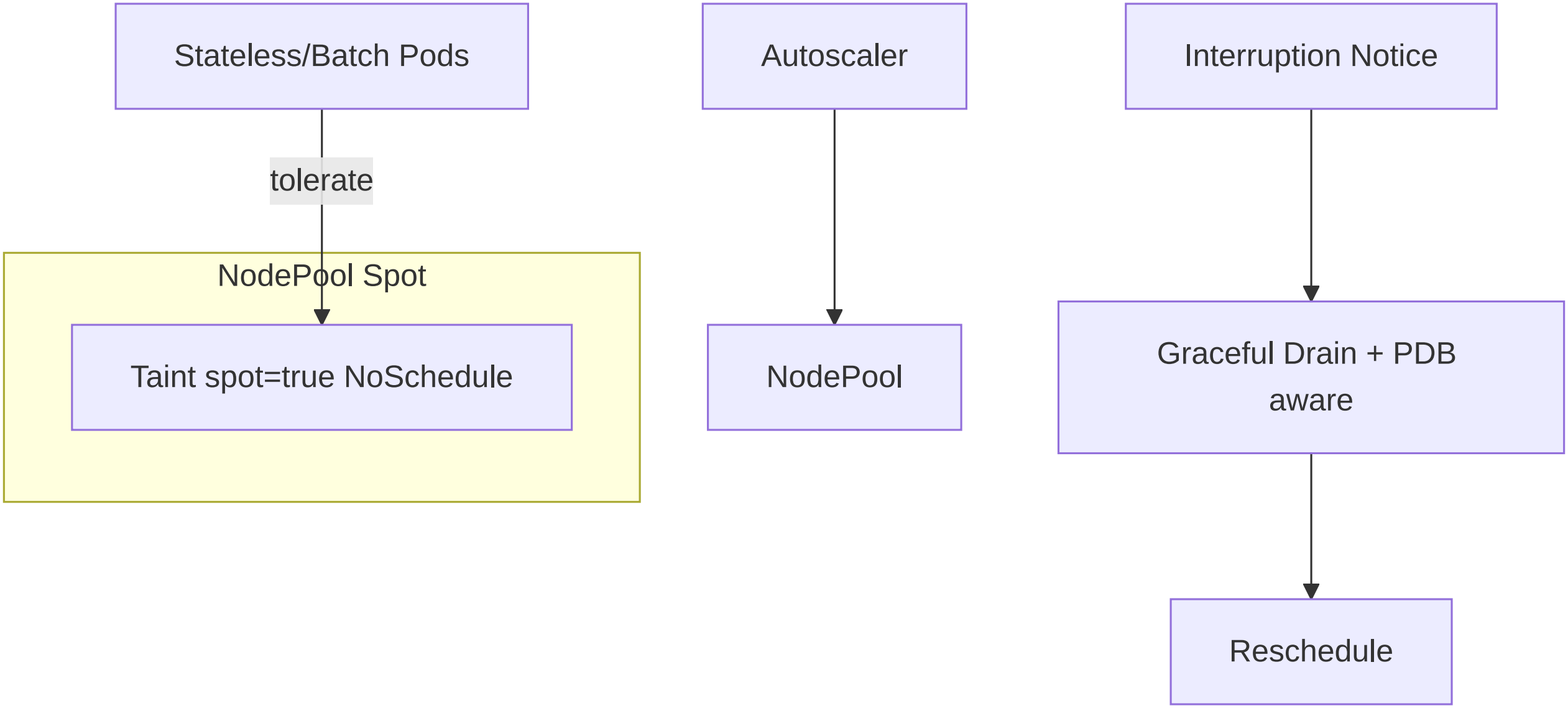
- HPA: Scales pods on metrics (CPU%, custom).
- VPA: Adjusts requests (recommend or enforce; careful with frequent restarts).
- Cluster Autoscaler (CA): Adds/removes nodes based on unschedulable pods.

Best practice: HPA + CA for breadth; VPA in recommend mode feeding SRE/SLO reviews or off-hours rollouts.

4) Spot / Preemptible Nodes

Benefits, Risks, Controls

- **Pros:** 60–90% cost savings; great for tolerant workloads.
- **Cons:** Short notice termination (30–120s), capacity volatility, noisy migrations.


Stateless/Batch Pods
tolerate
NodePool Spot
Taint spot=true NoSchedule
Autoscaler
NodePool
Interruption Notice
Graceful Drain + PDB aware
Reschedule

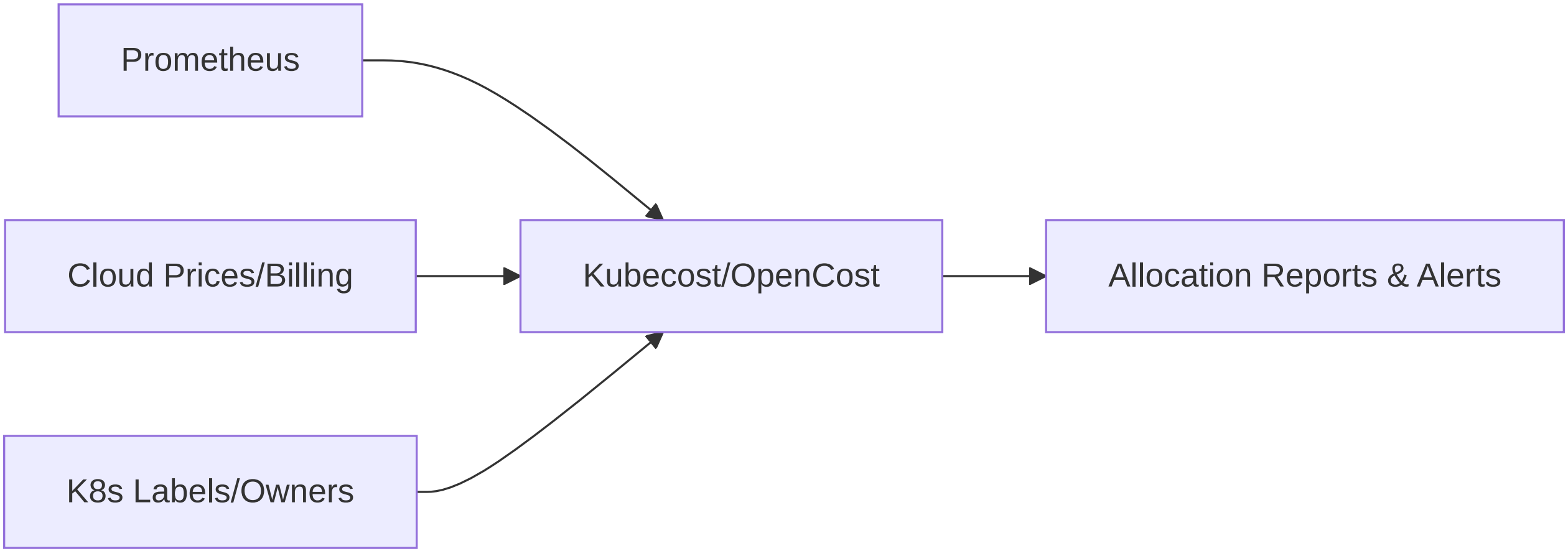
Design:

- Isolate in tainted pools; only tolerant pods schedule there.
- PDBs protect availability; ensure multiple replicas across pools/AZs.
- Use interruption handlers to pre-drain and checkpoint work.
- Keep one on-demand pool for baseline capacity.

5) Cost Monitoring (Kubecost / OpenCost)

Allocation & Accountability

- **Model:** Attribute spend to namespace, deployment, label, team, including shared costs (LBs, egress, storage).
- **Data sources:** Prometheus metrics (CPU/mem usage & requests), cloud pricing APIs/billing exports, Kubernetes metadata.


Prometheus
Cloud Prices/Billing
K8s Labels/Owners
Kubecost/OpenCost
Allocation Reports & Alerts

Practices:

- Set owner labels (e.g., team, cost-center).
- Watch request-based cost (over-requested memory dominates).
- Enforce via policies (max request/limit ratios) and SLO-driven rightsizing.

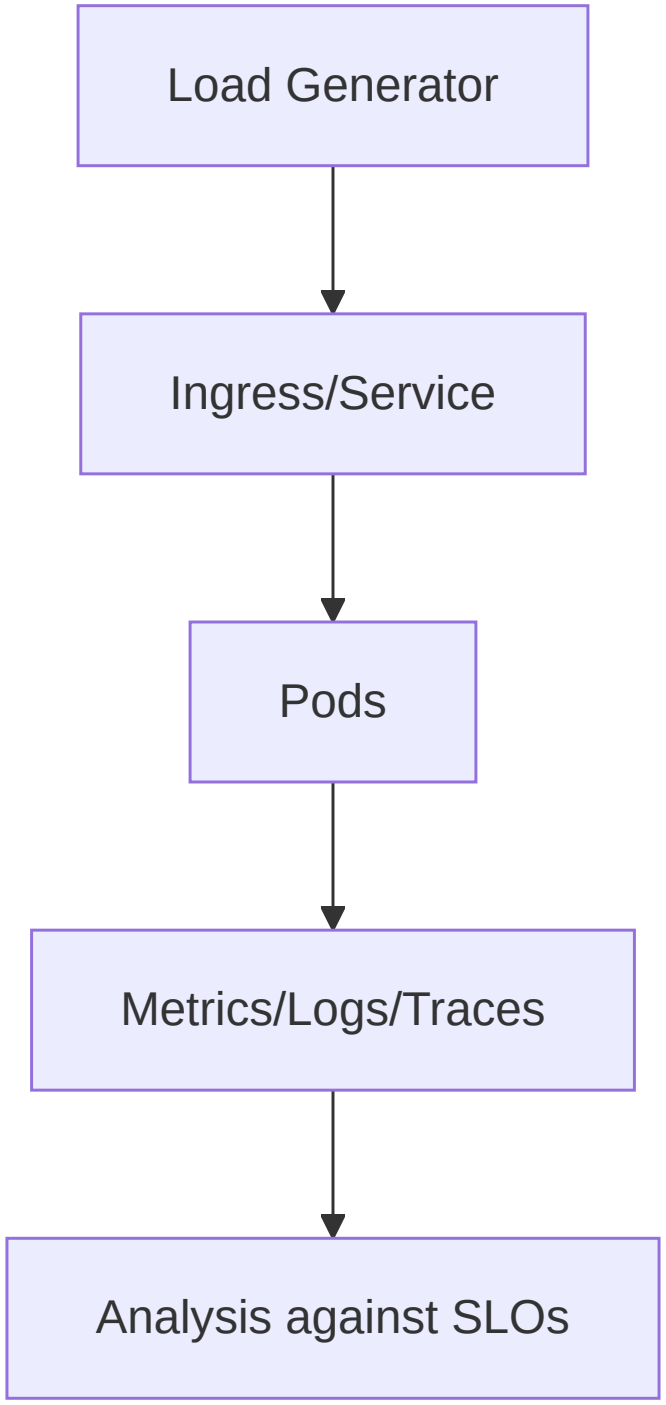
6) Performance Benchmarks & Load Testing

Method & Metrics

- **Define SLOs first:** e.g., p95 latency ≤ 200 ms, error rate < 0.1%.
- **Workload model:** Open vs. closed, arrival rates, burst patterns, think times.
- **Key metrics:** p50/p95/p99 latency, throughput (RPS), saturation (CPU%, run queue), GC/heap, throttling, OOMs.

Tooling & Environments

- **Load tools:** k6, Locust, Vegeta, Fortio (HTTP); wrk, JMH (microbench).
- **System profilers:** pprof, eBPF tools, perf, JFR for JVM.
- **K8s-specific:** kube-burner/clusterloader2 (cluster-scale), kubectl top/metrics-server for quick checks.
- **Env parity:** Mirror production node types, CNI, TLS, and limits; disable CDN caches if measuring backend.


Load Generator
Ingress/Service
Pods
Metrics/Logs/Traces
Analysis against SLOs

Tuning Loop (Theory)

- Profile under load → find bottleneck (CPU, GC, I/O, DB).
- Adjust requests/limits; remove CPU limits if throttling drives latency.
- Scale replicas (HPA) and/or nodes (CA).
- Optimize code/config (connection pools, keep-alive, GC).
- Re-test; compare against SLOs; iterate.

Patterns & Anti-Patterns (Summary)

Do:

- Size requests from data; keep QoS aligned with criticality.
- Separate node pools by workload; isolate spot capacity.
- Use HPA + CA, VPA in recommend mode, and regular rightsizing reviews.
- Track cost per label/namespace; set owner tags.

Avoid:

- Memory limits << realistic peak (causes OOMKill).
- High CPU limits with tiny requests (noisy neighbor risk).
- Scheduling all workloads on spot; mixing critical + spot without taints.
- Benchmarking on non-prod-like clusters.

References & Further Reading

- **Kubernetes:** Resource QoS, Eviction Policies, HPA/VPA/CA docs
- **OpenCost / Kubecost:** allocation methodologies
- **SRE Workbook:** SLOs, alerting on burn rates
- **Benchmarking:** k6/Locust/Vegeta, clusterloader2/kube-burner