Hosting Containers Across Different Cloud Providers

Audience: Senior platform engineers and SREs with working knowledge of Kubernetes, cloud networking, and CI/CD. **Job ID:** 9271fbc8-750d-4f8d-b396-48eca9507403 **Generated:** 2026-02-05

Table of Contents

1.1 Scope, Non-Goals, and Reader Contract (s01)

1.1.1 Objectives

- Define what “multi-cloud containers” means in this document, using consistent **Kubernetes cluster control plane** vs **platform management plane** terminology.
- Set explicit assumptions and non-goals so you don’t accidentally sign up for an unbounded platform program.
- Establish decision drivers and success criteria anchored to **SOC2**, **EU/US data residency**, **encryption everywhere**, and **99.9% availability**.

1.1.2 When to use / avoid

Use this guide when you:

- Need **two or more providers** for sovereignty, resilience, M&A, or regulated customer commitments.
- Want Kubernetes-based portability while accepting that **operations—not YAML—are the hard part**.

- Need auditability (SOC2), strong tenant/team boundaries, and repeatable controls across providers/regions.

Avoid (or narrow scope) when you:

- Only need **multi-region in one provider**; you'll usually get better operability and lower cost without cross-provider complexity.
- Have **stateful, latency-sensitive** workloads that can't tolerate cross-cloud network variance; you'll spend your error budget on physics.
- Don't have staffing for on-call, incident response, and security operations across providers; multi-cloud amplifies toil.

1.1.3 Architecture notes

1.1.3.1 Definitions (what "multi-cloud containers" means here)

- **Multi-cloud**: Your **data plane** runs workloads in **Kubernetes clusters** across **multiple providers**, with a consistent operating model (GitOps, policy, observability) and defined tenant/team boundaries.
- **Hybrid**: Your data plane spans cloud plus on-prem/edge. This document may reference patterns that apply, but hybrid adds hardware lifecycle, facility risk, and different failure modes.
- **Multi-region**: Multiple regions within one provider. Often a stepping stone and frequently the right answer when sovereignty constraints permit.

1.1.3.2 Control plane terminology (avoid ambiguity)

This document uses two related but different "control plane" concepts:

- **Kubernetes cluster control plane**: Kubernetes API server, scheduler, controller managers, etcd (managed or self-managed).
- **Platform management plane (a.k.a. management/control plane in this guide)**: the systems that manage clusters and enforce desired state across them (GitOps controllers, policy distribution, identity federation, artifact promotion, observability backends).

When you define SLOs and incident ownership, name which plane you mean.

1.1.3.3 Tenant/team boundary terminology

- **Team boundary**: internal engineering ownership boundary (on-call rotation, deployment rights, cost center).

- **Tenant boundary**: customer/regulatory isolation boundary (data classification, residency constraints, contractual controls).

Either boundary can map to different primitives depending on risk tier: provider **account/project**, **region**, **cluster**, **namespace**, and **network segmentation**.

1.1.3.4 Workload categories and fit

Workload type	Multi-cloud suitability	Primary constraint	Typical placement guidance
Stateless services	High	Traffic steering + identity	Active/active or active/passive across providers if dependencies allow
Stateful services	Medium/Low	Data consistency + replication + RPO/RTO	Prefer single provider with multi-region; multi-cloud only with explicit DR design
Batch/async	Medium	Data locality + cost variability	Run where data resides; use artifact promotion and policy checks
Edge	Variable	Network + upgradeability	Treat edge as separate clusters with strict tenant/team boundaries and limited blast radius

1.1.3.5 Success criteria (measurable outcomes)

- **Availability**: Meet **99.9% service availability** per defined SLO.
    - You explicitly define what is **in-scope** (e.g., data plane request success + latency, global traffic manager/DNS, ingress/gateway) and what is **out-of-scope** (commonly: CI availability, non-serving dev clusters), and you document all required dependencies.
    - You decide whether managed **Kubernetes cluster control plane** availability is part of the SLO or a separate platform SLO.
- **RTO/RPO**: Define per service tier; don't claim multi-cloud "resilience" without tested DR runbooks.
- **Portability**: Portability is *interface-level*: Kubernetes API, OCI artifacts, GitOps workflows, OPA policies—not "run anywhere with zero changes."

- **Auditability**: Every change to the platform management plane, Kubernetes resources, and workload configuration is attributable (who/what/when), logged, and reviewable.

1.1.4 Assumptions

- You operate at least **two regions** overall and at least **one region per provider** you deploy to.
- You can enforce **encryption in transit** (mTLS/TLS) and **encryption at rest** (provider KMS/HSM-backed where available) for all in-scope systems.
- You have centralized **audit logging** for cloud IAM events, Kubernetes API events, and GitOps reconciler actions, with retention aligned to SOC2 evidence needs.
- You can implement **least privilege** across IAM and Kubernetes RBAC, and you can rotate/expire credentials and certificates on a defined cadence.
- Teams accept that “one cluster per team” is usually the wrong default; you choose isolation boundaries intentionally (namespaces/projects/accounts first, clusters only when justified by tenant risk, residency, or blast radius).
- **Residency is a hard boundary**: for EU/US separation you may need duplicated stacks (artifact registries, observability backends, CI runners, backup targets) to avoid “global” systems becoming cross-boundary data movers.

1.1.5 Non-goals

- No promise of “write once, run anywhere” without provider-specific integration work (networking, IAM, storage classes, load balancing).
- This guide does not include full application rewrites or data model refactors; however, you may still need service changes to meet explicit RPO/RTO and residency tiers.
- No assumption that a service mesh is mandatory; it’s optional and comes with operational cost.
- No attempt to standardize away provider outages; you design to **degrade and recover**, not to be outage-proof.

1.1.6 Risks and mitigations (major, recurring failure modes)

Risk / failure mode	What it looks like in production	Mitigation you should adopt
	Multi-cloud clusters exist, but one hosted	Classify dependencies; require DR strategy per

Risk / failure mode	**What it looks like in production**	**Mitigation you should adopt**
Hidden coupling to a single provider service	DB/queue makes failover non-viable	dependency; test RTO/RPO with game days
Policy drift across clusters/ providers	A “secure baseline” exists only on paper	GitOps for cluster config; OPA admission policies; periodic conformance checks; audit logging
Identity sprawl and long-lived credentials	Break-glass keys become permanent	Workload identity patterns; rotation/expiry SLAs; least privilege reviews; incident drills
Cross-cloud network variance	Latency spikes, intermittent packet loss, MTU issues	Prefer locality; isolate chatty services; explicit SLOs; aggressive timeouts/circuit breaking
Operational overload	Too many clusters, inconsistent runbooks	Standardize cluster lifecycle; limit supported variants; invest in day-2 automation and observability

1.1.7 Operational considerations

- **Kubernetes cluster control plane vs platform management plane**: incidents often start as data plane symptoms but root-cause in platform management plane issues (policy distribution, identity, networking, registry access) or Kubernetes control plane health.
- **Provider variance is inevitable**: normalize interfaces (Kubernetes, OCI, GitOps), but keep a documented “provider adapter” layer (networking, IAM, storage) and budget time for it.
- **Data residency enforcement**: EU/US residency is not a label—it’s constraints on where data is stored, processed, and logged. Your observability pipelines and backups are common violations.
- **Encryption everywhere**: don’t rely on “private networks” as a control. Require TLS/mTLS for service-to-service, and encrypt backups and snapshots with managed keys and explicit rotation.

1.1.8 Checklist

- [ ] You have written definitions for **multi-cloud**, **hybrid**, and **multi-region** as used by your org.
- [ ] You have adopted unambiguous terminology for **Kubernetes cluster control plane**, **platform management plane**, and **data plane**.
- [ ] You have a service tiering model with **SLOs**, **RTO**, and **RPO** (per service, per dependency).
- [ ] You have documented **Assumptions** and **Non-goals** and can point to them in review cycles.
- [ ] You have an agreed baseline for **SOC2 evidence**: audit logs, change approvals, access reviews, retention.
- [ ] You have a documented approach for **EU/US data residency**, including backups, logs, CI/CD execution, and artifact storage.
- [ ] You have defined tenant/team boundaries and least-privilege access models for management plane changes and workload changes.
- [ ] You have identified the top three expected failure modes and mapped them to mitigations and test plans.

1.2 Design Principles and Key Tradeoffs (s02)

1.2.1 Objectives

- Give you a small set of principles that produce **predictable platform behavior** across cluster/region/provider boundaries.
- Make tradeoffs **explicit, testable, and measurable** (SLOs, RTO/RPO, error budget, cost).
- Keep the platform **SOC2-friendly**: auditability (GitOps), least privilege, and evidence-ready controls.
- Support **EU/US data residency** and **encryption everywhere** while targeting **99.9% availability**.

1.2.2 When to use / avoid

Situation	Use these principles when…	Avoid (or constrain scope) when…
Multi-region (single provider)	You need better availability/ sovereignty with lower variance than multi-provider	You can meet SLO and residency with one region and strong DR

Situation	Use these principles when…	Avoid (or constrain scope) when…
Multi-provider	Provider risk reduction and/or customer contracts require it	You're doing it primarily for "portability" without 24/7 ops maturity
Workload profile	You're mostly stateless or can accept asynchronous DR patterns	You're deeply coupled to provider-native stateful services without a clear DR story

1.2.3 Assumptions

- You must meet SOC2 expectations (access control, audit logging, change management, incident response evidence).
- You must support **EU/US residency separation** (data + telemetry + backups).
- You enforce **encryption in transit and at rest**, with key rotation/expiry.
- You target **99.9% service availability** and can define/measure SLOs.

1.2.4 Non-goals

- Not a "run anywhere with zero variance" claim—provider edges exist and must be documented.
- Not a refactor guide for application architecture (beyond portability constraints you enforce).
- Not a full hybrid/on-prem operating model (hardware lifecycle and physical networking are materially different).

1.2.5 Architecture notes (principles)

1.2.5.1 1) Standardize interfaces; isolate provider specifics behind documented seams

You want portability at the **interface** level, not at the "everything is identical" level.

- Prefer CNCF/Kubernetes primitives: Kubernetes APIs, Gateway API, CNI, OPA, GitOps controllers, OCI artifacts.
- Put provider-specific implementations behind **explicit adapters** and write them down as contracts (use ADRs).

- Keep the **platform management plane** (identity, GitOps, policy, CI metadata, observability backends) operationally independent of any single cluster's **data plane** to avoid circular dependencies during outages.

Tradeoff: abstraction reduces blast radius of provider variance, but can hide critical constraints (quotas, LB behavior, IAM edge cases). Counter with ADRs + conformance tests per provider.

1.2.5.2 2) Prefer declarative APIs and convergent reconciliation

If it can't converge from desired state, it won't be reliable at 03:00.

- Use GitOps reconciliation for Kubernetes resources and policy bundles.
- Treat imperative "hotfixes" as incidents: allow them, but require post-facto reconciliation into Git with audit trail.

Failure mode: drift (manual changes) becomes your de facto "source of truth." Mitigation: restrict write access, require PR-based change control, and alert on drift.

1.2.5.3 3) Design for blast-radius containment as a first-class requirement

Containment boundaries should map to how you respond to incidents and how you produce SOC2 evidence.

Recommended hierarchy (largest to smallest blast radius):

- Provider org/account/subscription/project boundary
- **Environment boundary** (prod vs non-prod, and regulated vs unregulated where applicable)
- Region boundary
- Cluster boundary
- Namespace boundary (tenant/team boundary)
- Network segment boundary (east-west controls)

Tradeoff: more boundaries reduce impact but increase operational overhead (more IAM, more pipelines, more observability wiring). Start with the smallest number that meets residency + availability targets.

1.2.5.4 4) Decide consistency vs availability per system boundary (don't default)

In multi-region/multi-provider, you choose where you want to sit on CAP tradeoffs.

In this guide, a **system boundary** is the boundary of a request path plus its dependency graph (e.g., API + database + queue) across region/provider.

- For “control” systems (policy, audit logs, CI metadata): prefer consistency and strong auditability.
- For “serving” paths (stateless APIs): prefer availability with clear degraded modes.
- For stateful data: choose explicitly per tier with documented RPO/RTO and tested DR.

Failure mode: accidental strong consistency across regions/providers via synchronous writes causes latency spikes and cascading timeouts. Mitigation: keep cross-region/provider dependencies asynchronous unless you can prove latency/error budget.

1.2.5.5 5) Use a day-0 / day-1 / day-2 maturity model

You should not treat multi-cloud as a day-0 architecture problem only; most outages are day-2.

- **Day-0:** baseline reference architecture, tenancy model, IAM model, encryption, network topology, GitOps bootstrap.
- **Day-1:** onboarding workflows, golden paths, dashboards/alerts, policy guardrails.
- **Day-2:** patching, certificate rotation, incident response, game days, DR tests, cost controls.

Tradeoff: investing early in day-2 automation slows initial delivery but is cheaper than scaling human toil across multiple providers.

1.2.6 Operational considerations

1.2.6.1 Evidence and auditability (SOC2)

- Enforce PR-based change control for cluster and platform config (GitOps).
- Centralize audit logs per provider/account **within each residency boundary** (EU-centralized and US-centralized, not globally centralized if that crosses boundaries).
- Retain logs per policy; ensure immutability/append-only properties where feasible.

1.2.6.2 Least privilege and identity hygiene

- Use workload identity patterns (no long-lived static cloud keys in pods).
- Scope IAM roles to tenant/team boundaries; use separate roles per environment and per cluster.
- Rotate credentials/certificates with defined expiry and alerting on near-expiry.

1.2.6.3 Availability and DR posture

- 99.9% availability implies you must define:
    - SLOs/SLIs per service boundary
    - Error budgets and what you do when you burn them
    - RTO/RPO tiers, with rehearsal schedules
- Keep dependencies (registries, policy bundles, DNS/traffic steering) resilient to single-cluster failures.

1.2.6.4 Residency enforcement

- Region/provider allowlists for telemetry, backups, and artifact replication.
- Residency-scoped keys (separate KMS keyrings/tenants) and explicit “no cross-residency” policy checks.

1.2.7 Risks and mitigations (major design-level)

Risk	What it looks like in production	Mitigation you should implement
More failure modes	Partial outages, inconsistent behavior across providers	Game days, conformance tests, SLO-based gating for rollouts
Residency violations	Telemetry/backup/artifacts silently crossing EU/US	Region-scoped pipelines, allowlists, residency-scoped keys, policy-as-code checks
Hidden provider coupling	Storage/LB/IAM assumptions break failover	ADR-backed dependency inventory, defined fallback patterns, periodic “provider switch” exercises
Operational overload	Too many clusters, too many knobs	Fewer clusters, start multi-region before multi-provider, invest in

Risk	What it looks like in production	Mitigation you should implement
		automation and standard interfaces

1.2.8 Checklist

- [ ] You have written ADRs for every provider-specific dependency (IAM, networking, storage, LB, DNS/traffic steering).
- [ ] You can state SLOs, error budgets, and RTO/RPO for each service tier—and you test them.
- [ ] GitOps is the default for platform and cluster configuration; drift detection is enabled.
- [ ] Tenant/team boundaries map to namespaces + IAM roles + network policies (least privilege).
- [ ] Audit logging is enabled, centralized **within** each residency boundary, and retention is defined.
- [ ] Encryption at rest/in transit is enforced; key and certificate rotation/expiry is automated and monitored.
- [ ] Multi-region is your default stepping stone; multi-provider is justified by residency/risk/availability requirements, not portability rhetoric.

1.3 Target Reference Architecture (TRA) Overview (s03)

1.3.1 Objectives

- Provide a canonical **multi-cloud container platform** reference architecture that cleanly separates the **platform management plane** from per-cluster **data plane** concerns.
- Make **tenant/team boundaries** and **blast radius** boundaries explicit (provider/account/project, region, cluster, namespace).
- Establish a baseline that can meet **SOC2**, **EU/US data residency**, **encryption everywhere**, and **99.9% availability** requirements.

1.3.2 When to use/avoid

Use when

- You have hard requirements for **EU/US residency separation** (including telemetry and backups), or regulated customers need evidence of strong isolation.
- You need **provider risk reduction** (ability to fail over across providers) and can afford the operational overhead.
- You can commit to GitOps/policy-as-code and regular DR testing (game days).

Avoid when

- Your primary motivation is “portability.” Kubernetes does not remove coupling to IAM, L4/L7 load balancing, DNS, managed storage/DB, KMS, and observability.
- You do not have 24/7 operations coverage or automation maturity; multi-cloud increases failure modes and time-to-diagnose.
- The portfolio is predominantly **stateful** with tight coupling to provider-native storage and databases; multi-provider DR is often expensive and operationally brittle.

1.3.3 Architecture notes

1.3.3.1 Assumptions

- You enforce **EU/US data residency** as a hard boundary for: customer data, logs/metrics/traces, backups, and artifact storage/replication.
- You enforce **encryption in transit and at rest** with KMS-backed keys and **rotation/expiry**.
- You maintain **least-privilege** IAM, and keep **audit logging** enabled and retained per compliance needs.
- Your availability target is **99.9%**; you explicitly define SLOs/SLIs and manage error budgets.

1.3.3.2 Non-goals

- This is not a “run anywhere with zero variance” design; provider-specific edges exist and must be documented behind stable interfaces (tracked via ADRs).

- This is not a hybrid (on-prem + cloud) reference; physical networking and hardware lifecycle change the model materially.
- This is not a migration guide; it is a target architecture you can incrementally adopt.

1.3.3.3 Reference architecture: boundaries and responsibilities

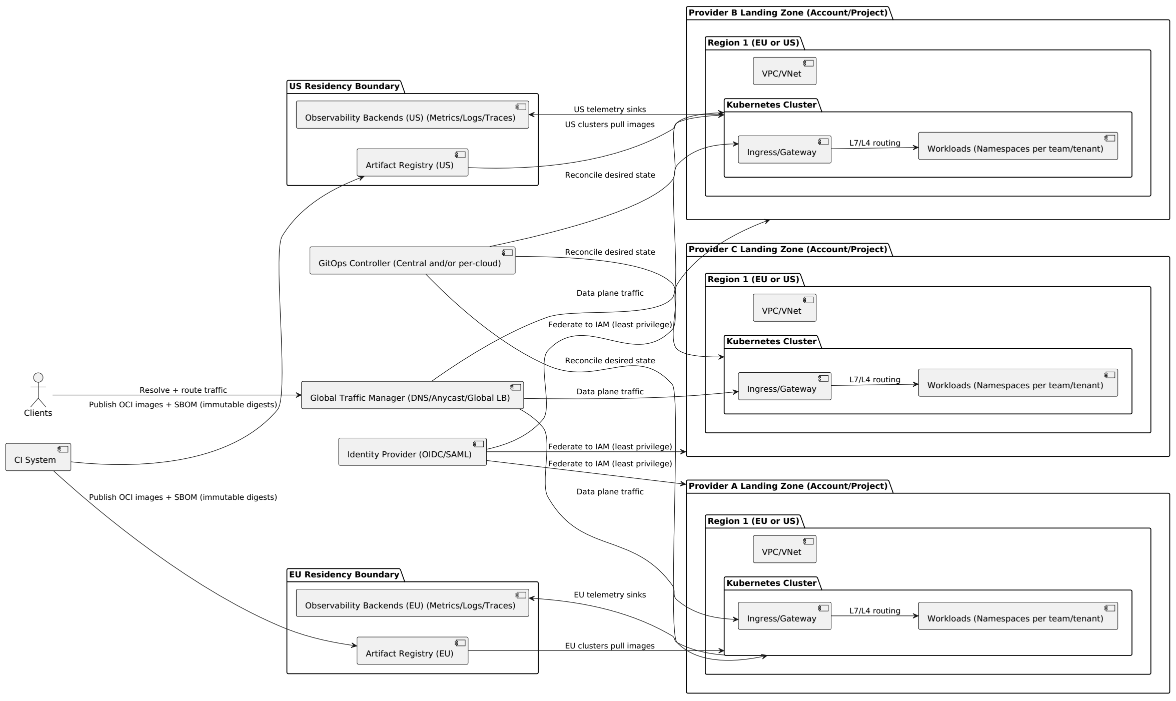
In a multi-cloud platform:

- Treat the **platform management plane** as the systems that define and enforce desired state across clusters (identity federation, policy, GitOps, artifact promotion, observability backends, and management workflows).
- Treat the **data plane** as the per-cluster runtime that serves traffic (nodes, pods, CNI, ingress/gateways, and service networking).
- Treat the **Kubernetes cluster control plane** as the per-cluster API and controllers you must keep healthy to schedule and manage workloads.

Key boundary: the **platform management plane must remain operationally independent from any single cluster's data plane**. If one cluster is down, you should still be able to audit, roll back, and promote changes safely.

1.3.3.4 Diagram: TRA overview (diagram_id: tra-overview)

Alt text: Reference architecture for multi-cloud Kubernetes showing residency-scoped (EU vs US) platform management plane services and per-cloud clusters pulling from the correct residency endpoints.


Provider B Landing Zone (Account/Project)
Region 1 (EU or US)
VPC/VNet
Kubernetes Cluster
Ingress/Gateway
L7/L4 routing
Workloads (Namespaces per team/tenant)
US Residency Boundary
Observability Backends (US) (Metrics/Logs/Traces)
Artifact Registry (US)
US telemetry sinks
US clusters pull images
Reconcile desired state
Provider C Landing Zone (Account/Project)
Region 1 (EU or US)
VPC/VNet
Kubernetes Cluster
Ingress/Gateway
L7/L4 routing
Workloads (Namespaces per team/tenant)
GitOps Controller (Central and/or per-cloud)
Reconcile desired state
Data plane traffic
Federate to IAM (least privilege)
Reconcile desired state
Clients
Resolve + route traffic
Publish OCI images + SBOM (immutable digests)
Global Traffic Manager (DNS/Anycast/Global LB)
Data plane traffic
CI System
Identity Provider (OIDC/SAML)
Federate to IAM (least privilege)
Federate to IAM (least privilege)
Publish OCI images + SBOM (immutable digests)
Data plane traffic
Provider A Landing Zone (Account/Project)
Region 1 (EU or US)
VPC/VNet
Kubernetes Cluster
Ingress/Gateway
L7/L4 routing
Workloads (Namespaces per team/tenant)
EU Residency Boundary
Observability Backends (EU) (Metrics/Logs/Traces)
Artifact Registry (EU)
EU telemetry sinks
EU clusters pull images

1.3.3.5 Cluster topology options (tradeoffs you should acknowledge)

Topology	What you get	What you pay	Common failure modes
Per-provider, per-region clusters (default)	Clear residency boundaries, smaller blast radius, simpler DR per region/provider	More clusters to operate; more upgrade windows	Version skew across clusters; inconsistent policy rollout if GitOps/ policy gating is weak
Workload-aligned clusters (by domain/tier)	Strong isolation for high-risk tenants or regulated workloads	Highest operational overhead; more networking/ egress complexity	“Snowflake clusters” and drift; teams bypass platform controls
Fewer larger clusters (per provider/ region)	Better utilization, fewer upgrades	Larger blast radius, noisier multi-tenant ops	Noisy neighbor, RBAC mistakes become high impact, more disruptive incidents

Opinionated default: start with **one cluster per region per provider** for critical workloads, and only add workload-aligned clusters when you can justify the isolation with a residency/regulatory or blast-radius argument.

1.3.3.6 Shared vs per-cluster services (platform management plane vs data plane)

Capability	Prefer shared (platform management plane) when	Prefer per-cluster when	Residency implications
Identity provider + IAM federation	You need centralized joiner/mover/leaver controls and audit	Rare; typically only for disconnected environments	Federation must respect EU/US separation where identity attributes are sensitive
GitOps management	You want consistent rollout, audit history, policy gates	You need provider-resilient operations (per-cloud GitOps)	Repos can be global, but secrets, deploy targets, and automation execution must be residency-scoped
Artifact registry	You need artifact promotion, signatures, replication control	You have disconnected regions/ providers	Replicate artifacts **within** EU and **within** US; avoid cross-boundary replication for restricted workloads
Observability backends	You need uniform SLOs, incident response, forensics	You must keep telemetry strictly in-region/ provider	Telemetry is frequently a residency violation vector; default to EU and US separate backends

1.3.4 Operational considerations

1.3.4.1 Risks and mitigations (major designs)

- **Risk: Residency violations via telemetry, backups, and artifact replication.**
  Mitigations: residency-scoped observability backends and registries; egress allowlists/private endpoints; CI/CD policy checks; residency-scoped KMS keys; explicit data classification labels; periodic audits of sink endpoints.
- **Risk: Hidden provider coupling (IAM, ingress/LB behavior, storage semantics).**
  Mitigations: ADR-backed dependency inventory; stable platform interfaces; portability tests; documented fallbacks (including "degrade mode" behaviors).
- **Risk: GitOps/controller blast radius.**
  Mitigations: progressive delivery for platform changes; environment and residency separation; mandatory policy checks (OPA) before merge and at admission; break-glass access with audited approval and expiry.
- **Risk: Operational overload (too many clusters, too much drift).**
  Mitigations: start multi-region before multi-provider; limit supported variants; automate upgrades; define SLOs/error budgets; run DR drills and standard incident playbooks.

1.3.4.2 Security and compliance baseline you should enforce

- **Least privilege:** minimize IAM roles and Kubernetes RBAC; prefer workload identity (no long-lived secrets); scope by team/tenant boundary.
- **Audit logging:** enable provider audit logs, Kubernetes audit logs, GitOps change logs; centralize storage *within the correct residency boundary*; define retention and access controls.
- **Rotation/expiry:** rotate KMS keys, workload credentials, and TLS certs; enforce short-lived tokens; alert on failed rotations and near-expiry.

1.3.4.3 Availability alignment (99.9%)

- Design for **single cluster** and **single region** failures without cascading into platform management plane outages.
- Use global traffic management to shift traffic across regions/providers, but explicitly test failover and understand that **DNS-based failover has slower convergence** than Anycast/LB approaches.

- Maintain “degrade safely” modes (read-only, limited functionality) to preserve availability when stateful dependencies fail.

1.3.4.4 Global traffic management (residency and logging caveat)

“Global” edge services (DNS, Anycast, global L7) frequently generate logs, analytics events, and WAF telemetry. Treat them as part of your residency design:

- choose configurations that allow EU vs US separation where required, or
- treat edge telemetry as regulated data and route/store it within the correct boundary.

1.3.5 Checklist

- [ ] You have documented **platform management plane vs Kubernetes cluster control plane vs data plane** ownership and failure boundaries (provider/region/cluster/namespace).
- [ ] EU and US residency boundaries are enforced for **artifacts, telemetry, backups, and keys** (not just customer databases).
- [ ] Clusters are pinned to residency-scoped endpoints via **policy + network controls** (admission/CI checks plus egress allowlists/private endpoints).
- [ ] GitOps is the source of truth, with auditable change history and defined break-glass paths (approval + expiry).
- [ ] Artifact promotion uses immutable digests and includes SBOM/signing controls.
- [ ] Observability backends are residency-scoped; egress allowlists prevent accidental cross-boundary data movement.
- [ ] You have defined SLOs/SLIs for 99.9% and runbooks for region/provider failover with tested RTO/RPO.

1.4 Compute: Kubernetes Distributions and Cluster Lifecycle (s04)

1.4.1 Objectives

- Help you choose a Kubernetes distribution per provider/region that meets **SOC2**, **EU/US data residency**, **encryption everywhere**, and **99.9% availability** targets.
- Define a **cluster lifecycle** approach that is auditable (GitOps), repeatable, and resilient to provider/region failures.

- Standardize **upgrade orchestration**, **node OS strategy**, and **autoscaling** patterns with explicit tradeoffs and rollback paths.

1.4.2 When to use / avoid

Use when

- You operate multiple clusters across **provider/region** boundaries and need consistent controls: IAM, policy-as-code, audit logging, encryption, and artifact promotion.
- You need a predictable model for Day-0/Day-1/Day-2 ops: provisioning, upgrades, scaling, and decommissioning.
- You must demonstrate auditability for SOC2 (change history, approvals, and reproducible builds).

Avoid when

- You can’t staff on-call and incident response for multi-cluster operations (multi-cloud multiplies failure modes and MTTR).
- Most workloads are stateful and tightly coupled to provider-native services without a tested DR plan; you’ll likely miss 99.9% due to operational fragility, not Kubernetes itself.

1.4.3 Architecture notes

1.4.3.1 Assumptions

- You have (or will build) a **management cluster** per residency domain (at minimum: one for EU, one for US) to avoid cross-residency control-plane data leakage (e.g., logs, cluster specs, secrets metadata).
- You use GitOps reconciliation for cluster and platform components, with environment separation and immutable artifact promotion.
- You have a defined tenant model (team/namespace boundaries) and an IAM strategy for workload identity per provider.

1.4.3.2 Non-goals

- Designing hybrid/on-prem-first Kubernetes as the default (covered only where it impacts lifecycle patterns).
- Eliminating provider coupling: Kubernetes reduces, but does not remove, dependencies on IAM, LBs, DNS, KMS, managed storage, and observability backends.

1.4.3.3 Distribution choices (managed vs self-managed)

Managed Kubernetes is the default recommendation for most teams targeting 99.9% because it reduces control plane toil and shifts some failure modes to the provider. Self-managed clusters are justified when you need strict control over versions, network plugins, or bare-metal performance, but you must staff that expertise.

Dimension	Managed Kubernetes (per provider)	Self-managed Kubernetes (IaaS or on-prem)	Operational tradeoff / failure mode
Control plane operations	Provider-operated	You operate	Self-managed increases risk of control-plane outages and slow incident response.
Upgrades	Often guided/ guardrailed	Fully your responsibility	Managed may constrain timing; self-managed can drift and accumulate CVEs.
Feature parity	Varies by provider	You choose components	“Same YAML everywhere” breaks on IAM/LB/storage differences; plan explicit abstractions.
SOC2 evidence	Easier for control plane logs/controls if integrated	More artifacts to produce	Self-managed requires strong audit logging, access reviews, and change control.
Data residency	Region selection + residency controls	You must enforce everywhere	Common failure: telemetry/backups/ artifacts escaping residency boundary.
Availability	Typically strong for control plane	Varies by your design	Managed doesn’t guarantee app availability; you still

Dimension	Managed Kubernetes (per provider)	Self-managed Kubernetes (IaaS or on-prem)	Operational tradeoff / failure mode
			need multi-AZ nodes and tested upgrades.

1.4.3.4 Node OS strategy

You should standardize on one of these patterns per provider/region, not per team.

Option	When it fits	Pros	Cons / failure modes
Container-optimized OS	High scale, minimal host drift	Smaller attack surface, easier patching	Debugging can be harder; kernel/module constraints may block niche workloads.
General-purpose Linux	Mixed workloads, legacy agents	Familiar tooling, broad compatibility	Bigger attack surface; configuration drift; inconsistent hardening across teams.

Minimum baseline you should enforce regardless of OS:

- Disk and network encryption (node volumes, in-transit where possible).
- Least privilege for node IAM; no broad cloud permissions on instance profiles.
- Audit logging shipped to residency-scoped sinks with retention policies.

1.4.3.5 Cluster lifecycle management options (CAPI, Terraform, Crossplane)

Declarative reconciliation is preferred for multi-cloud because you get convergence, drift detection, and an audit trail. The key design choice is **where** reconciliation runs and how you isolate tenants.

Tooling	Best for	Strengths	Tradeoffs / failure modes
Cluster API (CAPI)		Kubernetes-native lifecycle controllers; supports upgrade	Management cluster becomes critical dependency;

Tooling	Best for	Strengths	Tradeoffs / failure modes
	Multi-cluster Kubernetes lifecycle	orchestration patterns	controller bugs can fan out.
Terraform (IaC)	Infra provisioning with strong ecosystem	Mature workflows, policy checks, broad provider support	Drift can become “invisible” between applies; imperative workflows increase human error.
Crossplane	Kubernetes-native infra provisioning	Unified CRD model for infra + apps	Similar management cluster dependency; composition complexity can hide coupling.

Opinionated default:

- Use **CAPI for cluster lifecycle**, and use Terraform/Crossplane for shared primitives where you already have mature modules—*but avoid two tools managing the same resource types*.

1.4.3.6 Upgrade orchestration strategies

Pick one strategy per cluster class (dev/stage/prod), and enforce it.

Strategy	When to use	Pros	Cons / failure modes
In-place rolling (node pool rotation / surge)	Most services; strong PodDisruptionBudgets	Lower cost; incremental	Failure mode: bad CNI/CSI or version skew causes widespread brownouts.
Blue/green clusters	High criticality; strict rollback needs	Fast rollback by traffic switch	Higher cost; requires dual-writing or careful state handling.

Strategy	When to use	Pros	Cons / failure modes
Mixed	Tiered workloads	Balances cost and safety	Complexity; requires clear workload classification and automation.

Upgrade invariants you should enforce:

- One-step Kubernetes minor version upgrades (avoid skipping minors).
- Pre-flight compatibility checks for CNI/CSI/Ingress/Gateway API controllers.
- Residency isolation: upgrade automation, images, logs, and artifacts remain in-region.

1.4.3.7 Autoscaling patterns

You should treat autoscaling as a reliability feature (protects SLOs) and a cost control (FinOps), but it can also amplify incidents if misconfigured.

- **HPA**: scale pods based on metrics (good default for stateless services).
- **VPA**: right-size requests/limits (use with care; can cause restarts).
- **Cluster Autoscaler / Karpenter equivalents**: scale nodes to meet pod demand (critical for bursty workloads).

Common failure modes:

- Metrics pipeline failure causes HPA to freeze or behave unexpectedly; ensure observability backends are highly available and residency-scoped.
- Over-permissive node provisioning roles become a lateral movement path; apply least privilege and rotate credentials where applicable.

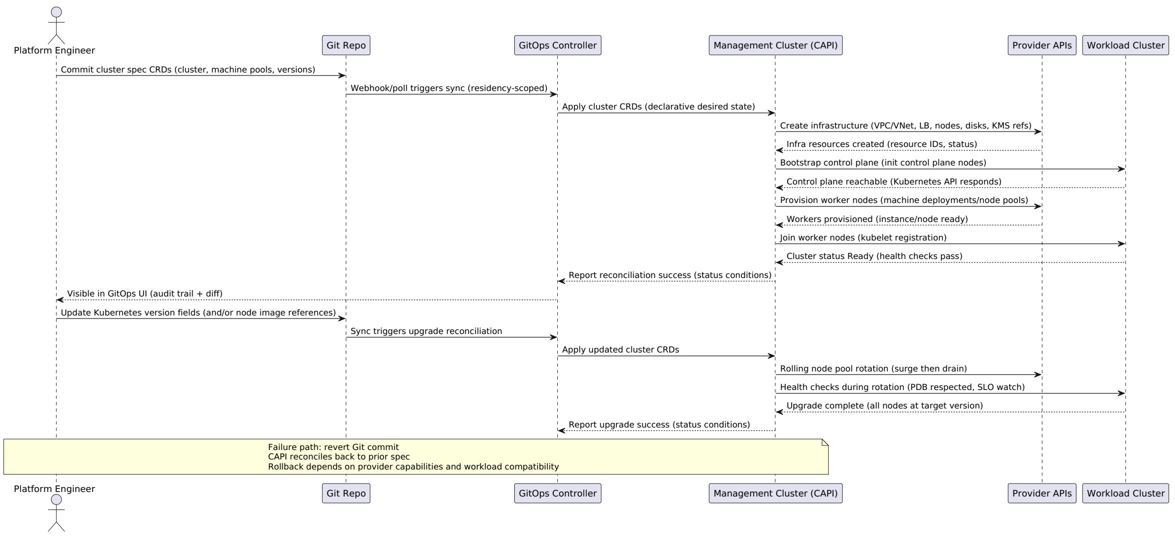
1.4.4 Risks and mitigations

- **Risk: Management cluster as a common failure point (blast radius).**
  Mitigation: separate management clusters per residency domain and per environment; restrict who can change cluster CRDs; progressive rollout of lifecycle controller versions.
- **Risk: Residency violations via logs/artifacts/telemetry during provisioning/upgrades.**
  Mitigation: region-scoped registries, KMS keys, logging sinks, and egress allowlists; CI policy checks that block cross-region endpoints.

- **Risk: Upgrade-induced outages (CNI/CSI incompatibility, PDB gaps, version skew).**
  Mitigation: conformance tests in staging; automated prechecks; enforce PDBs; canary node pool rotation before full rollout.
- **Risk: IAM drift and over-privilege (SOC2 finding and real compromise path).**
  Mitigation: workload identity over static secrets; audit logging for IAM and Kubernetes API; credential rotation/expiry; periodic access reviews.

1.4.5 Diagram: Cluster Lifecycle via Cluster API (CAPI) (diagram_id: cluster-lifecycle-capi)

Alt text: Sequence of provisioning and upgrading clusters with GitOps and Cluster API.


Platform Engineer
Git Repo
GitOps Controller
Management Cluster (CAPI)
Provider APIs
Workload Cluster
Commit cluster spec CRDs (cluster, machine pools, versions)
Webhook/poll triggers sync (residency-scoped)
Apply cluster CRDs (declarative desired state)
Create infrastructure (VPC/VNet, LB, nodes, disks, KMS refs)
Infra resources created (resource IDs, status)
Bootstrap control plane (init control plane nodes)
Control plane reachable (Kubernetes API responds)
Provision worker nodes (machine deployments/node pools)
Workers provisioned (instance/node ready)
Join worker nodes (kubelet registration)
Cluster status Ready (health checks pass)
Report reconciliation success (status conditions)
Visible in GitOps UI (audit trail + diff)
Update Kubernetes version fields (and/or node image references)
Sync triggers upgrade reconciliation
Apply updated cluster CRDs
Rolling node pool rotation (surge then drain)
Health checks during rotation (PDB respected, SLO watch)
Upgrade complete (all nodes at target version)
Report upgrade success (status conditions)
Failure path: revert Git commit
CAPI reconciles back to prior spec
Rollback depends on provider capabilities and workload compatibility
Platform Engineer
Git Repo
GitOps Controller
Management Cluster (CAPI)
Provider APIs
Workload Cluster

1.4.6 Operational considerations

1.4.6.1 Security and compliance baseline (SOC2-aligned)

- **Least privilege:** separate IAM roles for management controllers, node provisioning, and workload identity; deny-by-default for cross-region access.
- **Audit logging:** enable Kubernetes API audit logs and cloud IAM audit logs; ship to residency-scoped, immutable storage with retention aligned to your policies.
- **Rotation/expiry:** prefer workload identity mechanisms over static secrets; rotate any required long-lived credentials; enforce short TTL for tokens/certs where supported.

1.4.6.2 Availability and blast radius controls

- Run clusters across multiple AZs within a region; don't claim 99.9% if you're single-AZ.
- Separate clusters by environment (prod vs non-prod) and by residency domain (EU vs US).
- Treat the management cluster as production infrastructure; define its own SLOs, backups, and upgrade windows.

1.4.6.3 Day-2 operability

- Standardize "cluster classes" (e.g., small/medium/large) with fixed addons, policies, and upgrade cadence.
- Enforce policy-as-code at admission time (OPA or equivalent) for critical invariants: encryption settings, disallowed hostPath, required labels, allowed registries.
- Track version skew: control plane, nodes, CNI/CSI/Ingress/Gateway API controllers.

1.4.7 Checklist

- [ ] You have explicit **Assumptions** documented: residency domains, management cluster strategy, GitOps model.
- [ ] You chose **managed vs self-managed** Kubernetes per provider with an ADR, including failure modes and staffing impact.
- [ ] You standardized **node OS** and hardening baseline, including encryption and audit logging.
- [ ] You implemented **CAPI (or equivalent)** with environment and residency isolation; no shared control plane across EU/US.
- [ ] You defined an **upgrade strategy** (rolling vs blue/green) with prechecks, rollback, and verification gates.
- [ ] You deployed autoscaling (HPA + node autoscaling) with safe defaults and metrics pipeline HA.
- [ ] You can produce SOC2 evidence: GitOps change history, IAM audits, access reviews, and rotation/expiry controls.

1.5 Networking Fundamentals Across Clouds

1.5.1 Objectives

- Give you a **common networking model** that maps cleanly to each provider's VPC/VNet primitives and Kubernetes networking.
- Standardize **addressing, routing, and DNS** so multi-cloud does not devolve into per-team snowflakes.
- Reduce **blast radius** and enable **99.9% availability** with predictable failure domains (provider/region/cluster).

1.5.2 When to use/avoid

Use when

- You operate Kubernetes clusters across **multiple providers and/or regions** and need consistent controls for SOC2 (auditability, least privilege, encryption, change management).
- You must enforce **EU/US data residency separation**, including telemetry, backups, and artifact pulls.

Avoid when

- You cannot staff networking/on-call to handle multi-cloud failure modes (routing, DNS, certificate, and load balancer issues will page you).
- You have mostly stateful workloads with tight coupling to provider-native networking features and no budget for DR testing.

1.5.3 Assumptions

- Each Kubernetes cluster runs inside a single **VPC/VNet** in a single **region**, spanning **at least two AZs** for 99.9% availability.
- You use **private subnets** for Kubernetes nodes (data plane) and keep public exposure to **external load balancers** only.
- You have a real **IPAM** process (tooling can vary; governance cannot).
- You enforce **encryption everywhere** (in transit and at rest) and can produce **audit logs** for network/security changes (SOC2).

1.5.4 Non-goals

- Designing full **hybrid** (on-prem) connectivity and routing policy (covered in cross-cloud connectivity sections).

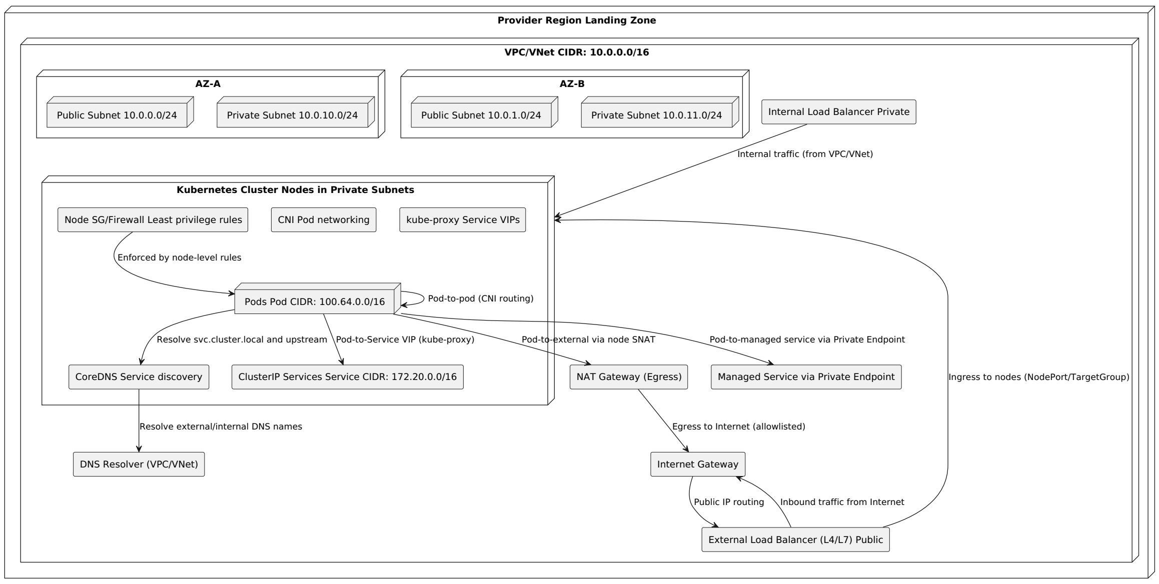
- Achieving “pure portability”: provider load balancers, DNS, IAM, and private endpoints still differ materially.
- Solving active-active stateful replication across clouds (networking enables it; it doesn’t make it easy).

1.5.5 Architecture notes

1.5.5.1 Common reference model (conceptual mapping)

- **Control plane**: Kubernetes API and management systems (GitOps, policy controllers, observability backends). Treat these as critical dependencies with their own isolation and DR.
- **Data plane**: Kubernetes nodes/pods + CNI + kube-proxy + load balancers.
- **Boundaries** you must name and enforce:
  - **Provider → region → VPC/VNet → AZ → cluster → namespace → workload**
- Default posture: **private-by-default**, explicit ingress/egress, explicit routes, explicit DNS behavior.

1.5.5.2 Diagram: `networking-common-model` (alt text: Common multi-cloud networking model for Kubernetes clusters within VPC/VNet.)


Provider Region Landing Zone
VPC/VNet CIDR: 10.0.0.0/16
AZ-A
Public Subnet 10.0.0.0/24
Private Subnet 10.0.10.0/24
AZ-B
Public Subnet 10.0.1.0/24
Private Subnet 10.0.11.0/24
Internal Load Balancer Private
Internal traffic (from VPC/VNet)
Kubernetes Cluster Nodes in Private Subnets
Node SG/Firewall Least privilege rules
CNI Pod networking
kube-proxy Service VIPs
Enforced by node-level rules
Pods Pod CIDR: 100.64.0.0/16
Pod-to-pod (CNI routing)
Resolve svc.cluster.local and upstream
Pod-to-Service VIP (kube-proxy)
Pod-to-external via node SNAT
Pod-to-managed service via Private Endpoint
CoreDNS Service discovery
ClusterIP Services Service CIDR: 172.20.0.0/16
NAT Gateway (Egress)
Managed Service via Private Endpoint
Ingress to nodes (NodePort/TargetGroup)
Resolve external/internal DNS names
Egress to Internet (allowlisted)
DNS Resolver (VPC/VNet)
Internet Gateway
Public IP routing
Inbound traffic from Internet
External Load Balancer (L4/L7) Public

1.5.5.3 Addressing and IPAM standards (opinionated defaults)

You want boring, collision-resistant allocations. Most multi-cloud “mystery outages” start with IP overlap or accidental route propagation.

- **VPC/VNet CIDR**: allocate per **region** and **environment** from a centrally governed RFC1918 pool (e.g., /16 per region/env as a baseline).
- **Pod CIDR**: do **not** reuse VPC/VNet CIDRs; treat Pod CIDRs as their own routable domain (often non-RFC1918 is used internally; if you do, ensure it never leaks to WAN/peering).
- **Service CIDR**: unique per cluster; never overlap across clusters if you plan cross-cluster service discovery or shared tooling.
- Plan growth explicitly: IP exhaustion is an outage, not an inconvenience. Choose subnet sizing that survives node scaling and AZ loss.

Decision matrix (CIDR sizing baseline)

Item	Conservative default	Tradeoff	Failure mode if wrong
VPC/VNet	/16 per region+env	More address space reserved	Re-IP is expensive and risky
Private subnets	/20 per AZ	Slightly larger blast radius if misrouted	Node scale hits ENI/IP limits
Pod CIDR	/16 per cluster	Larger routing tables in some CNIs	IP exhaustion causes scheduling failures
Service CIDR	/18 per cluster	Wastes space	Collisions break service discovery

1.5.5.4 Routing and egress (control it or it will control you)

- Prefer **explicit route tables** per subnet class (public/private) with documented intent.
- Default **no direct internet egress** from private subnets except via controlled NAT/egress gateway.
- Use **egress allowlists** for:
  - artifact registries (OCI pulls)
  - time sync endpoints
  - identity endpoints (OIDC/JWKS)
  - telemetry endpoints (residency-scoped)

- For SOC2, ensure changes to routes/NACLs/SGs are **audited** and tied to change tickets/PRs.

Tradeoffs

- Centralized egress simplifies audit and filtering, but increases shared dependency risk (a broken NAT policy can take down everything).
- Distributed egress reduces blast radius but is harder to govern consistently.

1.5.5.5 Load balancing: L4 vs L7, internal vs external

- **External LB**: terminate at the edge (L7) when you need WAF-like controls, TLS policy, and routing by host/path. Terminate in-cluster only when you can operate certs, cipher policy, and DDoS posture appropriately.
- **Internal LB**: use for east-west entry points (shared services, internal APIs) and for managed service proxies.
- Health checks must validate **real readiness**, not “port open”.

Common failure modes

- Health check mismatch causes flapping and partial outages (especially during rollouts).
- L7 timeouts/defaults differ per provider; you must standardize timeout budgets at the app layer and ingress layer.

1.5.5.6 DNS: split-horizon, TTL strategy, and service discovery

- **Split-horizon DNS**: keep internal names resolvable only inside the VPC/VNet (or via controlled resolvers). This reduces data exfil paths and accidental exposure.
- **CoreDNS**: treat it as critical infrastructure; scale it and isolate it. Configure upstreams explicitly and monitor NXDOMAIN/timeout rates.
- **TTL strategy** (multi-cloud friendly):
  - Low TTLs help failover but increase resolver load and amplify DNS outages.
  - High TTLs reduce load but slow failover.
  - Pragmatic baseline: low TTL only for **failover records**, normal TTL for stable internal names.

1.5.6 Risks and mitigations

- **Risk: IP overlap across providers/regions breaks peering, service discovery, and DR.**

Mitigation: enforce IPAM approvals; automated collision checks in CI; reserve pools per provider/region/env.
- **Risk: Residency violations via DNS/logging/telemetry egress.**
  Mitigation: residency-scoped resolvers and telemetry endpoints; egress allowlists; KMS keys scoped per region; policy-as-code checks on egress rules.
- **Risk: Shared egress/NAT becomes a single point of failure.**
  Mitigation: multi-AZ NAT/egress, monitored route tables, runbooks for rollback, and per-cluster circuit breakers for non-critical egress.
- **Risk: Provider LB behavior differences cause inconsistent availability.**
  Mitigation: standard ingress SLOs, conformance tests per cluster, and documented provider-specific defaults via ADRs.

1.5.7 Operational considerations

1.5.7.1 SOC2 controls you must be able to evidence

- **Least privilege**: security groups/firewalls scoped to required ports/protocols; separate admin access paths; restrict control plane access.
- **Audit logging**: route table, firewall/SG, DNS zone, and load balancer config changes logged and retained; GitOps history for Kubernetes objects.
- **Rotation/expiry**:
  - TLS cert rotation for ingress and mTLS (where used).
  - Credentials for DNS automation and IPAM tooling.
  - Short-lived auth for cluster access; eliminate long-lived kubeconfigs where possible.

1.5.7.2 Day-2 monitoring signals (networking SLO backstops)

- DNS: CoreDNS latency, SERVFAIL/NXDOMAIN rates, upstream timeout rates.
- Egress: NAT port utilization (or equivalent), dropped connections, top destinations, denied flows.
- Ingress: LB 4xx/5xx, target health, connection resets, p95/p99 latency, saturation metrics.
- Kubernetes: Pod IP exhaustion, node ENI/IP limits, kube-proxy sync errors, CNI errors.

1.5.7.3 Standard runbook patterns (what you should operationalize)

- “Ingress outage triage”: differentiate DNS vs LB vs in-cluster routing vs app readiness.
- “Egress blocked”: validate allowlists, NAT health, route tables, and per-namespace NetworkPolicy (if used).
- “AZ failure”: confirm cross-AZ routes and LB target behavior; ensure Pod disruption budgets don’t block rescheduling.

1.5.8 Checklist

- [ ] IPAM: unique, documented CIDRs for VPC/VNet, Pod CIDR, Service CIDR; collision checks automated.
- [ ] Subnets: nodes in private subnets across ≥2 AZs; public subnets limited to edge components.
- [ ] Routing: explicit route tables; controlled NAT/egress; documented peering/private endpoint routes.
- [ ] DNS: split-horizon in place; CoreDNS scaled/monitored; TTL policy defined for failover vs stable names.
- [ ] Load balancing: L4/L7 choice documented; health checks validate readiness; timeout budgets standardized.
- [ ] Security: least privilege SG/firewall rules; audit logging enabled/retained; rotation/expiry defined for certs and automation identities.
- [ ] Residency: EU/US DNS/telemetry/artifact flows are region-scoped and enforced with policy-as-code and egress allowlists.
- [ ] Availability: multi-AZ for NAT/LB/control dependencies; failure modes tested (DNS outage, NAT failure, AZ loss).

1.6 Cross-Cloud Connectivity Patterns (s06)

1.6.1 Objectives

- Give you secure, operable options to connect Kubernetes clusters across providers while preserving tenant/team boundaries and limiting blast radius.
- Set expectations for latency, throughput, and failover behavior (including asymmetric routing and inspection insertion).
- Keep you inside SOC2 constraints: least privilege, audit logging, encryption everywhere, and EU/US data residency separation.

1.6.2 When to use/avoid

Use when

- You need **EU/US data residency separation** with controlled, audited cross-boundary connectivity (ideally metadata-only).
- You want **provider risk reduction** (multi-provider failover or active/active reads) and you can fund/operate the network.
- You have clear RTO/RPO targets and you test failover routinely.

Avoid when

- You can't staff on-call to handle routing, tunnels, MTU issues, certificate rotation, and incident response.
- Your workload is tightly coupled to provider-native L4/L7 features and you expect "portable networking" to hide it (it won't).
- You expect low latency across continents; physics and egress billing will dominate.

1.6.3 Architecture notes

1.6.3.1 Assumptions

- You operate at least two providers and at least one region per provider; clusters are per region/provider failure domain.
- You enforce **encryption in transit** on every cross-cloud hop (IPsec/WireGuard and/or mTLS overlays).
- You maintain separate EU and US environments (including telemetry, backups, and artifact distribution) unless explicitly approved.

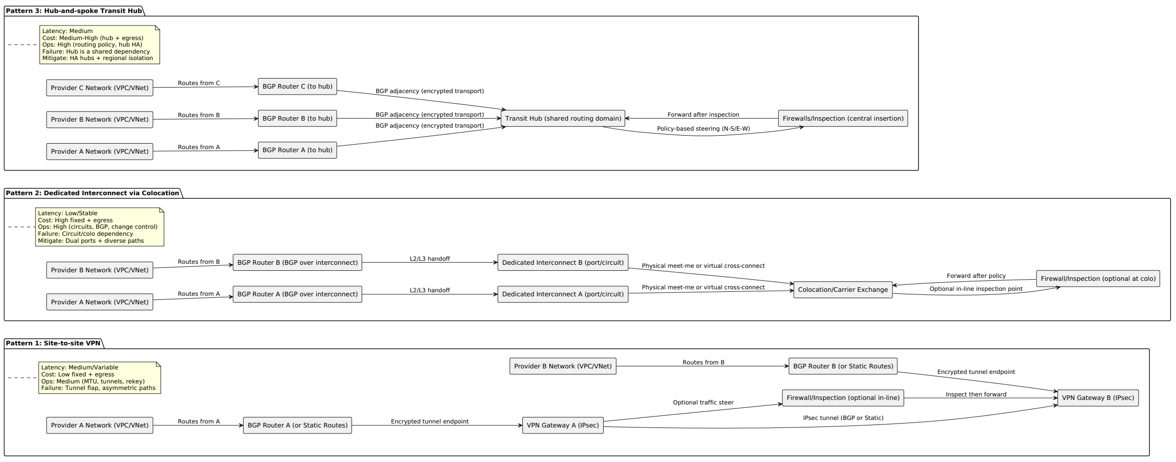
1.6.3.2 Non-goals

- Providing a single "flat network" across clouds with no segmentation (this increases blast radius and complicates SOC2 evidence).
- Solving global traffic management/ingress (covered in later sections); this section is about network underlay connectivity and routing.

1.6.3.3 Reference patterns (and what they optimize for)

- **Pattern 1: Site-to-site VPN (IPsec)**: lowest upfront cost, fastest to start; worst for jitter/throughput predictability.

- **Pattern 2: Dedicated interconnect via colocation/carrier exchange**: predictable performance, stronger operational leverage; higher fixed costs and longer lead time.
- **Pattern 3: Hub-and-spoke transit hub**: scales connectivity and centralizes inspection/policy; adds dependency on the hub (failure domain you must design around).



Pattern 3: Hub-and-spoke Transit Hub
Latency: Medium
Cost: Medium-High (hub + egress)
Ops: High (routing policy, hub HA)
Failure: Hub is a shared dependency
Mitigate: HA hubs + regional isolation
Provider C Network (VPC/VNet)
Routes from C
BGP Router C (to hub)
BGP adjacency (encrypted transport)
Provider B Network (VPC/VNet)
Routes from B
BGP Router B (to hub)
BGP adjacency (encrypted transport)
Transit Hub (shared routing domain)
Forward after inspection
Firewalls/Inspection (central insertion)
BGP adjacency (encrypted transport)
Policy-based steering (N-S/E-W)
Provider A Network (VPC/VNet)
Routes from A
BGP Router A (to hub)
Pattern 2: Dedicated Interconnect via Colocation
Latency: Low/Stable
Cost: High fixed + egress
Ops: High (circuits, BGP, change control)
Failure: Circuit/colo dependency
Mitigate: Dual ports + diverse paths
Provider B Network (VPC/VNet)
Routes from B
BGP Router B (BGP over interconnect)
L2/L3 handoff
Dedicated Interconnect B (port/circuit)
Physical meet-me or virtual cross-connect
Provider A Network (VPC/VNet)
Routes from A
BGP Router A (BGP over interconnect)
L2/L3 handoff
Dedicated Interconnect A (port/circuit)
Physical meet-me or virtual cross-connect
Colocation/Carrier Exchange
Forward after policy
Optional in-line inspection point
Firewall/Inspection (optional at colo)
Pattern 1: Site-to-site VPN
Latency: Medium/Variable
Cost: Low fixed + egress
Ops: Medium (MTU, tunnels, rekey)
Failure: Tunnel flap, asymmetric paths
Provider B Network (VPC/VNet)
Routes from B
BGP Router B (or Static Routes)
Encrypted tunnel endpoint
Optional traffic steer
Firewall/Inspection (optional in-line)
Inspect then forward
VPN Gateway B (IPsec)
IPsec tunnel (BGP or Static)
Provider A Network (VPC/VNet)
Routes from A
BGP Router A (or Static Routes)
Encrypted tunnel endpoint
VPN Gateway A (IPsec)

Figure (diagram_id: crosscloud-connectivity-options): Connectivity options between clouds: VPN, dedicated interconnect, and transit hub topologies.

1.6.4 Operational considerations

1.6.4.1 Routing and failover behavior (BGP vs static)

- **Prefer BGP** for anything beyond trivial point-to-point links. Static routes fail “silently” (blackholing) unless you build health checks and automation.
- Design explicitly for **asymmetric routing**:
    - Failure mode: return traffic takes a different path and bypasses inspection/NAT, breaking stateful firewalls and confusing troubleshooting.
    - Mitigation: keep symmetric routing for stateful inspection paths (policy-based routing, consistent next hops), or use stateless inspection + mTLS at L7.

1.6.4.2 Encryption strategy (meet “encryption everywhere”)

- Underlay: **IPsec** (common for VPN and over interconnect) or **WireGuard** (operationally simpler in some stacks, but validate provider/OS support and auditability).

- Overlay: **mTLS** between services (service mesh or app-level). This is not optional when you traverse shared networks or transit hubs.
- Key management:
  - Enforce rotation/expiry for IPsec/WireGuard keys and mTLS certificates.
  - Separate keys per residency boundary (EU vs US) and per environment (dev/test/prod) to reduce blast radius.

1.6.4.3 Security controls (SOC2-aligned)

- **Least privilege**: restrict who can change routing (BGP sessions, route tables, security policies). Treat route changes as high-risk.
- **Audit logging**:
  - Centralize network control plane logs per residency boundary (EU logs stay in EU, US logs stay in US).
  - Retain logs long enough for investigations and SOC2 evidence; ensure tamper resistance (append-only or immutability controls).
- **Segmentation**:
  - Do not build a shared flat address space across tenants/teams; route only the required prefixes.
  - Use explicit allowlists for cross-cloud prefixes and ports; deny-by-default.

1.6.4.4 Data residency failure modes you must plan for

- Telemetry and shared services often violate residency unintentionally.
  - Failure mode: metrics/logs/traces from EU clusters exported to US observability backends via “global” collectors.
  - Mitigation: residency-scoped collectors, endpoints, and encryption keys; CI policy checks to block cross-boundary endpoints; network egress allowlists.

1.6.4.5 Cost model realities (FinOps-sensitive)

- VPN: low fixed cost; **egress dominates** quickly and throughput scaling is messy (more tunnels, more ops).
- Interconnect: high fixed cost (ports/circuits/colo); lower variance and higher usable throughput; still pay egress in many cases.
- Transit hub: you pay for the hub plus egress; you also pay operationally for centralized routing/inspection complexity.

1.6.4.6 Risks and mitigations

- **Risk: Hub becomes a hard dependency (Pattern 3)**
  Mitigation: dual hubs per region/provider boundary, failure-domain isolation, and tested bypass/fallback routes.
- **Risk: Route leaks expose unintended networks**
  Mitigation: prefix filters, max-prefix limits, explicit route advertisements, and change-control with peer review.
- **Risk: MTU/fragmentation causes intermittent failures**
  Mitigation: set and test path MTU, standardize tunnel MTU, and include MTU checks in incident runbooks.
- **Risk: Inspection devices drop traffic under load**
  Mitigation: capacity planning, autoscaling where possible, and clearly defined “fail open vs fail closed” behavior per traffic class (document the tradeoff).

1.6.5 Checklist

- [ ] You have documented EU/US residency boundaries for **application data, telemetry, backups, and artifacts**.
- [ ] Cross-cloud connectivity uses **encrypted transport** (IPsec/WireGuard) and **mTLS** for service-to-service traffic.
- [ ] Routing is **BGP with prefix filters** (or static routes with health-checked automation and documented limitations).
- [ ] You’ve identified where **firewalls/inspection** are inserted and verified symmetric routing for stateful paths.
- [ ] IAM for network changes follows **least privilege**, with approvals and auditable change history.
- [ ] Network control plane logs are collected with **residency-aware storage** and retention.
- [ ] You have tested failover: tunnel/circuit failure, hub failure, and route withdrawal; results map back to your **99.9% SLO** and error budget.

1.7 Ingress, Egress, and Global Traffic Management (s07)

1.7.1 Objectives

- Define how clients reach services across **providers/regions/clusters** with predictable latency and controlled blast radius.

- Provide **global traffic management** patterns (DNS-based, anycast, global L7 LB) with explicit failover behavior.
- Standardize **ingress** using Kubernetes **Gateway API** (preferred) while allowing provider-specific implementations at the edge.
- Enforce **egress control** (NAT/egress proxy/policy) to reduce data exfiltration risk and residency violations.
- Meet constraints: **SOC2**, **EU/US data residency**, **encryption everywhere**, and **99.9% availability**.

1.7.2 When to use / avoid

Use when

- You run **stateless or loosely coupled** services across multiple providers and need provider risk reduction and/or EU/US separation.
- You can operate health checks, runbooks, and change control (GitOps) with auditability.
- You need explicit, testable **RTO/RPO** and controlled failover behavior.

Avoid when

- You cannot staff **24/7 operations** or lack mature incident response; global traffic steering is operationally unforgiving.
- Your workloads depend on provider-native L7 features that you can’t reasonably reproduce (WAF rulesets, proprietary routing, tight identity coupling).
- Your state tier cannot meet cross-provider DR requirements without unacceptable cost/complexity (consistency coupling becomes the real bottleneck).

1.7.3 Architecture notes (reference)

1.7.3.1 Assumptions

- You maintain hard **EU/US residency boundaries** for customer data and *derived data* (logs/metrics/traces, backups, artifacts).
- All ingress/egress paths enforce **TLS in transit**; secrets/certs have **rotation/ expiry** automation.
- Centralized **audit logging** exists per residency boundary (no “global” log sink that violates residency).
- You have centralized **IPAM** with non-overlapping RFC1918 allocations for VPC/VNet, Pod CIDRs, and Service CIDR.

1.7.3.2 Non-goals

- Guaranteed “write-anywhere” active/active for strongly consistent state across providers.
- A single, universal edge feature set; provider edges will differ and must be isolated behind standard interfaces.

1.7.3.3 Target patterns (opinionated defaults)

- **Global traffic management:** start with **DNS-based** steering + health checks; graduate to anycast/global L7 only when DNS failure modes are unacceptable for your user experience.
- **Ingress standard:** Kubernetes **Gateway API** as the stable contract; provider ingress controllers/gateways implement it per cluster.
- **Egress standard:** default-deny egress at the namespace level; allow only via **egress gateway/proxy** with explicit allowlists and logging.

1.7.3.4 Risks and mitigations (major)

- **Risk: DNS failover is not instant** (TTL + resolver caching + client retry behavior).
  Mitigation: keep TTL low where viable; design clients for retries; use idempotency; consider anycast/global L7 for strict RTO.
- **Risk: residency violations via telemetry and WAF/CDN logs.**
  Mitigation: region-scoped endpoints, residency-scoped keys, allowlisted exporters, and CI policy checks blocking cross-boundary sinks.
- **Risk: inconsistent WAF/DDoS posture across providers.**
  Mitigation: define a baseline control set (rate limits, bot rules, geo blocks, logging fields) and validate continuously with synthetic tests.
- **Risk: split-brain traffic during partial outages** (health checks disagree, propagation delays).
  Mitigation: multi-signal health (L7 + dependency probes), conservative failover thresholds, and explicit “fail closed/open” decisions per service tier.

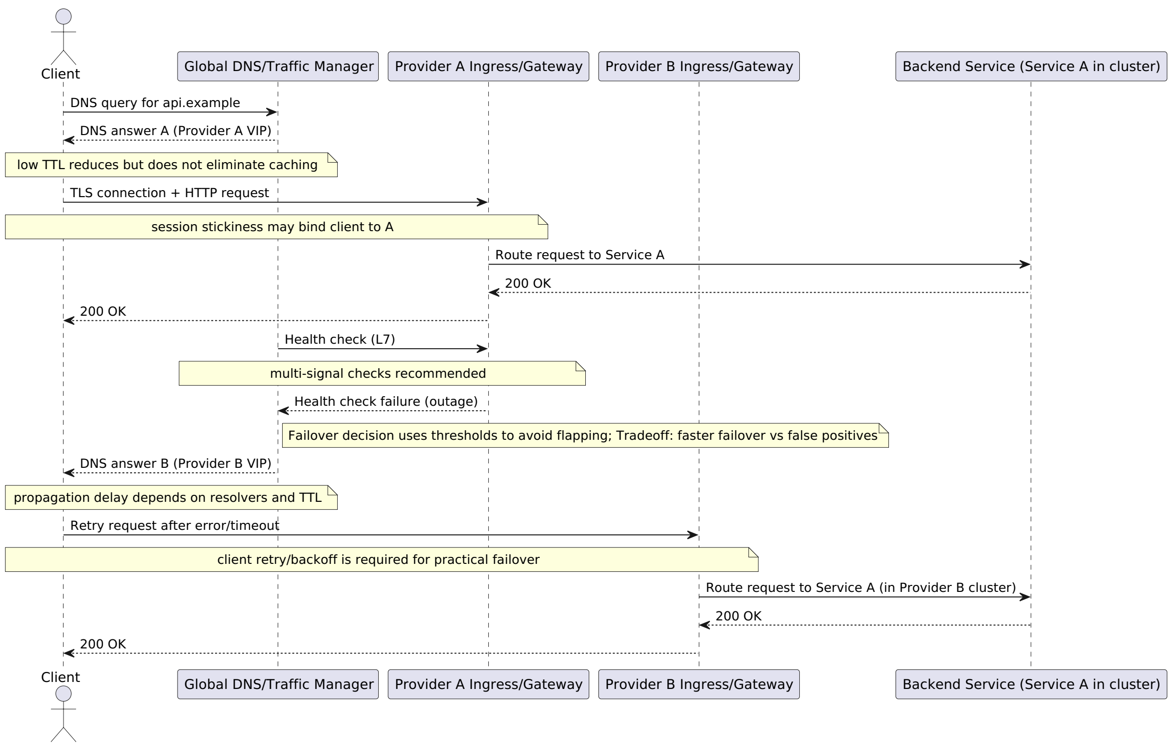
1.7.4 Operational considerations

- Treat global traffic policy changes as **high-risk**: require PR review, staged rollout, and rapid rollback.
- Health checks must validate **user-path reality** (L7 checks that cover auth, routing, and a lightweight backend dependency), not just “port open.”

- Define per-service **session semantics**:
  - If you need stickiness, document what happens on failover (expect session loss unless you have cross-provider session state replication —usually not worth it).
- Encryption and identity:
  - Terminate TLS at the edge only if you maintain **mTLS or TLS** to the cluster ingress and preserve end-user identity via validated headers (or re-auth at edge).
  - Use **least privilege** for DNS/traffic manager control APIs; log every change (who/what/when) for SOC2 evidence.
- Egress:
  - Default route all outbound traffic through **NAT + egress policy** and (where required) **egress proxy** with TLS inspection avoided unless you can prove compliance impact and handle key management safely.
  - Log egress by destination (domain/IP), namespace, and workload identity; rotate logs per retention policy and residency.

1.7.5 Global Traffic Management and Failover (diagram)

Diagram: `global-traffic-failover`
Alt text: DNS/global traffic failover from one cloud provider ingress to another.


Client
Global DNS/Traffic Manager
Provider A Ingress/Gateway
Provider B Ingress/Gateway
Backend Service (Service A in cluster)
DNS query for api.example
DNS answer A (Provider A VIP)
low TTL reduces but does not eliminate caching
TLS connection + HTTP request
session stickiness may bind client to A
Route request to Service A
200 OK
200 OK
Health check (L7)
multi-signal checks recommended
Health check failure (outage)
Failover decision uses thresholds to avoid flapping; Tradeoff: faster failover vs false positives
DNS answer B (Provider B VIP)
propagation delay depends on resolvers and TTL
Retry request after error/timeout
client retry/backoff is required for practical failover
Route request to Service A (in Provider B cluster)
200 OK
200 OK
Client
Global DNS/Traffic Manager
Provider A Ingress/Gateway
Provider B Ingress/Gateway
Backend Service (Service A in cluster)

1.7.6 Design decisions and tradeoffs (decision matrix)

Option	How it works	Strengths	Failure modes / tradeoffs	Best fit
DNS-based global traffic management	Health-checked DNS returns provider/ region endpoints	Simple, widely supported, easy to audit/ change	TTL/resolver caching delays; inconsistent client retry behavior; possible split traffic during partial outages	Default starting point; 99.9% with good client retries
Anycast (global IP advertised from multiple sites)	Routing sends clients to nearest/ available POP/ provider	Faster failover perception; stable IP	Harder to reason about routing; requires deeper network ops; debugging is harder	High-traffic edges with strict latency/ failover needs
Global L7 load balancer	Central L7 routing with health + policy	Rich L7 steering, uniform policy surface	Central dependency; can violate residency if misconfigured; higher complexity	When you need consistent L7 features and can prove residency controls

1.7.7 Ingress architecture (cluster-level)

1.7.7.1 Objectives

- Provide a consistent **north-south** entry point per cluster.
- Standardize routing objects using **Gateway API** to reduce provider coupling.

1.7.7.2 When to use / avoid

- **Use** Gateway API when multiple teams deploy services and you need consistent policy, review, and portability.

- **Avoid** bespoke per-team ingress controllers; it increases blast radius and makes SOC2 evidence harder.

1.7.7.3 Architecture notes

- One ingress/gateway layer per cluster (per tenant boundary if required) with:
    - TLS termination policy, certificate automation, and rotation/expiry
    - Rate limiting and basic L7 protections (where feasible)
    - Structured access logs with residency-scoped sinks
- Use provider-native load balancers at the edge, but hide specifics behind Gateway API objects and documented annotations (ADR required when annotations are unavoidable).

1.7.7.4 Operational considerations

- Enforce **change management**: Gateway changes via GitOps PRs with required reviewers and automated policy checks (OPA).
- Protect the ingress control plane:
    - Separate admin access from developer access (least privilege RBAC/IAM).
    - Audit log all config changes; retain evidence per SOC2 requirements (and residency).
- Capacity planning:
    - Define HPA/scale rules for ingress proxies; load test before reducing TTLs (failover increases burst load).

1.7.7.5 Checklist

- [ ] Gateway API chosen as the default interface; exceptions documented via ADR.
- [ ] TLS policy defined (minimum versions/ciphers), certificates automated, rotation/expiry tested.
- [ ] Access logs enabled and shipped to residency-appropriate sinks with retention.
- [ ] Ingress SLOs/alerts defined (latency, 5xx, saturation, health-check success).
- [ ] Break-glass procedure defined and audited (time-bound elevation, logged).

1.7.8 Egress control (cluster-level)

1.7.8.1 Objectives

- Reduce exfiltration risk and enforce residency by controlling outbound destinations.
- Provide deterministic outbound IPs where required (partner allowlists, auditing).

1.7.8.2 When to use / avoid

- **Use** egress proxies/gateways for regulated workloads, SaaS allowlists, and strict data boundary enforcement.
- **Avoid** “allow all egress” defaults; it’s cheap until you have an incident, then it’s expensive and hard to prove compliance.

1.7.8.3 Architecture notes

- Baseline: provider NAT gateway + Kubernetes NetworkPolicy (or CNI policy) with **default-deny egress**.
- For higher control: route outbound via an **egress gateway/proxy** that enforces:
    - Destination allowlists (FQDN where possible, otherwise IP/CIDR with IPAM discipline)
    - Per-namespace/workload identity attribution
    - Audit logging (who talked to what, when)
- Residency: ensure proxies and log sinks stay within the same **region/provider boundary** as required.

1.7.8.4 Operational considerations

- Rotation/expiry:
    - If the egress proxy uses mTLS to upstreams, automate cert rotation and alert on nearing expiry.
- Failure mode: egress proxy becomes a chokepoint.
    - Mitigate with HA, load tests, and explicit “fail open vs fail closed” per service tier (documented).
- Verification:
    - Continuous egress policy tests (synthetic attempts to blocked destinations) for drift detection.

1.7.8.5 Checklist

- [ ] Default-deny egress enforced at namespace/team boundary.
- [ ] Approved outbound destinations documented and version-controlled.
- [ ] Egress logs collected with residency constraints; access is least privilege.
- [ ] Egress path is HA; failure behavior (open/closed) documented and tested.
- [ ] Key/cert rotation for proxy identities implemented and monitored.

1.7.9 Runbook: DNS-based failover (canary + rollback)

1.7.9.1 Objectives

- Fail over client traffic from Provider A to Provider B with controlled blast radius and auditable steps.

1.7.9.2 When to use / avoid

- **Use** for provider/region outages or planned maintenance with proven Provider B readiness.
- **Avoid** as an ad-hoc fix without verification; you can amplify an outage by shifting traffic to an unready target.

1.7.9.3 Architecture notes

- This runbook assumes:
  - Health checks exist for both providers and reflect user-path correctness.
  - DNS records are managed via GitOps (preferred) or tightly controlled automation with audit logs.

1.7.9.4 Operational considerations

- Expect partial effectiveness until caches expire. Your real “failover time” is **TTL + client retry/backoff**.
- Communicate to incident channel: expected propagation window, user impact, and verification signals.

1.7.9.5 Procedure (step-by-step)

1. **Prechecks**

    1. Confirm incident scope: Provider A ingress health, cluster health, and dependency health (DB, IdP, KMS, etc.).
    2. Verify Provider B readiness:
        - Ingress is healthy (L7 checks passing).
        - Backend Service A is passing readiness and serving expected responses.
        - Observability and logging are functioning in the correct residency boundary.
    3. Confirm DNS TTL and current routing policy; record current state for rollback.

2. **Canary failover**

    1. Shift a small percentage of traffic (or a subset of regions/records) to Provider B if your DNS manager supports weighted/geo policies.
    2. Verify (within 5–10 minutes, adjusted for TTL):
        - Success rate and latency SLI for the canary slice
        - Error logs at ingress and service
        - Egress dependency success (common hidden failure)

3. **Full failover**

    1. Update DNS/traffic manager to route primary traffic to Provider B.
    2. Increase monitoring sensitivity temporarily (page on elevated 5xx, latency, saturation).

4. **Verification**

    1. Confirm new client traffic distribution (ingress request counts by provider).
    2. Validate end-to-end critical transactions (synthetic checks from multiple geos).
    3. Confirm audit evidence captured: change request/PR, approver, timestamps, and system audit logs.

5. **Rollback (if Provider B degrades)**

    1. Revert DNS/traffic manager to Provider A (or to the last known-good state) via Git revert/automation rollback.

2. Restore previous weights/policies; document reason and observed signals.
3. Verify traffic returns and SLI stabilizes; keep incident open until steady state is observed across TTL windows.

1.7.9.6 Checklist

- [ ] Provider B readiness validated (L7 checks + key dependencies).
- [ ] DNS change performed through audited workflow (GitOps/PR or equivalent).
- [ ] Canary executed when possible; SLIs verified before full shift.
- [ ] Rollback path exercised and documented.
- [ ] Post-incident: ADR/update for any uncovered coupling (identity, egress, WAF, telemetry, KMS).

1.8 Identity, Authentication, and Authorization (s08)

1.8.1 Objectives

- Standardize identity for humans and workloads across clusters/regions/providers.
- Enforce least privilege with auditable access paths that meet SOC2 controls.
- Eliminate long-lived credentials via short-lived token exchange and rotation/expiry defaults.
- Support EU/US data residency without "identity backdoors" (e.g., shared admin accounts crossing tenant/team boundaries).

1.8.2 When to use / avoid

Use when

- You run Kubernetes across multiple providers and need consistent access patterns with provider-specific implementations hidden behind a common model.
- You need provable auditability (GitOps history, IAM logs, Kubernetes audit logs) and operationally safe break-glass.
- You have clear tenant/team boundaries and you want predictable blast radius.

Avoid when

- You cannot operate the ongoing complexity: IdP integrations, provider IAM mappings, RBAC drift control, token lifetimes, and audit pipelines.
- You rely on legacy systems that require static credentials everywhere (you will create chronic rotation incidents and SOC2 exceptions).

1.8.3 Assumptions

- You have a centralized Identity Provider (IdP) supporting OIDC and/or SAML.
- Each provider supports federated access to provider IAM and a workload identity equivalent (e.g., IRSA/Workload Identity/managed identities).
- Kubernetes API audit logging is enabled and shipped to a centralized SIEM with immutable retention.
- You can operate separate EU and US environments (clusters/regions) with region-scoped secrets, keys, and telemetry sinks.

1.8.4 Non-goals

- Building a custom IdP or custom auth proxy for Kubernetes.
- Full hybrid/on-prem identity bridging patterns (out of scope per program context).
- “Portable IAM” abstractions that hide all provider differences (you will still need provider IAM constructs for KMS, storage, DB access, and network policy).

1.8.5 Architecture notes (reference model)

You implement identity as two related control paths:

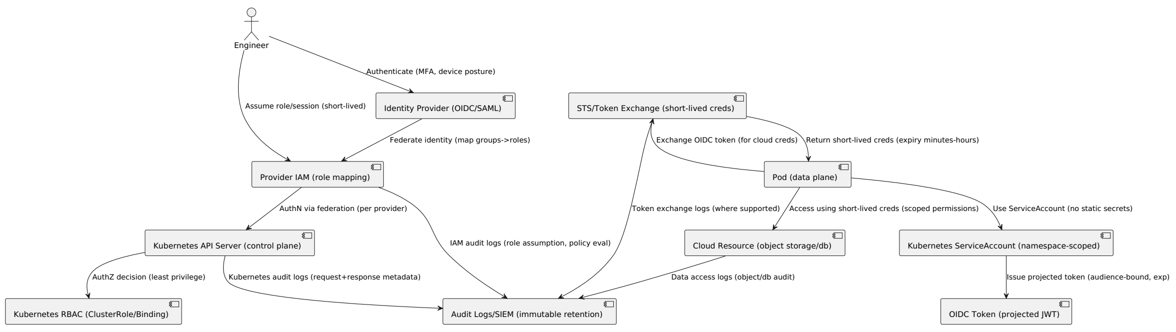
1. **Human access (control plane access)**

    - Engineers authenticate to the **IdP**.
    - The IdP federates to **provider IAM** (per provider) to obtain a role/session.
    - Kubernetes API access uses that federated identity and is authorized via **Kubernetes RBAC** (and optionally admission controls like OPA/Gatekeeper).
    - You treat each cluster as its own security boundary; do not rely on “global cluster-admin.”

2. **Workload access (data plane access to cloud resources)**
    - Pods use a **Kubernetes ServiceAccount**.
    - The ServiceAccount issues a **projected OIDC token**.
    - A provider-specific **STS/token exchange** issues **short-lived cloud credentials**.
    - Workloads access cloud resources (object storage/DB/etc.) without static secrets in Kubernetes.

1.8.5.1 Diagram: Identity Federation and Workload Identity (diagram_id: identity-federation-workload-identity)

Alt text: Federated human access and workload identity using OIDC token exchange to cloud IAM.


Engineer
Authenticate (MFA, device posture)
Assume role/session (short-lived)
Identity Provider (OIDC/SAML)
STS/Token Exchange (short-lived creds)
Federate identity (map groups->roles)
Exchange OIDC token (for cloud creds)
Return short-lived creds (expiry minutes-hours)
Provider IAM (role mapping)
Pod (data plane)
AuthN via federation (per provider)
Token exchange logs (where supported)
Access using short-lived creds (scoped permissions)
Use ServiceAccount (no static secrets)
IAM audit logs (role assumption, policy eval)
Kubernetes API Server (control plane)
Cloud Resource (object storage/db)
Kubernetes ServiceAccount (namespace-scoped)
AuthZ decision (least privilege)
Kubernetes audit logs (request+response metadata)
Data access logs (object/db audit)
Issue projected token (audience-bound, exp)
Kubernetes RBAC (ClusterRole/Binding)
Audit Logs/SIEM (immutable retention)
OIDC Token (projected JWT)

1.8.6 Risks and mitigations

Risk / failure mode	Why it happens	Mitigation you should implement
RBAC drift and privilege creep	Manual changes in-cluster; ad-hoc bindings	GitOps-manage RBAC; block manual edits; periodic access reviews; alert on cluster-admin bindings
Residency violations via centralized identity or logs	Shared SIEM/ telemetry sinks; cross-region role reuse	Separate EU/US log sinks and retention; region-scoped IAM roles; CI policy checks preventing cross-residency bindings
Break-glass becomes the normal path	On-call friction; SSO outages; mis-scoped roles	Tight runbook + approvals; timeboxed access; post-incident

Risk / failure mode	Why it happens	Mitigation you should implement
		review; ensure SSO reliability and caching strategy
Token exchange outages break workloads	STS dependency; misconfigured audiences/issuers	Multi-AZ control plane; health checks; conservative token TTL; circuit breakers and graceful degradation in apps
Over-broad workload permissions	Copy/paste IAM policies	Start from deny-by-default; resource-level scoping; per-namespace/per-team roles; continuous policy linting

1.8.7 Operational considerations

- **Least privilege**
    - Humans: map IdP groups to provider IAM roles, then map roles to Kubernetes RBAC with the minimum ClusterRole/Role.
    - Workloads: one ServiceAccount per workload identity boundary (typically per app per namespace), not “one per namespace.”
- **Audit logging (SOC2)**
    - Enable Kubernetes audit logs (control plane) and ship to SIEM with tamper-resistant storage and defined retention.
    - Enable IAM audit logs and data access logs for critical resources (object storage, KMS, databases).
    - Correlate identities end-to-end (IdP subject → IAM role session → Kubernetes user → workload identity).
- **Rotation/expiry**
    - Human sessions: short-lived, MFA enforced; no shared accounts.
    - Workload credentials: short-lived by design; fail closed on expiry; monitor token exchange latency/errors.
    - Keys/certs: use automated rotation; document maximum validity; alert before expiry.

- **Tenant/team boundaries**
  - Treat **namespace** as a baseline boundary, not a hard security boundary by itself.
  - Combine: namespace + RBAC + NetworkPolicy + PSA (Pod Security Admission) + admission policies (OPA) for defense-in-depth.
- **Availability (99.9%)**
  - Your identity path is a dependency for both control plane access and workload-to-cloud access. Plan for:
    - IdP outages (operationally: break-glass; technically: session caching where safe).
    - Provider IAM/STS brownouts (graceful degradation, retries with jitter, conservative TTLs).

1.8.8 Checklist

- [ ] IdP integrated with each provider IAM using OIDC/SAML federation; no long-lived IAM users for engineers.
- [ ] Kubernetes API authentication uses federated identities; cluster-admin is restricted and timeboxed.
- [ ] RBAC is GitOps-managed; periodic access reviews are scheduled and evidenced (SOC2).
- [ ] Workload identity enabled per provider; no static cloud keys stored in Kubernetes Secrets.
- [ ] Token TTLs, audience, issuer, and namespace scoping are documented and policy-checked in CI.
- [ ] Kubernetes audit logs + IAM audit logs + critical data access logs are shipped to SIEM with immutable retention.
- [ ] EU and US environments have residency-scoped roles, keys, and log sinks; cross-residency bindings are prevented by policy.
- [ ] Break-glass process exists with step-up auth, approvals, time limits, and post-use review.

1.9 Policy, Governance, and Compliance Controls (s09)

1.9.1 Assumptions

- You run Kubernetes across multiple providers/regions with GitOps as the default delivery mechanism.
- You have (or will create) separate EU and US tenant/team boundaries at minimum for customer data **and derived data** (logs/metrics/traces, backups, artifacts).
- You can centralize *policy authoring* but must keep *policy enforcement and data sinks* residency-scoped (EU stays EU, US stays US).
- You target SOC2-aligned controls (change management, access control, audit trails, incident response evidence) and **99.9%** availability.

1.9.2 Non-goals

- Building a single global “god” control plane that can change any cluster in any region without residency-aware guardrails.
- Achieving identical provider feature parity. You will standardize interfaces and accept provider-specific edges (captured via ADRs).
- Replacing provider IAM/KMS with Kubernetes-only constructs.

1.9.3 Objectives

- Provide consistent guardrails across providers without making operations brittle.
- Define policy-as-code, enforcement points, and escalation paths (deny vs audit).
- Enforce SOC2-friendly auditability: every change is attributable, reviewable, and reversible.
- Enforce EU/US data residency and encryption everywhere by default.

1.9.4 When to use/avoid

Use when

- You need to prevent policy drift across clusters/regions/providers and reduce blast radius.

- You must prove control effectiveness (SOC2): approvals, logs, retention, and access review.

Avoid when

- You cannot commit to operational ownership of policy lifecycle (testing, staging, rollout, break-glass). Policy without operations becomes an availability risk.
- Your “policy” is actually product logic (e.g., tenant entitlements). Keep that in application control plane, not admission controllers.

1.9.5 Architecture notes

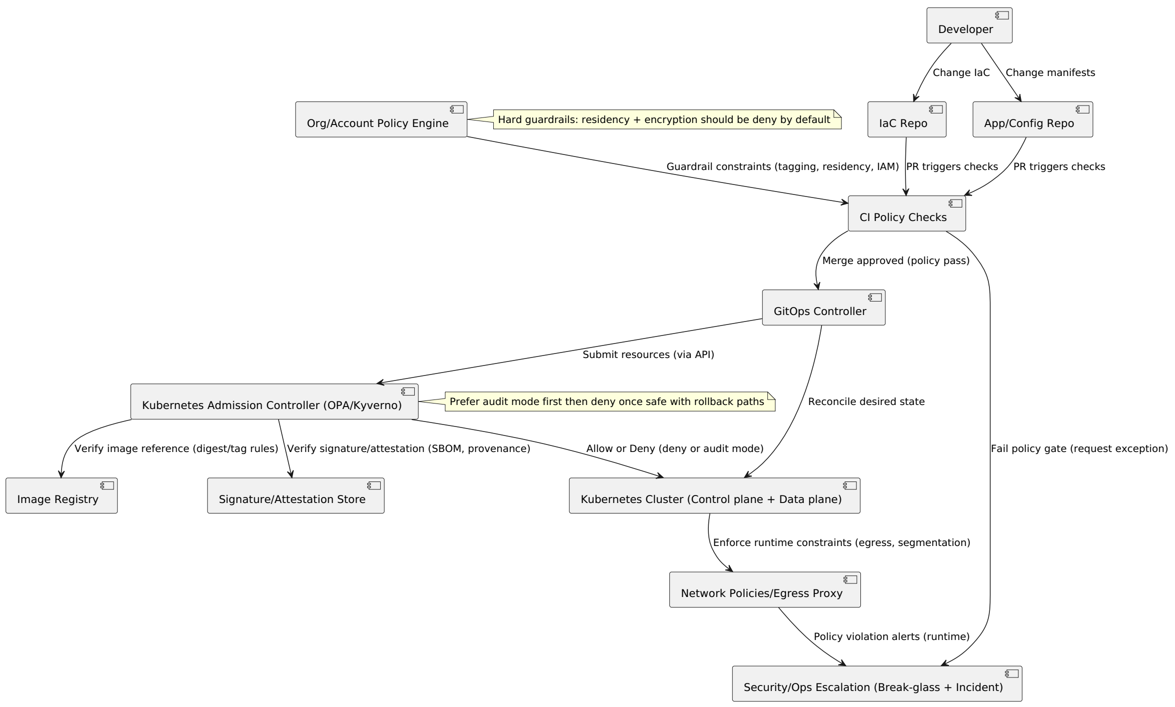
You should treat policy as a layered system with independent enforcement points. The operational rule: **fail closed only where you can recover quickly** (fast rollback + clear ownership), otherwise start in audit mode and graduate.

Layers and enforcement responsibilities

- **Provider org/account/project guardrails (control plane boundary)**
    - Prevent unsafe primitives: public buckets, open security groups, cross-residency replication, overly broad IAM.
    - Enforce baseline encryption (at rest, in transit) and mandatory audit logging.
- **CI policy checks for IaC and Kubernetes manifests (change boundary)**
    - Block merges that violate residency, encryption, tagging, IAM least privilege, or network egress rules.
    - Produce attestations tied to commits (evidence for SOC2).
- **GitOps reconciliation (delivery boundary)**
    - Ensures declared state and provides deterministic rollback (commit revert).
    - Removes “kubectl apply from laptop” except time-bound, logged break-glass.
- **Kubernetes admission control (cluster control plane boundary)**
    - Enforce namespace/tenant/team boundaries, image provenance, Pod Security, and workload placement rules.
    - Deny unsafe objects early (before they hit the data plane).
- **Runtime controls (data plane boundary)**
    - NetworkPolicies, egress proxy/allowlisting, and (optionally) service mesh policies.
    - Runtime controls are your last line of defense when admission misses something.

1.9.5.1 Diagram: Policy Enforcement Points Across the Stack (diagram_id: policy-enforcement-points)

Alt text: Policy enforcement chain from CI and org guardrails to Kubernetes admission and runtime controls.


Developer
Change IaC
Change manifests
Org/Account Policy Engine
Hard guardrails: residency + encryption should be deny by default
IaC Repo
App/Config Repo
Guardrail constraints (tagging, residency, IAM)
PR triggers checks
PR triggers checks
CI Policy Checks
Merge approved (policy pass)
GitOps Controller
Submit resources (via API)
Kubernetes Admission Controller (OPA/Kyverno)
Prefer audit mode first then deny once safe with rollback paths
Reconcile desired state
Verify image reference (digest/tag rules)
Verify signature/attestation (SBOM, provenance)
Allow or Deny (deny or audit mode)
Fail policy gate (request exception)
Image Registry
Signature/Attestation Store
Kubernetes Cluster (Control plane + Data plane)
Enforce runtime constraints (egress, segmentation)
Network Policies/Egress Proxy
Policy violation alerts (runtime)
Security/Ops Escalation (Break-glass + Incident)

1.9.6 Operational considerations

1.9.6.1 1) Org/account/project structure and "SCP-equivalents"

Objective: reduce blast radius and prevent unsafe primitives before they reach Kubernetes.

Recommended structure (neutral pattern)

- Separate by **provider × region × residency boundary (EU/US) × environment (prod/non-prod)**.
- Define tenant/team boundaries at the account/project level for high-risk workloads; otherwise enforce within clusters using namespaces + admission + IAM.

Tradeoffs / failure modes

- **Tradeoff:** more accounts/projects improves blast radius but increases operational overhead (quotas, billing, IAM sprawl).

- **Failure mode:** “shared services” accounts accidentally become cross-residency bridges (e.g., central logging in US receiving EU logs). Treat shared services as **residency-scoped**.

Controls you should enforce at this layer

- Default encryption at rest (storage, databases, registries) and TLS in transit.
- Mandatory audit logging enabled and retained per residency boundary.
- Deny cross-region/cross-residency replication for data and derived data unless explicitly approved via ADR.

Security requirements

- Least privilege via scoped roles and separation of duties (platform vs app teams).
- Rotation/expiry for any long-lived credentials; prefer workload identity.

1.9.6.2 2) Policy-as-code and enforcement points (CI + GitOps + admission)

Policy authoring

- Keep policies in version control with reviews (SOC2 change management evidence).
- Separate:
  - **Baseline policies** (platform-owned, rarely changed, deny-capable)
  - **Application policies** (team-owned, mostly audit-first)

Deny vs audit progression (opinionated)

- Start new policies in **audit** mode with alerting and dashboards.
- Promote to **deny** only after:
  - false positive rate is low,
  - rollback is proven,
  - an on-call owner is named.

Failure modes

- Admission controller outage becomes a cluster-wide deploy outage. You must:
  - run admission controllers with HA,
  - set resource limits,
  - monitor webhook latency/error rates,
  - design emergency bypass (break-glass) with tight scoping and audit logs.

1.9.6.3 3) Kubernetes admission control: OPA Gatekeeper vs Kyverno

Dimension	Gatekeeper (OPA)	Kyverno
Policy language	Rego (powerful, steeper learning curve)	YAML-native rules (easier for many teams)
Common fit	Complex constraints, shared libraries	K8s-native mutation/validation, quick wins
Operational risk	Miswritten Rego can be hard to debug	Rule sprawl and unintended mutation risk
Recommendation	Use when you need expressive, reusable logic	Use when you want faster adoption and simpler policies

Minimum admission controls (baseline)

- Pod Security baseline enforcement (prefer restricted where feasible).
- Disallow privileged pods/hostPath/hostNetwork unless explicitly approved.
- Require resource requests/limits (avoid noisy neighbor outages).
- Require readiness/liveness probes for services (availability protection).
- Require images by digest (or at least allowlist registries + enforce signing verification).
- Enforce namespace labels for tenant/team ownership (used by policy and audit queries).

1.9.6.4 4) Image provenance, signing, and SBOM requirements

Policy goals

- Only run images you can trace to a build pipeline (provenance).
- Prevent mutable tag attacks; prefer **immutable digests**.
- Require SBOM generation and retention per residency boundary.

Enforcement pattern

- CI produces:
  - image (pushed to residency-scoped registry),
  - SBOM,
  - signature/attestation.
- Admission verifies:
  - registry allowlist,
  - signature/attestation presence,

    - (optionally) SBOM/provenance checks by policy.

Failure modes

- Registry outage blocks deploys. Mitigate with:
    - multi-AZ registry (per residency),
    - caching where compliant (don’t cross EU/US),
    - clear break-glass rules (time-bound, logged).

Rotation/expiry

- Signing keys must be rotated; prefer short-lived identities from workload identity rather than long-lived secrets in CI.

1.9.6.5 5) Data residency and workload placement constraints

Residency is a **hard boundary**: prevent cross-boundary replication/log shipping for customer data and derived data.

Mechanisms

- **At CI:** block manifests that reference non-resident registries, endpoints, or telemetry sinks.
- **At admission:** enforce node/region constraints via labels and allowed topologies; block cross-residency endpoints in config (where detectable).
- **At runtime:** egress proxy allowlists per residency; deny unknown external endpoints.

Failure modes

- Silent residency drift via “helpful” shared observability or backups. You must tag and enforce residency on:
    - log/metric/trace sinks,
    - object storage buckets,
    - backup targets,
    - registries/artifact stores,
    - KMS keys.

1.9.6.6 6) Audit logging (control plane + data plane), retention, SIEM integration

What to log

- **Control plane:** org/account/project policy decisions, IAM changes, GitOps sync events, Kubernetes API audit logs, admission decisions (deny/audit).
- **Data plane:** egress proxy logs, network policy violations (where available), workload identity token issuance events, container runtime alerts.

SOC2-oriented practices

- Immutable retention (WORM-capable storage) per residency boundary.
- Time sync, log integrity controls, and documented access reviews.
- SIEM integration must be **residency-scoped** (EU-to-EU, US-to-US). If you need global security visibility, aggregate only non-sensitive metadata with explicit classification and ADR.

Availability tradeoff

- If audit logging pipelines fail closed, you can take outages. Prefer:
    - local buffering + retry,
    - alerts on lag/drop,
    - explicit “degraded logging” runbooks with time limits and approvals.

1.9.7 Risks and mitigations (major designs)

Risk	How it fails	Mitigation
Policy drift across clusters	Different rules per cluster cause inconsistent behavior	GitOps for policy bundles; policy version pinning; regular conformance tests
Admission controller becomes an outage amplifier	Webhook latency/ errors block deploys	HA deployment, SLOs for webhook latency, progressive rollout, break-glass with audit
Residency violation via derived data	Logs/traces/backups shipped cross-boundary	Residency-scoped sinks, egress allowlists, CI checks for endpoints, residency-scoped KMS keys

Risk	How it fails	Mitigation
Over-broad IAM	Lateral movement and large blast radius	Least privilege, workload identity, periodic access reviews, short-lived creds
“Audit-only forever”	Controls exist but aren’t effective	Explicit policy maturity gates: audit → warn → deny with owner and rollback

---

1.9.8 Checklist

- [ ] Provider guardrails deny cross-residency replication for data + derived data by default.
- [ ] CI policy checks gate IaC and manifests; failures have an exception workflow (ADR + time-bound).
- [ ] GitOps is the only steady-state delivery path; break-glass is logged, time-bound, and reviewed.
- [ ] Admission enforces baseline security + provenance (registry allowlist, signing/attestation, Pod Security).
- [ ] Runtime egress is controlled (egress proxy/allowlists) and NetworkPolicies enforce tenant/team boundaries.
- [ ] Audit logs collected for control plane and data plane; retention meets SOC2; SIEM is residency-scoped.
- [ ] Key material and credentials rotate; long-lived secrets avoided via workload identity.
- [ ] Policy rollout process exists: staging, canary, deny-mode promotion criteria, and rollback steps.

1.10 Supply Chain: Build, Artifact, and Registry Strategy (s10)

1.10.1 Objectives

- You ship **portable OCI artifacts** across clusters/regions/providers while acknowledging edge coupling (IAM, KMS, egress, scanning, admission control).
- You enforce **immutable artifact promotion** (dev→stage→prod) with SOC2-grade auditability and rollback.

- You meet hard constraints: **EU/US data residency**, **encryption everywhere**, and **99.9% availability** (including supply-chain dependencies).

1.10.2 When to use / avoid

Use when

- You run multiple Kubernetes clusters across regions/providers and need repeatable, auditable delivery with controlled blast radius.
- You must prove: who built what, from which source, with which dependencies, and why it was deployed (SOC2 evidence).

Avoid (or down-scope) when

- You only want “portability.” Most coupling moves to IAM/KMS, registries, and admission controls; the complexity isn’t free.
- You can’t operate 24/7 on the supply chain (registry outages, signing key incidents, scanner false positives). A brittle pipeline becomes your new single point of failure.

1.10.3 Architecture notes

1.10.3.1 Assumptions

- Your clusters support **deployment-time policy enforcement** (e.g., OPA-based admission) and you can integrate with provider IAM and KMS.
- You can run separate **EU and US supply-chain lanes** (CI workers, registries, signing keys, logs) to enforce residency.

1.10.3.2 Non-goals

- Designing cross-region replication for stateful application data (handled in data/storage and DR sections).
- Making a single pipeline that can freely move artifacts across residency boundaries. You should treat that as a policy violation by default.

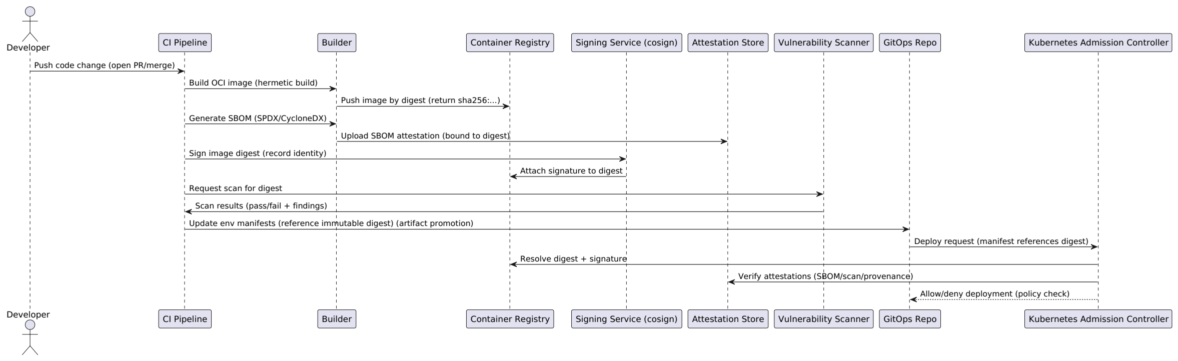
1.10.3.3 Reference architecture (neutral descriptive)

- **Source → CI Pipeline → Builder → Registry → Signing/Attestation → Scanner → GitOps Repo → Admission-controlled deploy**
- All deployable artifacts are addressed by **immutable digests** (not tags) and promoted via Git changes (GitOps), preserving an audit trail.

- EU and US are **separate control boundaries**:
  - Separate registries (or strictly partitioned tenants), separate KMS keys, separate signing identities, separate audit logs.
  - If you replicate, replicate **within** EU or **within** US only.

1.10.3.4 Diagram: container supply chain flow

Alt text: Supply chain flow from build to signed image promotion with deployment-time verification. (diagram_id: supply-chain-signing-promotion)


Developer
CI Pipeline
Builder
Container Registry
Signing Service (cosign)
Attestation Store
Vulnerability Scanner
GitOps Repo
Kubernetes Admission Controller
Push code change (open PR/merge)
Build OCI image (hermetic build)
Push image by digest (return sha256:...)
Generate SBOM (SPDX/CycloneDX)
Upload SBOM attestation (bound to digest)
Sign image digest (record identity)
Attach signature to digest
Request scan for digest
Scan results (pass/fail + findings)
Update env manifests (reference immutable digest) (artifact promotion)
Deploy request (manifest references digest)
Resolve digest + signature
Verify attestations (SBOM/scan/provenance)
Allow/deny deployment (policy check)
Developer
CI Pipeline
Builder
Container Registry
Signing Service (cosign)
Attestation Store
Vulnerability Scanner
GitOps Repo
Kubernetes Admission Controller

1.10.4 Operational considerations

1.10.4.1 Build strategy (remote builders, BuildKit, hermetic builds)

- Prefer **remote builders** per residency boundary (EU builders for EU artifacts; US builders for US artifacts).
- Use **BuildKit** (or equivalent) with:
  - Reproducible builds where feasible (pin base images by digest; pin package versions).
  - Controlled egress: allowlist registries and package mirrors; block arbitrary internet by default.
- **Tradeoff:** hermetic builds reduce “works on my machine” and supply-chain drift, but increase up-front work (mirrors, caching, and dependency pinning).

1.10.4.2 Registry strategy (multi-region replication + access controls)

- Store artifacts in **OCI-compatible registries** per provider/region with:
  - **Encryption at rest** (KMS-backed) and **TLS in transit**.
  - **Multi-region within the same residency boundary** to hit 99.9% supply availability goals.
  - Strict IAM: builders can push; deployers can pull; humans rarely need write access.

- **Failure mode:** registry outage blocks deploys and scaling (image pulls). Mitigate with:
    - Regional replication, pull-through caches per cluster/region, and pre-pull for critical workloads.
    - Runbooks for “registry brownout” (freeze promotions; rely on already-cached images).

1.10.4.3 Signing and attestations (cosign), SBOM generation

- Make signing and attestations mandatory for every promoted digest:
    - Signatures bind **identity → digest**.
    - Attestations bind **metadata → digest** (SBOM, provenance, scan results).
- Enforce at deploy time via admission:
    - Deny unsigned images.
    - Deny missing/expired attestations.
    - Deny images signed by non-approved identities (least privilege).
- **Rotation/expiry:** treat signing identities as production credentials:
    - Rotate signing keys/certs on a schedule; expire old identities; keep verification trust roots updated via change control.
    - Log all signing events to residency-scoped audit logging (SOC2 evidence).

1.10.4.4 Promotion model (dev→stage→prod using immutable digests)

- Promotion is **not “rebuild in prod.”** You promote the *same digest* through environments.
- Use GitOps to promote by updating environment manifests to the digest:
    - dev: fast feedback, broader allowlists
    - stage: production-like policy gates
    - prod: strict policy + progressive delivery
- **Rollback:** revert Git to the previous known-good digest; do not “retag latest.”

1.10.4.5 Air-gapped or restricted networks

- Use “import lanes” per residency boundary:
    - Mirror base images and dependencies into an internal registry.
    - Scan and sign at the boundary before artifacts can enter the restricted network.
- **Tradeoff:** improved control and fewer surprise egress failures, but higher ops burden (mirror freshness, CVE response, cache sizing).

1.10.4.6 Risks and mitigations (major design)

Risk	Why it happens	Mitigation you should implement
Residency violation via artifacts/telemetry	CI logs, SBOMs, scan metadata, or registry replication crosses EU/US	Separate lanes: region-scoped CI, registries, signing keys, and audit logging; policy checks preventing cross-boundary pushes
Common tooling becomes a shared failure domain	One global registry/ scanner/signing service outage blocks all deploys	Per-residency redundancy; cache images; define “deploy freeze” procedures; keep verification keys locally available in clusters
Policy drift blocks deploys unexpectedly	Admission policies evolve faster than teams	Version policies; stage enforcement (audit→enforce); canary policy changes; documented break-glass with time-bound approvals
Scanner false positives / noisy gating	CVE feeds fluctuate; base images change	Gate on severity + exploitability where justified; pin base images by digest; exception workflow with expiry and audit trail

1.10.5 Checklist

- [ ] You use **OCI images referenced by digest** everywhere in prod manifests.
- [ ] You have **EU and US separated lanes** (CI runners, registries, signing identities, audit logs).
- [ ] Registry access is least-privilege: builders push, clusters pull, humans read-only by default.
- [ ] Images are **signed** and have **attestations** (SBOM + provenance + scan result) bound to the digest.
- [ ] Kubernetes admission enforces signature/attestation verification before pods are allowed.
- [ ] Encryption is enforced: TLS for pulls/pushes; KMS-backed encryption at rest; key rotation/expiry is defined.
- [ ] Promotion is GitOps-based with an auditable trail and a deterministic rollback (Git revert to prior digest).

- [ ] You have documented runbooks for registry/scanner/signing outages, including deploy freeze and recovery steps.

1.11 Deployment Across Clouds: GitOps and Progressive Delivery (s11)

1.11.1 Objectives

- Deliver changes consistently across many clusters/regions/providers with an auditable trail (SOC2-friendly).
- Reduce blast radius via controlled placement, staged rollout, and fast rollback.
- Enforce hard EU/US data residency boundaries for *artifacts* and *telemetry*, not just application data.
- Preserve 99.9% availability by gating releases on SLO signals and keeping the control plane resilient.

1.11.2 When to use / avoid

Use when

- You run multiple Kubernetes clusters across providers/regions and need consistent change control, drift detection, and evidence retention.
- You need deterministic artifact promotion (by OCI digest) and approvals per environment/tenant/team boundary.
- You must prove *what changed, who approved it, and where it ran* (SOC2).

Avoid when

- You can't staff on-call to operate the *deployment control plane* (Git, GitOps controllers, policy, registries). Multi-cloud adds failure modes.
- Your workloads are tightly coupled to provider-native data services and require frequent stateful schema migrations without robust rollback.
- You're doing multi-cloud primarily for "portability"; GitOps won't remove coupling to IAM, ingress/load balancers, DNS, KMS, or observability.

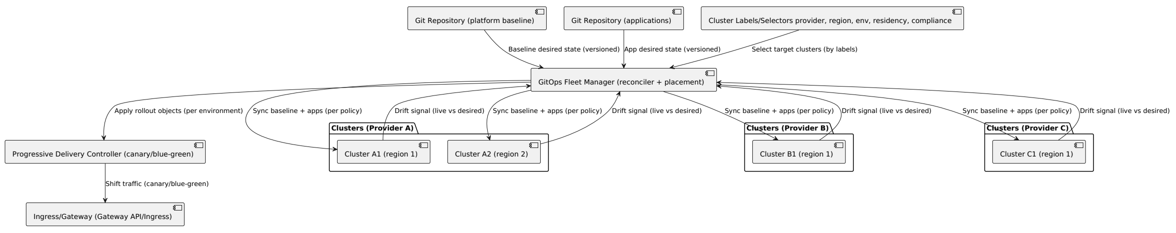
1.11.3 Architecture notes (reference architecture)

1.11.3.1 Core pattern: layered reconciliation + placement

- **Two Git repositories (or two top-level dirs with strict separation):**
  - **Platform baseline repo**: cluster add-ons and guardrails (CNI, CSI, policy engines, observability agents, ingress/gateway controllers).
  - **Applications repo**: workload manifests and progressive delivery objects.
- **Independent reconciliation loops**:
  - Baseline reconciles at the *cluster* boundary.
  - Apps reconcile at the *namespace/tenant* boundary.
- **Fleet placement** uses **cluster labels/selectors** (e.g., `provider`, `region`, `env`, `residency`, `compliance_tier`) to avoid “snowflake” targeting logic embedded in pipelines.

1.11.3.2 Diagram: GitOps multi-cluster placement (gitops-multicluster-placement)

Alt text: GitOps multi-cluster placement using labels and separate baseline vs application layers.


Git Repository (platform baseline)
Git Repository (applications)
Cluster Labels/Selectors provider, region, env, residency, compliance
Baseline desired state (versioned)
App desired state (versioned)
Select target clusters (by labels)
GitOps Fleet Manager (reconciler + placement)
Apply rollout objects (per environment)
Sync baseline + apps (per policy)
Drift signal (live vs desired)
Sync baseline + apps (per policy)
Drift signal (live vs desired)
Sync baseline + apps (per policy)
Drift signal (live vs desired)
Sync baseline + apps (per policy)
Drift signal (live vs desired)
Clusters (Provider A)
Cluster A1 (region 1)
Cluster A2 (region 2)
Clusters (Provider B)
Cluster B1 (region 1)
Clusters (Provider C)
Cluster C1 (region 1)
Progressive Delivery Controller (canary/blue-green)
Shift traffic (canary/blue-green)
Ingress/Gateway (Gateway API/Ingress)

1.11.3.3 Drift detection and reconciliation boundaries (opinionated)

- **Drift detection is mandatory**, but **auto-remediation is conditional**:
  - Auto-reconcile **platform baseline** only when you have strong testing and staged rollout; otherwise, you can brick multiple clusters at once.
  - Auto-reconcile **apps** within a tenant/team boundary, but enforce progressive delivery gates.
- **Define “break-glass” boundaries**:
  - Emergency patches may be applied out-of-band, but must be reconciled back into Git with an incident/ADR reference to satisfy SOC2 evidence expectations.

1.11.3.4 Progressive delivery integration

- Progressive delivery controller should own **traffic shifting** via **Ingress/ Gateway** resources, not ad-hoc scripts.
- Rollouts should be gated on:
    - Availability/error-rate SLO signals (99.9% target; protect your error budget).
    - Key business health checks (synthetic probes) scoped per residency boundary.

1.11.4 Operational considerations

1.11.4.1 SOC2 + residency + encryption everywhere requirements

- **Least privilege**:
    - GitOps controllers get cluster-scoped permissions only for baseline components; apps operate with namespace-scoped RBAC per tenant/ team.
    - Use **workload identity** (avoid long-lived secrets) for:
        - Pulling OCI artifacts (images/Helm charts).
        - Writing deployment events/audit logs.
- **Audit logging and evidence retention**:
    - Retain and correlate: Git commit, PR approval, GitOps sync event, Kubernetes API audit logs, and progressive delivery decision logs.
    - Keep audit logs **residency-scoped** (EU logs stay in EU; US logs stay in US), including derived data (metrics/traces).
- **Encryption and rotation/expiry**:
    - Encrypt at rest (Git secrets store, registry, artifact store, log storage) with residency-scoped keys.
    - Enforce TLS for GitOps and controller webhooks; rotate controller credentials and signing keys on a defined schedule.

1.11.4.2 Common failure modes (and what you do about them)

- **Common control plane outage** (Git provider, fleet manager, registry) can stall *all* deployments across providers.
    - Mitigation: residency-scoped replicas, cached artifact pulls, and the ability for clusters to continue running without the GitOps control plane.

- **Bad baseline rollout** can break networking/ingress across many clusters simultaneously.
    - Mitigation: canary baseline changes to a small cluster set (labels like `baseline_ring=0/1/2`), plus fast rollback via Git revert.
- **Residency leakage through artifacts/telemetry** (e.g., EU cluster pulling US-only image or exporting traces to US).
    - Mitigation: allowlisted egress, residency-scoped registries and observability sinks, CI policy checks that fail builds if targets violate residency labels.

1.11.4.3 Provider-variant comparison (you standardize the interface, not the implementation)

Capability	What you standardize (portable contract)	Provider-specific “edge” (allowed variance)	Failure mode to watch
GitOps reconciliation	Kustomize/Helm conventions, repo layout, sync windows, health checks	Controller hosting model (in-cluster vs management cluster)	Fleet manager outage impacts deployments
Artifact promotion	OCI digest pinning, signed artifacts, staged environments	Registry replication mechanics per provider/region	Cross-residency replication accident
Traffic shifting	Gateway API/ Ingress semantics, rollout CRDs, SLO gates	Load balancer behavior, health probe quirks	Partial cutover due to LB differences
Identity	Workload identity mapping per service account, no static creds	Underlying IAM primitives differ	Over-permissioned roles, token expiry issues
Policy	OPA/Gatekeeper/ Kyverno constraints, admission rules	Provider org policy equivalents	Drift between cluster and org-level controls

1.11.5 Risks and mitigations (major design callouts)

- **Risk: operational overload from too many clusters and too much autonomy**
  - Mitigation: fewer, larger clusters per residency boundary; enforce tenant/team boundaries with namespaces/RBAC/network policy.
- **Risk: progressive delivery depends on telemetry that violates residency**
  - Mitigation: keep metrics/logs/traces pipelines region/residency-scoped; avoid global aggregations that copy raw events across boundaries.
- **Risk: GitOps becomes a single point of failure**
  - Mitigation: separate management control plane from workload clusters; stage changes; define “deploy freeze” modes and manual break-glass.

1.11.6 Checklist

- [ ] Repo layout separates **platform baseline** from **applications** with independent reconciliation.
- [ ] Cluster labels include `provider`, `region`, `env`, `residency`, `compliance_tier`; placement rules are declarative and reviewed.
- [ ] Artifact promotion uses **immutable OCI digests**; approvals are recorded; signatures/SBOM attached per SOC2 expectations.
- [ ] Progressive delivery is mandatory for user-facing services; rollback is automated and tested.
- [ ] Least-privilege RBAC is enforced; workload identity is used; no long-lived secrets for GitOps controllers/apps.
- [ ] Audit logs are enabled for Git, GitOps controller actions, Kubernetes API, and rollout decisions; retention meets SOC2 and is **residency-scoped**.
- [ ] Egress controls prevent cross-residency artifact pulls and telemetry export; encryption in transit/at rest with rotation/expiry is verified.
- [ ] Baseline changes are rolled out in rings with clear rollback (Git revert) and post-change verification gates aligned to 99.9% SLOs.

1.12 Service-to-Service Communication and Service Mesh Options (s12)

1.12.1 Objectives

- Help you choose patterns for **east-west** (service-to-service) traffic across **clusters** and **providers**.
- Make the “mesh vs no mesh” decision explicit, including **operational costs**, **failure modes**, and **blast radius**.
- Define **mTLS**, identity, and **trust domain** boundaries that satisfy **SOC2**, **EU/US residency**, **encryption everywhere**, and a **99.9% SLO** target.

1.12.2 When to use/avoid

Use a service mesh when you need (and will operate) all of the following:

- **Default-on mTLS** with strong workload identity and consistent authZ policy across namespaces/teams.
- **Fine-grained L7 policy** (service-level allowlists, request attributes) enforced close to workloads.
- **Consistent telemetry** (traces/metrics) for east-west calls across clusters/providers without per-app SDK sprawl.
- **Progressive delivery hooks** (traffic shifting, retries/timeouts/circuit breaking) that you want standardized.

Avoid (or delay) a service mesh when:

- You can meet requirements with **Kubernetes NetworkPolicy + TLS at the app** and a small set of **gateways**.
- You cannot staff **Day-2 operations** (cert rotation, control plane upgrades, policy lifecycle, incident debugging).
- Your traffic is dominated by **cross-region/provider latency**, making L7 retries/timeouts expensive and failure-prone.
- You cannot tolerate the **data plane overhead** (CPU/memory, connection churn) or the added failure modes.

Tradeoff (opinionated): In multi-cloud, a mesh is rarely “set and forget.” If you can’t commit to ownership, you’re usually better off with **explicit gateway-mediated cross-cluster calls** + **app-managed TLS** + **tight egress controls**, then adopt mesh selectively for high-value domains.

1.12.3 Architecture notes

1.12.3.1 Baseline patterns (from simplest to most standardized)

Pattern	How it works	Best for	Main drawbacks / failure modes
1) Direct service calls + app TLS	Apps call endpoints via DNS/IP; you manage TLS and auth in code	Small footprint, few services, strong platform SDK maturity	Inconsistent policy; uneven telemetry; certificate sprawl; harder audits
2) Gateway-mediated cross-cluster	In-cluster calls stay local; cross-cluster goes through **east-west gateways**	Clear blast radius; simpler routing; good for residency boundaries	Gateway becomes a chokepoint; needs capacity and DDoS/ backpressure planning
3) Per-cluster mesh (no multi-cluster)	Mesh inside each cluster; cross-cluster is “north-south” via gateways	Good stepping stone; isolates complexity per cluster	Split policy model; uneven identity semantics across clusters
4) Multi-cluster mesh (multi-network)	Mesh spans clusters via gateways; service discovery and identity federated per trust domain	Standardized mTLS/policy/ telemetry across clusters	Highest operational cost; most failure modes; tight change control required

1.12.3.2 Trust domains and residency boundaries

- Treat **EU and US** as separate **trust domains** unless you have a formally reviewed cross-boundary data flow and evidence that **telemetry, artifacts, and backups** obey residency constraints.
- You can still use the same tooling, but you should run **separate CAs/roots, separate policy distribution, separate observability backends/sinks** per boundary.

- Cross-boundary calls should be **explicitly routed** via gateways with **allowlists**, and you should assume these links will fail (partial outages and asymmetric routing are common).

1.12.3.3 Service discovery and DNS options (cross-cluster)

- **Split-horizon DNS**: same service name resolves differently per cluster/region; good for locality, but increases troubleshooting complexity.
- **Explicit FQDNs per cluster/region**: simplest to reason about and debug; more app config.
- **Mesh service discovery** (if you adopt it): reduces app config but increases coupling to mesh control plane availability.

1.12.3.4 Risks and mitigations (major designs)

- **Risk: policy drift across clusters/providers → Mitigation:** GitOps-managed policy-as-code; admission control for required mTLS/authZ; periodic conformance tests.
- **Risk: mesh control plane outage impacts rollout/telemetry/policy → Mitigation:** ensure data plane fails "open/closed" intentionally per policy type; keep cross-cluster comms possible via gateway fallback paths; test control-plane-down scenarios.
- **Risk: certificate/identity failures cause widespread outage → Mitigation:** short-lived certs with automated rotation; monitor expiry; staged root rotations; break-glass procedures with time-bound access and audit logging.
- **Risk: retries amplify incidents across clouds → Mitigation:** conservative retry budgets, per-hop timeouts, circuit breaking, and load shedding; prefer idempotent APIs; set global caps.

1.12.4 Operational considerations

1.12.4.1 Reliability and failure handling (99.9% target)

- Prefer **local-first routing**: keep calls within a cluster/region when possible; reserve cross-cloud calls for explicit dependencies.
- Put **east-west gateways** in HA mode and scale them like critical infra (HPA, PDBs, surge capacity).
- Design for **partial connectivity failures**:
    - gateways unreachable but clusters healthy
    - DNS resolving stale endpoints

    - asymmetric routing across cross-cloud links
- Define and enforce **timeouts** and **retry budgets**; uncontrolled retries are a common multi-cloud outage multiplier.

1.12.4.2 Security (SOC2-aligned)

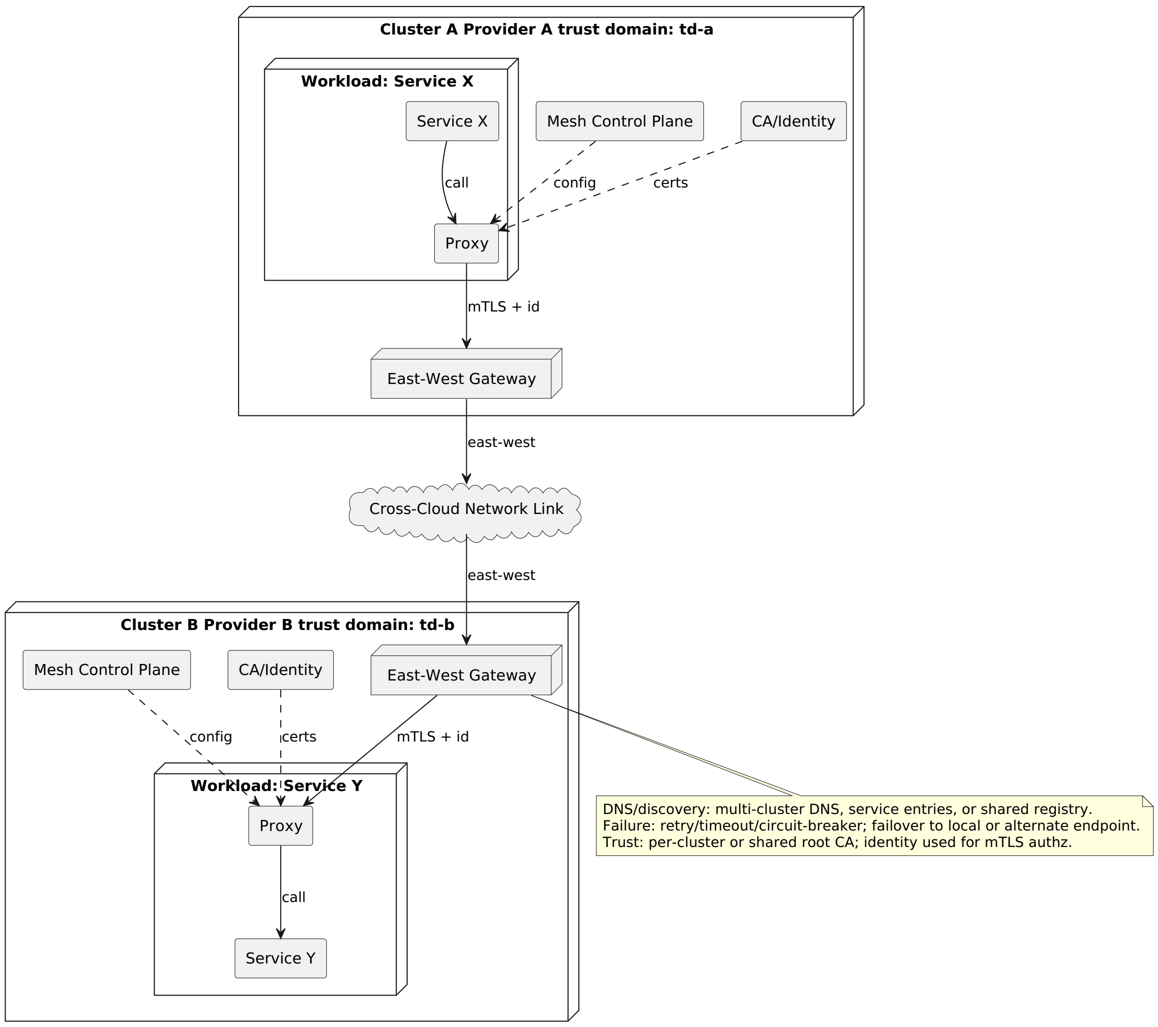
- **Least privilege:** service identity must map to least-privilege permissions and minimal allowed peers (deny-by-default).
- **Audit logging:** log policy changes (GitOps), gateway access logs, and mesh authZ decisions where feasible; keep logs **residency-scoped**.
- **Rotation/expiry:** use short-lived workload identities/certs; document and test root/intermediate CA rotation; avoid long-lived tokens/secrets in namespaces.

1.12.4.3 Cost and performance

- Sidecar/ambient proxies add **CPU/memory** overhead and can reduce node density; plan capacity headroom.
- Cross-cloud L7 features (retries, mTLS handshakes) add **latency**; measure p95/p99 and watch tail amplification.
- Gateways concentrate traffic; plan for **egress costs** and per-provider interconnect billing.

1.12.5 Reference diagram: Multi-Cluster Service Mesh Across Clouds (diagram_id: service-mesh-multicluster)

Alt text: Cross-cloud multi-cluster service mesh using east-west gateways and mTLS identity.



Cluster A Provider A trust domain: td-a
Workload: Service X
Service X
Mesh Control Plane
CA/Identity
call
config
certs
Proxy
mTLS + id
East-West Gateway
east-west
Cross-Cloud Network Link
east-west
Cluster B Provider B trust domain: td-b
Mesh Control Plane
CA/Identity
East-West Gateway
config
certs
mTLS + id
Workload: Service Y
Proxy
call
Service Y
DNS/discovery: multi-cluster DNS, service entries, or shared registry.
Failure: retry/timeout/circuit-breaker; failover to local or alternate endpoint.
Trust: per-cluster or shared root CA; identity used for mTLS authz.

1.12.6 Checklist

- [ ] You have an ADR that selects one of: **gateway-only**, **per-cluster mesh**, or **multi-cluster mesh**, with explicit **tradeoffs** and **exit criteria**.
- [ ] EU and US are treated as **hard residency boundaries** for **telemetry, backups, and artifacts**, not just application data.
- [ ] Cross-cluster calls traverse **east-west gateways** with **allowlists**, capacity planning, and clear **failure behavior**.
- [ ] **mTLS everywhere** is enforced with short-lived certs and documented **rotation/expiry** procedures.
- [ ] Policies are **GitOps-managed** with audit trails; break-glass access is time-bound and logged.
- [ ] Retry/timeout/circuit-breaking defaults are set to prevent **retry storms** across providers.
- [ ] You have game days for: **gateway outage**, **mesh control plane outage**, **CA rotation**, and **cross-cloud link degradation**.

1.13 Data and Storage: Stateful Workloads in Multi-Cloud

1.13.1 Objectives

- Define supported stateful data patterns and their failure expectations (RPO/RTO, consistency, blast radius).
- Provide replication and consistency guidance that survives real multi-cloud failure modes (network partitions, IAM/KMS outages, operator error).
- Keep you compliant: SOC2 evidence, EU/US data residency, encryption everywhere, and auditability without turning your data plane into a science project.

1.13.2 When to use/avoid

When to use

- You have a hard requirement for **EU/US residency separation** (including backups and telemetry) and you need a **documented, tested DR story** across regions/providers.
- Your availability target is **99.9%** and you can accept that stateful DR is usually about **bounded downtime + bounded data loss**, not magical “zero downtime” across clouds.
- You can staff **24/7 operations** (or on-call with tight response) and run regular DR tests.

When to avoid

- Your primary objective is “portability.” Stateful systems are coupled to **storage semantics, KMS, IAM, network latency, and operational tooling**.
- You cannot tolerate any data loss but also cannot tolerate added write latency (multi-cloud synchronous replication will punish you).
- You don’t have disciplined change management (GitOps + policy-as-code). Stateful DR fails most often due to **humans + drift**.

1.13.3 Architecture notes

1.13.3.1 Assumptions

- You enforce hard **EU and US residency boundaries** for: primary data, backups, artifacts, logs/metrics/traces, and audit logs.

- You use **encryption in transit (TLS/mTLS where applicable)** and **encryption at rest** with KMS-backed keys; keys have **rotation/expiry** and access is **least privilege**.
- You have a consistent multi-cloud control-plane approach (GitOps reconciliation, policy distribution, centralized—yet residency-scoped—audit logging).

1.13.3.2 Non-goals

- Designing a single universal storage layer that behaves identically across providers.
- Promising “active-active for everything.” Most databases cannot do this safely without application-level constraints.
- Making cross-cloud synchronous replication the default; it’s a specialized tool with sharp edges.

1.13.3.3 Supported stateful patterns (opinionated)

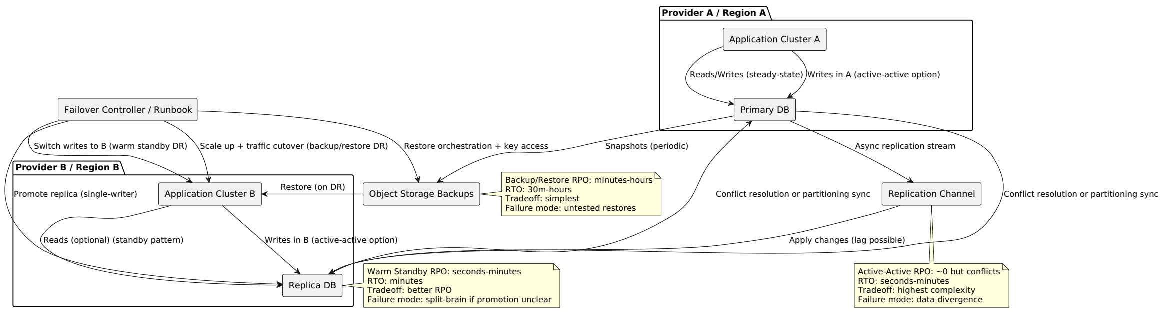
You should standardize on a small number of patterns and force teams to choose explicitly via ADR.

Pattern	Default recommendation	Typical RPO/RTO	Operational difficulty	Common failure mode
Backup/ restore (snapshots + object storage)	**Baseline for most teams**	RPO: hours–minutes (schedule-dependent) / RTO: tens of minutes–hours	Low–Medium	Restores fail due to untested procedures, missing keys, or incompatible versions
Warm standby (single-writer, async replica)	**Preferred for Tier-1 stateful**	RPO: seconds–minutes / RTO: minutes	Medium–High	Split-brain due to unclear promotion authority; replication lag surprises

Pattern	Default recommendation	Typical RPO/RTO	Operational difficulty	Common failure mode
Active-active (multi-writer)	**Use only with strong constraints** (partitioning or conflict resolution)	RPO: ~0 (but conflicts exist) / RTO: seconds–minutes	High	Data divergence/ conflicts; “eventual correctness” not understood by consumers

1.13.3.4 Diagram: Stateful DR patterns

Caption (alt text): Three DR options for stateful services across clouds with differing RPO/RTO tradeoffs. (diagram_id: `stateful-dr-patterns`*)*


Provider A / Region A
Application Cluster A
Reads/Writes (steady-state)
Writes in A (active-active option)
Primary DB
Failover Controller / Runbook
Switch writes to B (warm standby DR)
Scale up + traffic cutover (backup/restore DR)
Restore orchestration + key access
Snapshots (periodic)
Async replication stream
Provider B / Region B
Promote replica (single-writer)
Application Cluster B
Restore (on DR)
Object Storage Backups
Backup/Restore RPO: minutes-hours
RTO: 30m-hours
Tradeoff: simplest
Failure mode: untested restores
Conflict resolution or partitioning sync
Replication Channel
Conflict resolution or partitioning sync
Reads (optional) (standby pattern)
Writes in B (active-active option)
Apply changes (lag possible)
Replica DB
Warm Standby RPO: seconds-minutes
RTO: minutes
Tradeoff: better RPO
Failure mode: split-brain if promotion unclear
Active-Active RPO: ~0 but conflicts
RTO: seconds-minutes
Tradeoff: highest complexity
Failure mode: data divergence

1.13.3.5 Storage layer reality: CSI and StorageClass variance

- **CSI drivers are not portable APIs** in practice. Even if Kubernetes objects look similar, behaviors vary: snapshot semantics, expansion rules, volume attachment limits, failure handling, and performance profiles.
- Treat `StorageClass` as a **provider/region-specific interface** hidden behind a platform-owned abstraction:
  - You publish “blessed” classes like `fast-ssd`, `throughput-hdd`, `encrypted-default` per cluster/region/provider.
  - You block ad-hoc classes via policy (OPA) to reduce drift and SOC2 review scope.

1.13.3.6 Replication and consistency choices

- **Async replication** is the default across clouds. It’s survivable and you can reason about it (RPO is non-zero; you measure replication lag).
- **Sync replication across clouds** is almost always a latency and availability tax:
    - Latency: cross-cloud RTT inflates commit time.
    - Availability: partitions force you to choose consistency or availability; many teams end up “down everywhere” during partial failures.

1.13.3.7 Data residency boundaries (EU/US)

- Keep EU and US data planes **logically and cryptographically isolated**:
    - Separate KMS keys per residency boundary; no cross-boundary decrypt permissions.
    - Separate object storage buckets/containers per boundary for backups and exports.
    - Separate observability sinks (logs/metrics/traces) per boundary; avoid “global” log aggregation that quietly violates residency.

1.13.4 Operational considerations

1.13.4.1 Risks and mitigations (major)

- **Risk: split-brain during failover (warm standby).**
  Mitigations: single authority for promotion (runbook + automation), fencing (disable writes on old primary), and forced DNS/traffic control sequencing.
- **Risk: residency violation via backups/telemetry replication.**
  Mitigations: residency-scoped pipelines, allowlisted destinations, policy checks in CI, and residency-scoped keys that cannot decrypt outside boundary.
- **Risk: restores fail (keys, versions, permissions).**
  Mitigations: quarterly restore tests, versioned runbooks, least-privilege break-glass with audit logging, and “restore to isolated cluster” drills.
- **Risk: egress cost + recovery time explode during DR.**
  Mitigations: pre-position backups in-region, compress/deduplicate, measure restore throughput, and set tiered RPO/RTO that matches budget.

1.13.4.2 Security requirements you must implement (SOC2-aligned)

- **Least privilege:** workloads access storage via workload identity; no shared long-lived credentials.
- **Audit logging:** log all access to backups, snapshots, KMS key usage, and replication configuration changes; retain logs per SOC2 policy within residency boundary.
- **Rotation/expiry:** rotate KMS keys per policy; certificates/tokens for mTLS/replication must have expiry and automated rotation. Validate rotation doesn't break restore.

1.13.4.3 DR operational model (what tends to work)

- Define **tiered statefulness** and enforce it:
    - Tier 0/1: warm standby or managed cross-region replication (within the same residency boundary); documented RPO/RTO.
    - Tier 2+: backup/restore with scheduled snapshots.
- Use **GitOps** for cluster manifests *and* database infrastructure where feasible; treat manual console changes as incidents unless retroactively captured.
- Make replication lag and backup freshness **first-class SLO signals** (not just "CPU and memory").

1.13.5 Checklist

- [ ] You have an ADR per stateful system selecting **one** pattern (backup/restore, warm standby, active-active) with explicit RPO/RTO.
- [ ] `StorageClass` offerings are platform-owned, documented per cluster/region/provider, and enforced via policy.
- [ ] Backups, telemetry, artifacts, and audit logs are **residency-scoped** (EU/US) with separate keys and sinks.
- [ ] Encryption in transit and at rest is enabled; keys are rotated; restore procedures validate key access.
- [ ] DR runbooks include promotion/fencing steps to prevent split-brain; runbooks are tested on a schedule.
- [ ] You measure and alert on: backup age, snapshot success, restore success rate, replication lag, and DR execution time.
- [ ] Rollback strategy is defined for failed promotions (e.g., revert traffic, re-seed replica, invalidate divergent writers) and is practiced.

1.14 Observability: Metrics, Logs, Traces, and SLOs

1.14.1 Objectives

- Give you a **unified, multi-cloud observability stack** with consistent telemetry standards and metadata so you can operate across **cluster/region/provider** boundaries without bespoke dashboards per environment.
- Make operations **SLO-driven**: define SLIs/SLOs, manage **error budgets**, and gate risky changes with evidence.
- Meet **SOC2** requirements: least privilege access, audit logging, encryption everywhere, rotation/expiry, and evidence retention—while respecting **EU/US data residency**.

1.14.2 When to use/avoid

Use when

- You run Kubernetes clusters across multiple providers and need **consistent operations** (triage, paging, incident response) across regions.
- You must enforce **EU/US telemetry residency** (including “derived data” like logs, traces, and metrics).
- You target **99.9% availability** and need SLOs, burn-rate alerting, and change gating to keep within error budgets.

Avoid when

- You cannot staff or automate at least “follow-the-sun or 24/7 on-call” for the services you’re operating. Multi-cloud observability increases the number of failure modes.
- You need “portability” but have no concrete residency/reliability driver; you’ll pay complexity tax for little benefit.
- Your workloads are mostly stateful with provider-native semantics and you cannot define realistic RTO/RPO; the telemetry won’t save you from a DR plan gap.

1.14.3 Architecture notes

1.14.3.1 Assumptions

- You have at least one Kubernetes cluster per relevant **region/provider**, with **GitOps** managing baseline agents and policies (dependency: **s11**).
- You have a defined **tenant/team boundary model** (namespaces, separate clusters, or separate accounts/projects/subscriptions) and baseline IAM patterns (dependency: **s03**).
- You can provide private connectivity (VPN/interconnect) or mTLS over the internet with strict egress controls.

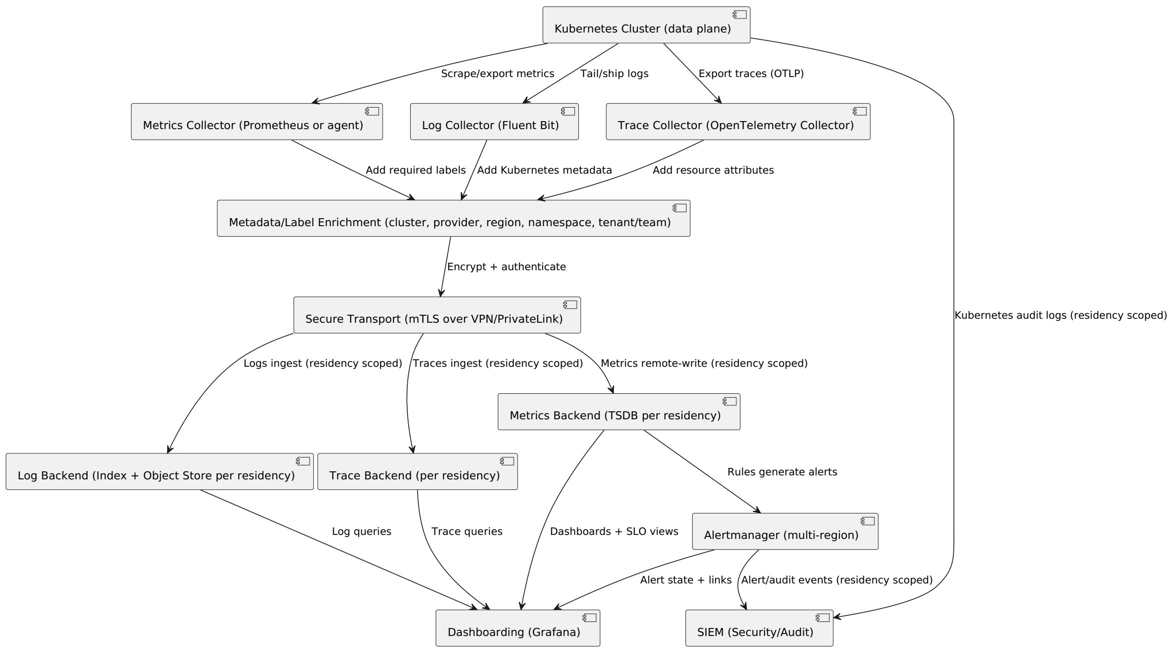
1.14.3.2 Non-goals

- Picking a single vendor backend; this section defines **interfaces and operational properties**, not a procurement decision.
- Achieving “single pane” by centralizing everything into one region. That violates residency for many teams and becomes a large blast radius.
- Solving application-level instrumentation for every language. You standardize **OpenTelemetry** and enforce minimum conventions.

1.14.3.3 Target reference architecture (multi-cluster)

- **Per-cluster collectors**:
    - Metrics: Prometheus (or equivalent scrape agent) plus optional remote-write.
    - Logs: Fluent Bit (or equivalent) for structured logs and Kubernetes metadata enrichment.
    - Traces: OpenTelemetry Collector (or language SDKs exporting to Collector).
- **Metadata/label enrichment** happens *before* leaving the cluster boundary to ensure consistent routing, storage, and retention:
    - Required attributes: `cluster`, `provider`, `region`, `namespace`, `workload`, `tenant/team`, `environment`.
- **Backends**:
    - Metrics TSDB (centralized per residency domain or federated query).
    - Log store (index + object storage) per residency domain.
    - Trace backend per residency domain.
- **Alerting**:
    - Alertmanager topology supports **multi-region** and minimizes shared fate.

    - Alert routing uses labels (`severity`, `service`, `tenant/team`, `region`) and integrates with paging and ticketing.
- **Security and compliance**:
    - mTLS for telemetry transport where possible; otherwise TLS with strong identity and network policy.
    - Least-privilege IAM for agents; no broad read of cluster secrets.
    - Audit logs forwarded to a **SIEM** while respecting EU/US boundaries.



Kubernetes Cluster (data plane)
Scrape/export metrics
Tail/ship logs
Export traces (OTLP)
Metrics Collector (Prometheus or agent)
Log Collector (Fluent Bit)
Trace Collector (OpenTelemetry Collector)
Add required labels
Add Kubernetes metadata
Add resource attributes
Metadata/Label Enrichment (cluster, provider, region, namespace, tenant/team)
Encrypt + authenticate
Secure Transport (mTLS over VPN/PrivateLink)
Kubernetes audit logs (residency scoped)
Logs ingest (residency scoped)
Traces ingest (residency scoped)
Metrics remote-write (residency scoped)
Metrics Backend (TSDB per residency)
Log Backend (Index + Object Store per residency)
Trace Backend (per residency)
Rules generate alerts
Log queries
Trace queries
Dashboards + SLO views
Alertmanager (multi-region)
Alert state + links
Alert/audit events (residency scoped)
Dashboarding (Grafana)
SIEM (Security/Audit)

Figure (diagram_id: observability-architecture): Multi-cluster observability pipeline using collectors and centralized backends for metrics, logs, and traces.

1.14.3.4 Centralized vs federated backends (decision matrix)

Choice	When it works	Tradeoffs	Failure modes to plan for
Centralized backend per residency domain (EU stack + US stack)	You can operate one backend per domain and route telemetry by region/ provider	Easier standardization; larger shared blast radius inside domain	Backend outage impacts all clusters in that domain; noisy neighbor risk
Federated storage (per-cluster or per-	You need strong blast-radius isolation or	Operational overhead; harder	Query layer becomes critical path;

Choice	When it works	Tradeoffs	Failure modes to plan for
region storage + global query)	regulatory constraints per country	global SLOs/ dashboards	inconsistent retention/ indexing
Hybrid (regional backends + domain-level “summary” rollups)	You need local depth but global ops reporting	Complexity in aggregation and data correctness	Rollup lag causes delayed alerting; double-counting in SLOs

Operations-focused opinion: **start with “centralized per residency domain”** unless you have a strong reason not to. Federated systems are easy to design and hard to operate reliably at 99.9% without a dedicated platform team.

1.14.4 Operational considerations

1.14.4.1 Telemetry standards and conventions (enforceable)

- **Metrics**: Prometheus exposition, stable naming, and required labels (`service`, `tenant/team`, `environment`, `cluster`, `region`, `provider`).
- **Logs**: structured JSON logs; include trace correlation fields (`trace_id`, `span_id`) where applicable; avoid PII by default (residency + SOC2).
- **Traces**: OpenTelemetry OTLP to per-cluster Collector; use consistent `service.name`, `deployment.environment`, `k8s.cluster.name`.
- **Correlation**: require propagation of W3C trace context; dashboards should pivot by `service` then jump to logs/traces by ID.

1.14.4.2 SLOs, error budgets, and release gating

- Define SLOs at the **service boundary** (user-visible), not component internals:
  - Availability SLO: 99.9% monthly for the service endpoint(s).
  - Latency SLO: p95/p99 per critical operation.
- Alert on **burn rate** (fast + slow) rather than raw thresholds.
- Gate deployments when:
  - Fast burn-rate alerts are firing, or
  - Error budget remaining is below your policy threshold (e.g., <20% remaining in window), or

    - Telemetry coverage drops below minimum (missing metrics/logs/traces is an operations incident).

Tradeoff: strict gating reduces change velocity; it also prevents “deploying into an outage,” which is a common multi-cloud failure amplifier.

1.14.4.3 Alerting topology and noise control

- Run Alertmanager in at least **two regions per residency domain**; avoid a single control-plane dependency for paging.
- Route alerts by:
    - `severity` (page vs ticket),
    - `tenant/team` (ownership),
    - `region/provider` (blast radius context).
- Noise reduction tactics you should standardize:
    - Grouping by `service` + `region`.
    - Deduplication keys that include `cluster` only when needed.
    - Explicit “dependency down” suppression to avoid alert storms (but document it—suppression hides reality).

1.14.4.4 Data residency and retention

- Maintain **EU-only** and **US-only** telemetry pipelines and storage by default.
- If you must support global views, use **aggregation that does not include customer/derived data** (e.g., SLO compliance percentages) and validate with compliance.
- Apply retention policies:
    - Logs: shortest by default (cost + sensitivity); archive with encryption and access controls.
    - Traces: sampled; keep long enough to support incident investigations and SOC2 evidence needs.
    - Metrics: downsampled long-term; raw short-term.

1.14.4.5 Security (SOC2-aligned)

- **Least privilege**
    - Collectors should only read what they need (Kubernetes API metadata, not Secrets).
    - Backend write identities should be scoped per cluster and per telemetry type.

- **Audit logging**
    - Audit access to telemetry backends and dashboards (who queried what).
    - Preserve GitOps change history for agents, rules, and dashboards.
- **Rotation/expiry**
    - Prefer workload identity over static keys.
    - If you must use client certs or tokens: short TTL, automated rotation, and alert on near-expiry.

1.14.4.6 Major risks and mitigations

Risk: Residency violation via “central observability.”
Mitigation: separate EU/US backends, enforce routing by `region/residency` labels, egress allowlists, policy checks in CI for telemetry configs, and periodic audits of storage locations and replication settings.

Risk: Observability control plane becomes a shared failure point.
Mitigation: multi-region Alertmanager, separate dashboards from paging, backpressure controls on collectors, and SLOs for the observability platform itself.

Risk: Signal loss during incidents (agents overloaded, backends throttling).
Mitigation: local buffering where feasible, resource requests/limits for agents, rate limits and sampling policies, and “graceful degradation” (drop debug logs before dropping error logs).

1.14.5 Checklist

- [ ] You have separate **EU** and **US** telemetry pipelines and storage, with documented boundaries and replication disabled across domains.
- [ ] Every metric/log/trace includes required metadata: `cluster`, `provider`, `region`, `namespace`, `tenant/team`, `environment`, `service`.
- [ ] Transport from cluster to backend is encrypted and authenticated (mTLS preferred); egress is allowlisted.
- [ ] Alerting is multi-region per residency domain; paging does not depend on a single region/provider.
- [ ] SLOs exist for each tier-1 service; burn-rate alerts are implemented; release gating policy is documented and enforced.
- [ ] Access to telemetry backends and dashboards is least-privilege and audited; credentials/certs rotate automatically with expiry alerts.

- [ ] Retention policies are defined per signal type and aligned to SOC2 evidence needs and cost constraints.
- [ ] You have run at least one game day validating: backend outage, network partition, agent overload, and “missing telemetry” detection.

1.15 Reliability Engineering: DR, Failover, and Chaos Testing (s15)

1.15.1 Objectives

- You implement **explicit DR tiers** (backup/restore, warm standby, active-active) with **tested RTO/RPO** per service and per data store.
- You standardize **failover orchestration** so traffic cutover is gated by **data readiness** and **smoke tests**, not operator optimism.
- You run **GameDays and chaos experiments** to validate failure modes (provider outage, region outage, dependency brownouts) and convert results into a **reliability backlog**.
- You maintain **99.9% service availability** at the service level using SLOs and error budgets, acknowledging DR mechanisms introduce new failure modes.

1.15.2 When to use/avoid

Use when

- You have **provider risk reduction** requirements (commercial/contractual) or must survive **provider or regional outages**.
- You must enforce **EU/US data residency boundaries** (including telemetry, backups, and artifacts) and need DR patterns that don’t violate residency.
- You operate services where **downtime cost** justifies added complexity and you can staff/automate on-call accordingly.

Avoid (or postpone) when

- Your system is tightly coupled to **provider-native state** (managed databases without cross-provider replication) and you cannot meet RTO/RPO without an app/data redesign.
- You cannot sustain the operational load (multi-cloud DR increases **blast radius** of configuration drift: DNS, IAM, networking, quotas, KMS).

- Your current maturity lacks: GitOps discipline, change management, observability SLOs, and tested restore procedures. Start **multi-region within one provider** first when possible.

1.15.3 Architecture notes (reference architecture)

1.15.3.1 Assumptions

- **SOC2**: least privilege IAM, audit logging, change management evidence, incident response procedures, and evidence retention.
- **Data residency**: hard EU/US boundaries for **application data plus derived data** (logs/metrics/traces), backups, and artifacts.
- **Encryption everywhere**: in transit (mTLS/TLS) and at rest (KMS-backed keys) with **rotation/expiry**.
- **Availability target**: 99.9% at the **service** boundary (not every component individually).

1.15.3.2 Non-goals

- Not an application refactor guide to make state “magically portable.”
- Not prescribing a single vendor stack.
- Not covering on-prem/hybrid in depth (different networking and lifecycle constraints).

1.15.3.3 DR tiers and what you’re really buying

DR tier	Typical RTO	Typical RPO	Cost/ complexity	What breaks first (common failure modes)	Best fit
Backup/ restore	Hours	Minutes–hours	Low–medium	Backups not restorable; restore too slow; missing secrets/	Internal tools, batch, non-critical services

DR tier	Typical RTO	Typical RPO	Cost/ complexity	What breaks first (common failure modes)	Best fit
				KMS access	
Warm standby	15–60 min	Seconds–minutes	Medium–high	Standby not capacity-ready; data lag; DNS/ traffic cutover errors	Customer-facing services with moderate tolerance
Active-active	Seconds–minutes	Near-zero (hard)	High	Consistency conflicts; split-brain; global traffic steering mistakes	Only when you can design for it (idempotency + conflict handling)

Opinionated guidance: Most teams should standardize on **warm standby** for critical stateless services and keep **backup/restore** for state until you can prove replication correctness under stress. Active-active is a product-level commitment, not an infra toggle.

1.15.3.4 Failover orchestration: traffic + data gating

- Treat **traffic management** (global DNS/Anycast/L7) as a *separate control plane* from **data readiness** (replication status, restore completion, schema compatibility).
- Failover requires pre-provisioned **quotas/capacity** in the target provider/ region (or automation that can prove it quickly).

- Keep provider-specific mechanics behind documented interfaces (ADRs), e.g.:
    - “Global Traffic Manager” abstraction (health checks, weighted routing, failover policy).
    - “Data Readiness” abstraction (replication lag, backup inventory, restore runbooks).
    - “Cluster readiness” abstraction (node pools, ingress, egress allowlists, policy parity).

1.15.3.5 Risks and mitigations (major designs)

- **Risk: false failover** (alert noise triggers cutover).
  Mitigation: multi-signal detection, explicit incident declaration, and *gated* traffic switch after smoke tests.
- **Risk: data inconsistency / lost writes** during failover.
  Mitigation: define RPO per datastore; enforce write-freeze or queueing; verify replica catch-up; document “non-failoverable” dependencies.
- **Risk: residency violations** via telemetry/backups during DR.
  Mitigation: region-scoped observability pipelines, residency-scoped KMS keys, allowlists, CI policy checks (OPA) blocking cross-boundary sinks.
- **Risk: IAM/KMS lockout** in the standby provider during an outage.
  Mitigation: pre-provision least-privilege roles, test workload identity/IRSA equivalents, and rotate credentials with auditable change control.
- **Risk: operational overload**.
  Mitigation: fewer clusters, automate runbooks, schedule GameDays, and push findings into a reliability backlog with owners/dates.

1.15.4 Operational considerations

1.15.4.1 What you must have before you claim “DR-ready”

- **RTO/RPO contracts** per service and per datastore (owned by the service team).
- A **dependency inventory** (ADR-backed) classifying each dependency as: failoverable, degraded-mode, or non-failoverable.
- **Pre-provisioned capacity** (or fast capacity proof) in the standby provider/ region:
    - cluster/node pool limits, IP space, egress NAT, ingress LBs, certificates.

- **Security readiness**:
    - least-privilege IAM mapped to workload identity; no long-lived credentials in-cluster.
    - audit logging enabled for control plane actions (GitOps, IAM, KMS, DNS, traffic manager).
    - key rotation/expiry policy tested in both primary and standby.
- **Change management** aligned with SOC2:
    - GitOps history for configuration, artifact promotion via immutable digests, and approvals for DR-affecting changes.

1.15.4.2 Capacity planning under failover

- Decide your **surge policy** explicitly:
    - **N+0 failover (no spare)**: cheaper, but RTO/Reliability suffer due to scaling and quota delays.
    - **N+1 partial spare**: typical for warm standby; reserve enough to serve critical paths.
    - **Full spare**: fastest failover; highest cost.
- Verify dependencies under failover load: egress limits, rate limits, DNS TTL behavior, and third-party APIs.

1.15.4.3 GameDays and chaos testing (what to test, not just how)

- Provider and region outage simulations (control plane unavailability, data plane packet loss, DNS resolution failures).
- Dependency fault injection: managed DB latency, KMS throttling, registry unavailability, identity provider brownouts.
- Validate you can still:
    - freeze deploys,
    - promote artifacts,
    - reconcile GitOps safely,
    - collect audit logs and evidence during incidents.

1.15.5 Provider outage failover runbook (traffic + data)

1.15.5.1 Objectives

- You execute a repeatable, auditable workflow to fail over from Provider A to Provider B with **data readiness gates** and **verification** steps.

1.15.5.2 When to use/avoid

- **Use** when Provider A has a confirmed outage impacting the service SLO and your tier's RTO/RPO require cross-provider recovery.
- **Avoid** when the service is in a state that makes failover unsafe (e.g., non-failoverable dependency is down, replica lag exceeds RPO, or you cannot enforce a write-freeze).

1.15.5.3 Architecture notes

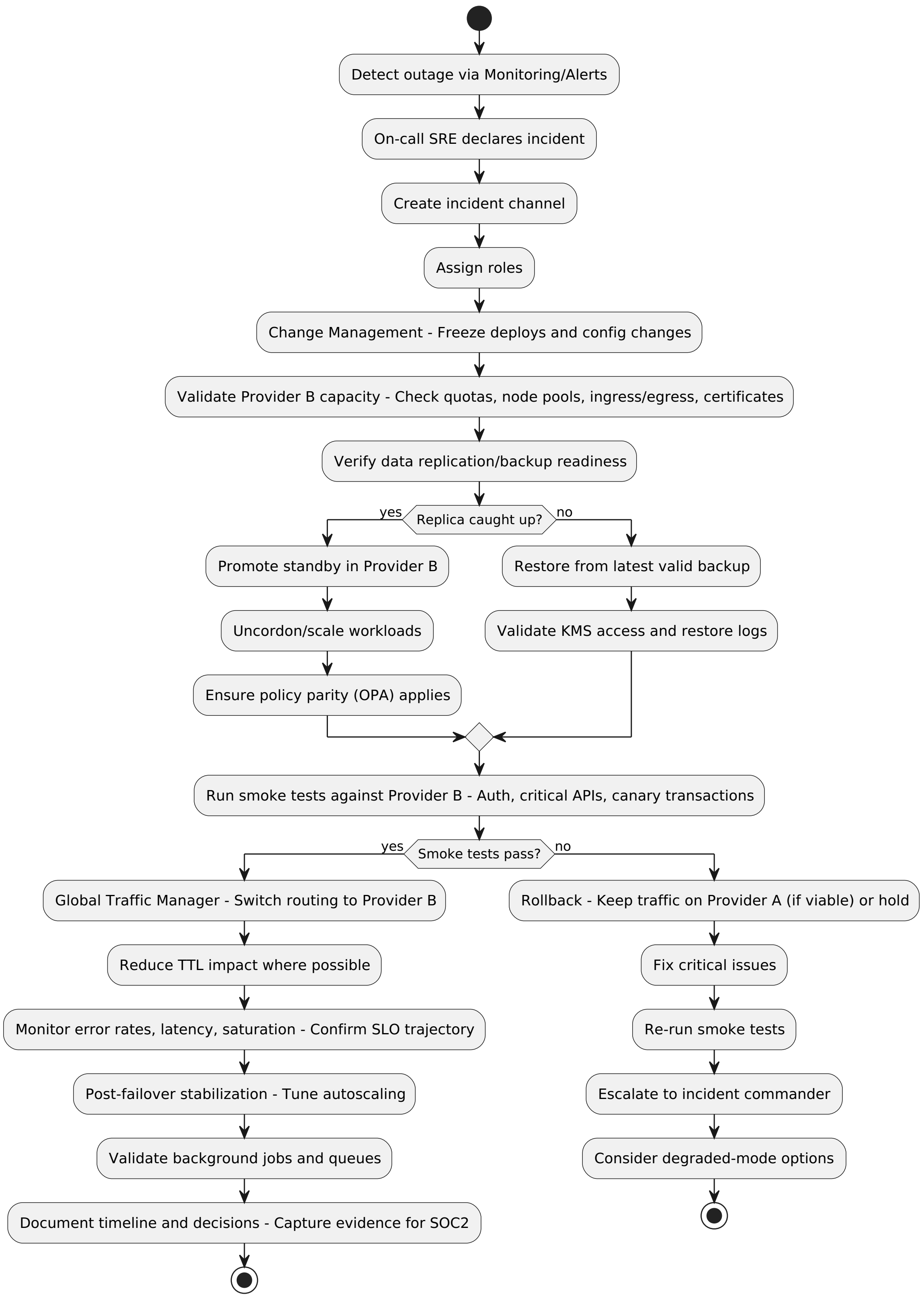
- The runbook assumes you have:
  - a **Global Traffic Manager** with health checks and weighted/failover routing,
  - a **standby** cluster in Provider B (warm standby) or a restore path (backup/restore),
  - **region-scoped** telemetry and backups respecting EU/US boundaries.

1.15.5.4 Operational considerations

- **Freeze deploys** early: the fastest way to extend an outage is deploying while you're blind.
- Prefer **automated checks** over human judgment: replica lag, cluster readiness, smoke test suite.
- Maintain a **rollback path**: the rollback is often "stay failed over" (if Provider A is unstable), so define rollback criteria carefully.

1.15.5.5 Diagram (diagram_id: dr-failover-runbook-flow)

Alt text: Failover runbook activity flow from detection through validation and traffic cutover.


Detect outage via Monitoring/Alerts
On-call SRE declares incident
Create incident channel
Assign roles
Change Management - Freeze deploys and config changes
Validate Provider B capacity - Check quotas, node pools, ingress/egress, certificates
Verify data replication/backup readiness
Replica caught up?
yes
no
Promote standby in Provider B
Uncordon/scale workloads
Ensure policy parity (OPA) applies
Restore from latest valid backup
Validate KMS access and restore logs
Run smoke tests against Provider B - Auth, critical APIs, canary transactions
Smoke tests pass?
yes
no
Global Traffic Manager - Switch routing to Provider B
Reduce TTL impact where possible
Monitor error rates, latency, saturation - Confirm SLO trajectory
Post-failover stabilization - Tune autoscaling
Validate background jobs and queues
Document timeline and decisions - Capture evidence for SOC2
Rollback - Keep traffic on Provider A (if viable) or hold
Fix critical issues
Re-run smoke tests
Escalate to incident commander
Consider degraded-mode options

1.15.6 Checklist

- [ ] RTO/RPO defined per service and datastore; tested quarterly (minimum).

- [ ] EU/US residency boundaries enforced for app data, telemetry, backups, and artifacts.
- [ ] Encryption in transit (TLS/mTLS) and at rest (KMS) with rotation/expiry tested in both providers.
- [ ] Least-privilege IAM + workload identity configured; audit logs enabled for IAM/KMS/DNS/GitOps.
- [ ] Standby capacity verified (quotas, IPAM, ingress/egress, certificates) and monitored.
- [ ] Failover runbook is automated where possible; includes prechecks, verification, and rollback criteria.
- [ ] GameDays scheduled; findings tracked in a reliability backlog with owners and dates.

1.16 Security Posture: Hardening, Threat Models, and Incident Response (s16)

1.16.1 Objectives

- You define a baseline security posture that is consistent across **providers/ regions/clusters** while keeping provider-specific controls at the edges.
- You separate **control plane** and **data plane** threats and controls, so incidents don’t cascade through shared tooling.
- You operate in a way that supports **SOC2 evidence**, **EU/US data residency**, **encryption everywhere**, and **99.9% availability** (including during containment actions).

1.16.2 When to use/avoid

When to use

- You run multi-cloud Kubernetes with tenant/team boundaries and need repeatable hardening, detection, and incident response (IR).
- You have strict **EU/US residency** requirements for **customer data and derived data** (logs/metrics/traces, backups, artifacts).
- You need auditable controls: least privilege, change control, access reviews, and provable incident handling.

When to avoid (or defer)

- You cannot staff 24/7 on-call or you lack disciplined change management; multi-cloud security increases operational load and failure modes.

- You rely heavily on stateful/provider-native services without a clear residency and IR story (containment and evidence handling gets messy fast).

1.16.3 Architecture notes

1.16.3.1 Assumptions

- You already have baseline multi-cloud foundations from **s08/s09/s14** (networking/identity/observability) and can enforce org-level guardrails.
- You maintain **separate EU and US management planes for derived data** (at minimum separate telemetry sinks, artifact registries, and backup targets), or you can prove derived-data residency controls end-to-end.
- All clusters use GitOps and policy-as-code; “kubectl from a laptop” is **break-glass only** with time bounds and audit trails.

1.16.3.2 Non-goals

- Designing a bespoke cryptosystem (you use provider KMS/HSM capabilities and standard TLS/mTLS).
- Promising “portable security” across providers without acknowledging IAM/KMS/logging differences; you standardize interfaces and document deltas via ADRs.
- Full forensic playbooks for every platform; this section defines baseline patterns and minimum operational behaviors.

1.16.3.3 Threat model: control plane vs data plane

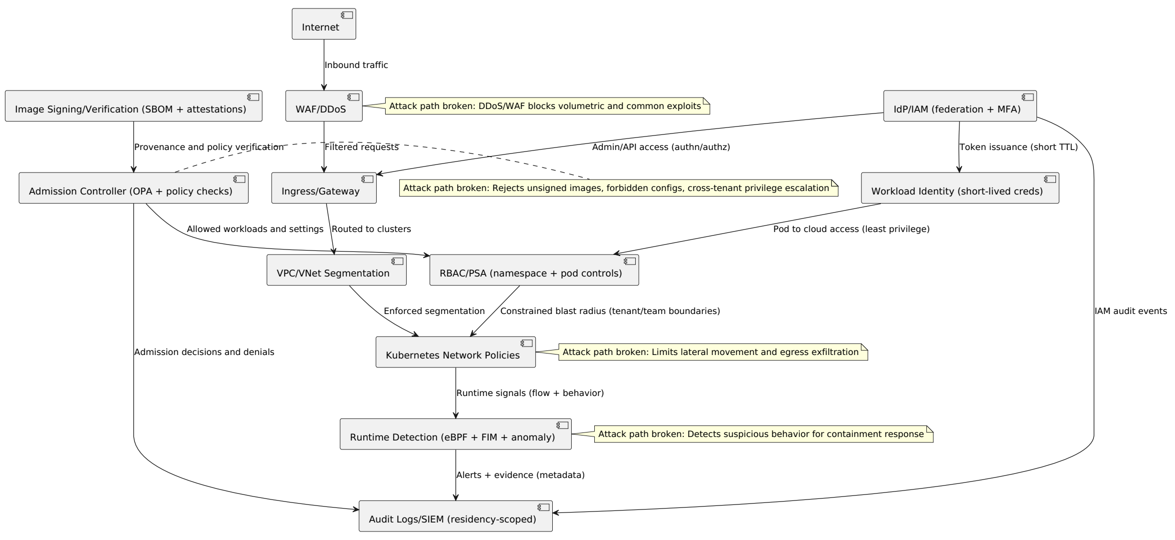
- **Control plane threats (high blast radius):**
  - Compromised CI/GitOps controller, registry, signing keys, or policy controller (OPA) can push malicious configuration across clusters/providers.
  - IAM misconfiguration (over-broad roles, stale credentials) enables cross-tenant or cross-region access.
  - Central observability/alerting compromise can blind detection across environments.
- **Data plane threats (frequent, local impact if isolated well):**
  - Container escape, node compromise, malicious sidecars, credential theft from pods, lateral movement via permissive network policies.
  - Unsafe egress enabling data exfiltration or command-and-control.
  - Vulnerable images or runtime drift due to unpatched nodes.

1.16.3.4 Risks and mitigations (major design callouts)

- **Risk: residency violations via derived data replication** (telemetry, backups, artifacts).
  Mitigations: residency-scoped sinks and keys, egress allowlists, CI policy checks that block cross-boundary destinations, and explicit “EU-only/US-only” routing for logs/metrics/traces.
- **Risk: shared control tooling becomes a common point of failure** (and can break 99.9% by causing widespread outages).
  Mitigations: isolate control-plane components by residency boundary; prefer “fail-open vs fail-closed” explicitly per control (see tradeoffs below), and run game days.
- **Risk: policy drift across clusters/providers** leading to inconsistent posture.
  Mitigations: GitOps reconciliation, conformance tests, continuous compliance scans, and periodic access reviews with evidence retention.

1.16.3.5 Defense-in-depth reference architecture

Figure: defense-in-depth layers for securing Kubernetes workloads across multiple clouds (diagram_id: security-defense-in-depth).


Internet
Inbound traffic
Image Signing/Verification (SBOM + attestations)
WAF/DDoS
Attack path broken: DDoS/WAF blocks volumetric and common exploits
IdP/IAM (federation + MFA)
Provenance and policy verification
Filtered requests
Admin/API access (authn/authz)
Token issuance (short TTL)
Admission Controller (OPA + policy checks)
Ingress/Gateway
Attack path broken: Rejects unsigned images, forbidden configs, cross-tenant privilege escalation
Workload Identity (short-lived creds)
Allowed workloads and settings
Routed to clusters
Pod to cloud access (least privilege)
VPC/VNet Segmentation
RBAC/PSA (namespace + pod controls)
Enforced segmentation
Constrained blast radius (tenant/team boundaries)
IAM audit events
Admission decisions and denials
Kubernetes Network Policies
Attack path broken: Limits lateral movement and egress exfiltration
Runtime signals (flow + behavior)
Runtime Detection (eBPF + FIM + anomaly)
Attack path broken: Detects suspicious behavior for containment response
Alerts + evidence (metadata)
Audit Logs/SIEM (residency-scoped)

1.16.3.6 Tradeoffs you must decide explicitly

Control	Fail-closed benefit	Fail-closed failure mode	When you choose fail-open
Admission (policy/	Blocks malicious or	Can block all deploys during	During availability-critical incident

Control	Fail-closed benefit	Fail-closed failure mode	When you choose fail-open
signature verification)	non-compliant deploys	controller outage; may extend incidents	response; only with compensating controls and time-boxed exception
Egress restrictions	Reduces exfiltration and malware C2	Breaks dependencies if allowlists incomplete	Early migration phases with heavy discovery; you still log and progressively tighten
Runtime enforcement (kill/ quarantine)	Fast containment	False positives can cause outages and breach error budgets	Start detect-only, graduate to enforce for high-confidence policies

1.16.4 Operational considerations

1.16.4.1 Baseline hardening controls (minimum)

- **Identity and access (SOC2-aligned)**
  - You enforce MFA at the IdP, conditional access, and device posture for human access.
  - You use **least privilege**: separate roles for break-glass, CI/CD, and platform operators; no shared accounts.
  - You use **workload identity** for pod-to-cloud access with **short-lived credentials**; avoid long-lived secrets in clusters.
  - You implement rotation/expiry for: signing keys, KMS key policies, OIDC trust configs, and emergency credentials; document rotation intervals and owners.
- **Cluster hardening**
  - You enforce Kubernetes **RBAC** with namespace-level tenant/team boundaries; minimize cluster-admin bindings.
  - You enforce **Pod Security Admission (PSA)** (or equivalent) with baseline/restricted profiles; exceptions via reviewed policy.

    - You restrict node access (SSH disabled where possible; audited session manager otherwise) and harden node images.

- **Network segmentation and zero trust**
    - You segment at **VPC/VNet** and at Kubernetes via **NetworkPolicies**; default-deny for inter-namespace traffic.
    - You control **egress** with allowlists and/or egress gateways; log denied egress attempts as security signals.
    - You enforce **encryption in transit** (TLS/mTLS where appropriate) and **encryption at rest** with KMS-backed keys scoped per residency boundary.

- **Supply chain and artifact promotion**
    - You sign images and verify signatures at admission; you store SBOMs and build attestations with immutable references (digests).
    - You enforce promotion rules: dev → staging → prod via approvals and policy checks; no “latest” tags in production.
    - You keep registries **residency-scoped** (EU registry for EU workloads/artifacts; same for US) to avoid derived-data boundary violations.

- **Logging and auditability**
    - You collect: Kubernetes audit logs, admission decisions, IAM audit logs, and runtime detection alerts.
    - You ship logs to **residency-scoped SIEM/sinks**; you do not centralize EU and US logs into a single cross-boundary store.
    - You define retention, access controls, and integrity protections (WORM/immutability where available) suitable for SOC2 evidence.

1.16.4.2 Runtime security: detection-first, then enforcement

- Start with **detect-only** for eBPF sensors/FIM/anomaly detection to avoid false-positive outages.
- Promote to enforce for a small set of high-confidence rules (e.g., unexpected privilege escalation, known crypto-miner patterns, outbound to forbidden destinations).
- Treat runtime controls as part of your SLO strategy: measure alert quality and mean time to acknowledge/contain; avoid “alert floods” that cause missed incidents.

1.16.4.3 Incident response (IR) patterns for multi-cloud

- **Control-plane compromise containment priority:** if CI/GitOps/signing keys are suspected, you assume blast radius is multi-cluster and act accordingly (freeze promotions, revoke credentials, rotate keys).
- **Residency-safe evidence handling:** evidence from EU clusters stays in EU; US stays in US. If you need cross-boundary coordination, share **metadata-only** incident summaries, not raw logs/traces containing customer/derived data.
- **Break-glass access:** time-bound, approved, audited; revoke after use; ensure actions are logged in both IAM and Kubernetes audit logs.

1.16.5 Checklist

1.16.5.1 Baseline readiness (SOC2 + residency + encryption)

- [ ] EU and US have separate telemetry sinks and retention policies; no cross-boundary log shipping.
- [ ] Workload identity is enabled; no long-lived cloud keys in cluster secrets.
- [ ] Kubernetes audit logging is enabled and forwarded to residency-scoped SIEM with access controls.
- [ ] Admission enforces: signature verification, PSA baseline/restricted, and least-privilege RBAC patterns.
- [ ] Default-deny NetworkPolicies and controlled egress are in place for production namespaces.
- [ ] Artifact promotion is digest-based with approvals and policy checks; registries are residency-scoped.
- [ ] Key rotation/expiry is defined for KMS keys, signing keys, and OIDC trust relationships.

1.16.5.2 IR operational readiness

- [ ] You have an incident severity model that includes “residency breach” as a top-tier event.
- [ ] You can revoke/rotate: workload identity trust, CI credentials, signing keys, and cluster-admin access within your RTO expectations.
- [ ] You run game days that include: control-plane tooling outage, policy controller outage, and false-positive runtime detection scenarios.

1.17 FinOps and Cost Engineering for Multi-Cloud (s17)

1.17.1 Objectives

- Identify the major cost drivers and “hidden multipliers” across providers (especially data movement).
- Define practical showback/chargeback mechanics that work with Kubernetes tenant/team boundaries.
- Implement cost guardrails as policy (not tribal knowledge) while preserving 99.9% SLO delivery.

1.17.2 When to use/avoid

Use when

- You run Kubernetes clusters across multiple providers/regions and need cost predictability without breaking SOC2 controls.
- You have hard EU/US data residency boundaries and must prevent “accidental cross-residency spend” (telemetry, backups, artifacts).
- You’re mature enough to treat cost as an operational SLO with alerts, ownership, and rollback.

Avoid when

- You can’t staff the operational overhead (cost optimization is continuous, not a one-off).
- You mainly want “portability.” Multi-cloud often increases cost through duplication and egress.
- Your workloads are heavily stateful and tightly coupled to provider-native data services; cost levers get constrained and DR becomes expensive.

1.17.3 Architecture notes

1.17.3.1 Assumptions

- SOC2-aligned controls: least privilege IAM, audit logging, PR-based change management, evidence retention.
- EU/US residency separation applies to **telemetry, backups, and artifacts** (not just primary data).
- Encryption in transit and at rest with rotation/expiry is enforced everywhere.

- Target availability is 99.9%, managed via SLOs/error budgets (you don’t “optimize” by silently degrading reliability).

1.17.3.2 Non-goals

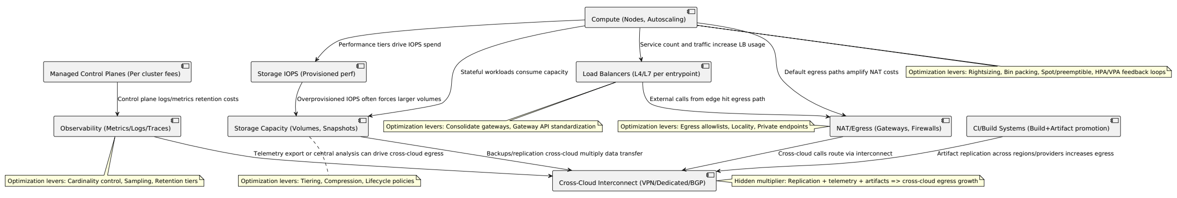
- Selecting a single vendor tool for cost management.
- “Perfect” per-request cost accounting (usually not worth the operational complexity).
- Eliminating all provider variance; you’re aiming for consistent interfaces and comparable unit economics.

1.17.3.3 Cost drivers you should model explicitly

1. **Data movement**
   - Cross-zone/region charges inside a provider.
   - Cross-cloud egress (often the largest hidden bill driver).
   - NAT/egress gateways as “tax collectors” for poorly planned egress.
2. **Traffic entry/exit**
   - Load balancers per Service/Ingress/Gateway.
   - Global traffic management and health checks.
3. **Compute**
   - Node instance choice, overprovisioning, bin packing, and headroom for failover.
   - Managed control plane fees (per cluster) pushing you toward fewer, larger clusters—at the expense of blast radius.
4. **Storage**
   - Capacity plus **IOPS/throughput** (and the trap: paying for provisioned performance you don’t use).
   - Snapshots, backups, replication (especially cross-region/cross-cloud).
5. **Observability**
   - Cardinality explosion (labels, high-churn dimensions).
   - Retention and rehydration costs.
6. **CI/build and artifacts**
   - Build concurrency, cache misses, artifact replication across residency boundaries.

1.17.3.4 Diagram: Multi-Cloud Cost Drivers Map (diagram_id: cost-drivers-map)

Alt text: Key multi-cloud Kubernetes cost drivers and how data movement amplifies spend.


Compute (Nodes, Autoscaling)
Performance tiers drive IOPS spend
Service count and traffic increase LB usage
Managed Control Planes (Per cluster fees)
Storage IOPS (Provisioned perf)
Stateful workloads consume capacity
Load Balancers (L4/L7 per entrypoint)
Default egress paths amplify NAT costs
Optimization levers: Rightsizing, Bin packing, Spot/preemptible, HPA/VPA feedback loops
Control plane logs/metrics retention costs
Overprovisioned IOPS often forces larger volumes
External calls from edge hit egress path
Observability (Metrics/Logs/Traces)
Storage Capacity (Volumes, Snapshots)
Optimization levers: Consolidate gateways, Gateway API standardization
Optimization levers: Egress allowlists, Locality, Private endpoints
NAT/Egress (Gateways, Firewalls)
CI/Build Systems (Build+Artifact promotion)
Telemetry export or central analysis can drive cross-cloud egress
Backups/replication cross-cloud multiply data transfer
Cross-cloud calls route via interconnect
Artifact replication across regions/providers increases egress
Optimization levers: Cardinality control, Sampling, Retention tiers
Optimization levers: Tiering, Compression, Lifecycle policies
Cross-Cloud Interconnect (VPN/Dedicated/BGP)
Hidden multiplier: Replication + telemetry + artifacts => cross-cloud egress growth

1.17.3.5 Risks and mitigations (major failure modes)

- **Residency leakage via telemetry/artifacts/backups**
  - *Failure mode:* EU workloads export traces to US; US build pipeline pushes artifacts into EU registry (or vice versa).
  - *Mitigations:* residency-scoped sinks and registries, egress allowlists, policy-as-code checks in CI, residency-scoped KMS keys, and audit logs per boundary.
- **Cost optimization breaking availability**
  - *Failure mode:* aggressive rightsizing removes failover headroom; spot usage spikes evictions; reducing observability retention blinds incident response.
  - *Mitigations:* SLO-based guardrails, “do-not-evict” for tier-0, minimum replica and PDB standards, and explicit retention tiers with incident requirements.
- **Shared service mis-attribution**
  - *Failure mode:* platform team absorbs all LB/NAT/observability spend, encouraging waste by tenants.
  - *Mitigations:* showback models for shared services, chargeback for controllable usage, and “cost-to-serve” reporting per tenant/team.

1.17.4 Operational considerations

1.17.4.1 Cost allocation model (showback/chargeback)

You need allocation that matches your isolation boundaries: **provider/account → region → cluster → namespace (tenant/team)**.

Required signals

- **Namespace ownership**: `tenant`, `team`, `env`, `residency` labels (enforced by policy).
- **Workload identity** attribution for cloud resources created by controllers (e.g., ingress controllers, CSI drivers) so you can map spend to tenant/team.
- **Cluster inventory**: cluster name, provider, region, residency boundary, and environment.

Tradeoff: fine-grained accuracy vs operational overhead

- Namespace-based allocation is usually “good enough” and auditable. Per-pod network egress attribution is more accurate but often not worth the complexity unless egress dominates.

1.17.4.2 Cost guardrails as policy (what you enforce)

- **LB/NLB creation controls**
    - Limit per-namespace gateway/ingress objects; require shared Gateway instances where appropriate.
    - Enforce approved annotations/classes to avoid accidental premium SKUs.
- **Egress controls**
    - Default-deny egress + explicit allowlists for destinations; route via centralized NAT/egress points for auditability.
    - Require locality: prefer in-region dependencies; cross-cloud calls require an ADR and budget owner.
- **Storage controls**
    - Allowed storage classes per tier; prohibit high-IOPS defaults in dev/test.
    - Snapshot/backup lifecycle policy required for any PVC over threshold size.
- **Observability controls**
    - Cardinality budgets (drop/deny high-risk labels); sampling requirements for traces; retention tiers aligned to SOC2 evidence needs.
- **CI/build controls**
    - Artifact promotion must be immutable (digest-pinned) and residency-scoped; replication is explicit and logged.

Security requirements to keep intact while optimizing:

- **Least privilege** for cost APIs and billing exports (read-only for most roles; write limited to automation with approvals).
- **Audit logging** for budget policy changes and egress policy changes.
- **Rotation/expiry** for any integration credentials (prefer workload identity; avoid long-lived keys).

1.17.4.3 Commitment models vs portability (decision matrix)

Lever	When it helps	When it hurts	Operational cautions
Provider commitments (reserved/committed use)	Steady baseline compute/storage in a residency boundary	Reduces provider flexibility; can lock in poor architecture choices	Treat as capacity planning input; revisit quarterly with SLO/error budget context
Spot/preemptible	Stateless, horizontally scalable, batch, CI runners	Stateful, latency-sensitive, tier-0	Require eviction-aware design, PDBs, and surge capacity for failover
Multi-region within one provider (vs multi-cloud)	Improves availability with lower variance	Doesn’t mitigate provider-wide risk	Often the right first step before multi-cloud

1.17.4.4 Rightsizing + autoscaling feedback loop (what you operationalize)

- Use HPA/VPA (or equivalents) with **SLO gates**:
    - Don’t reduce requests/limits below what preserves p95 latency and error rate.
- Enforce “headroom budgets”:
    - Keep capacity for failover consistent with RTO/RPO tiering; otherwise you’ll fail over into an outage.

1.17.4.5 Budget alerts and response

Budget alerts are only useful if they trigger action.

- Alert on **rate of spend change** (week-over-week) per residency boundary and tenant/team.
- Tie alerts to a runbook: investigate top deltas (egress, LB count, log volume), then apply safe levers (sampling, retention reduction in non-prod, scaling caps).

1.17.5 Checklist

- [ ] You enforce required namespace labels: `tenant`, `team`, `env`, `residency` (policy-as-code, audited).
- [ ] You can attribute shared services (LB, NAT, observability) to tenant/team with a documented methodology.
- [ ] You have residency-scoped telemetry sinks, artifact registries, and backup targets; cross-boundary flows require an ADR.
- [ ] You have cost guardrails for LB creation, egress destinations, storage classes, and observability cardinality/retention.

- [ ] You run monthly rightsizing reviews with SLO impact checks and rollback plans.
- [ ] Budget alerts exist per provider/region/cluster/tenant and have an on-call action path.
- [ ] Commitment and spot/preemptible usage are documented with failure modes and tested during game days.

1.18 Operational Runbooks and Day-2 Procedures (s18)

1.18.1 Objectives

- Give you repeatable, auditable procedures for common day-2 events across **provider/region** boundaries.
- Standardize execution so on-call outcomes depend less on individual expertise and more on **checked, versioned runbooks**.
- Preserve **SOC2** evidence trails (who/what/when/why), enforce **EU/US data residency**, and maintain **encryption everywhere** while meeting **99.9% service availability** targets.

1.18.2 When to use/avoid

Use when

- You operate multiple Kubernetes clusters across **providers** and/or **regions** with shared control plane tooling (GitOps, policy, CI, observability).
- You must prove change control, access control, and DR testing (SOC2) without relying on tribal knowledge.

Avoid (or narrow scope) when

- You cannot staff incident response and change execution reliably (multi-cloud increases failure modes and on-call cognitive load).
- Your workloads are heavily stateful and have not defined/tested **RTO/RPO**; these runbooks won’t save you from unclear data semantics.

1.18.3 Architecture notes

- Runbooks assume a **control plane/data plane separation**: your GitOps/policy/identity foundations must not depend on any single cluster’s data plane being healthy.

- Every runbook has explicit prerequisites (observability, access, backups, etc.) to reduce “blind changes” that cause extended outages.
- Residency is treated as a hard boundary: every operational step must be doable *within* EU and within US without cross-shipping **derived data** (logs/metrics/traces), backups, or artifacts.

1.18.4 Assumptions

- GitOps is the default change mechanism; emergency changes are the exception and require follow-up reconciliation.
- Audit logging exists for: cloud IAM, Kubernetes API, Git, CI/CD, registry, and secrets/KMS access.
- Encryption in transit (TLS/mTLS where justified) and at rest (KMS-backed) is already enabled; rotation/expiry processes exist.
- Dependencies from **s14/s15/s16** are in place: observability/SLOs, DR strategy, and incident-response/security posture.

1.18.5 Non-goals

- This section does not define your entire incident response program; it provides the operational “how” for common events.
- This section does not attempt to make workloads “portable” across providers beyond defined interfaces; provider-specific edges still exist (IAM/LB/DNS/KMS/telemetry).

1.18.6 Risks and mitigations (major designs)

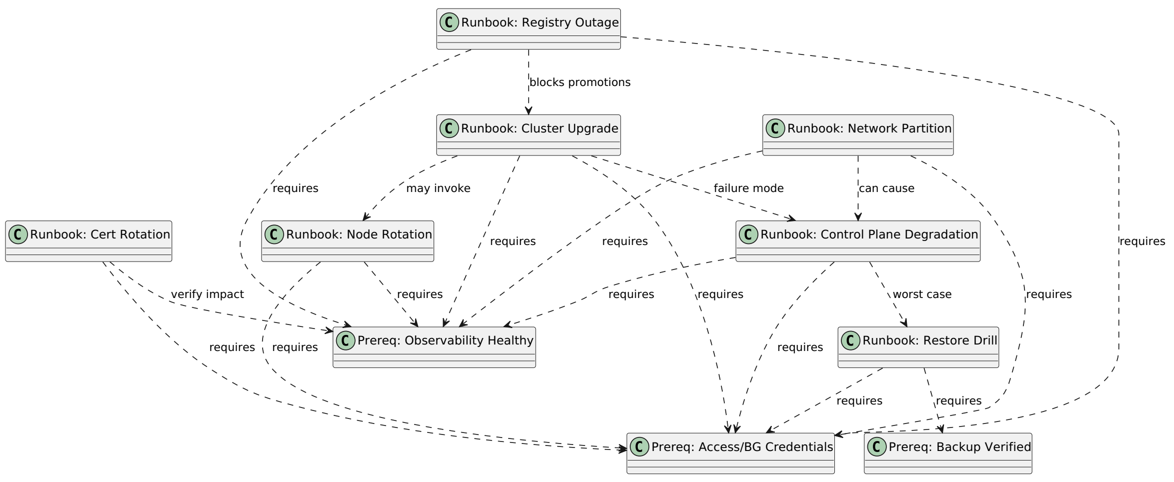
- **Risk: Common-tooling blast radius (control plane tooling outage impacts all clusters).**
  Mitigation: isolate tooling by residency boundary (EU vs US), use staged rollouts, and require local “break-glass” paths per provider/region.
- **Risk: Residency violations via telemetry, backups, or artifact replication during incidents.**
  Mitigation: region-scoped pipelines, allowlists, residency-scoped keys, and CI/policy checks that fail closed.
- **Risk: Change-induced multi-cluster outage (same mistake everywhere).**
  Mitigation: progressive rollout, canaries, freeze windows, and explicit rollback steps with verification gates.

1.18.7 Operational considerations

- Treat runbooks as code: versioned, reviewed, and tested (game days/ restore drills).
- Prefer small blast radius: per-tenant/team boundaries, per-cluster staged execution, and explicit “stop conditions”.
- Evidence capture is part of “done”: ticket/ADR links, approvals, logs, timestamps, and verification artifacts.

1.18.8 Diagram: Runbook dependency index (diagram_id: day2-ops-runbook-index)

Alt text: Runbook dependency map for common day-2 operations in multi-cloud Kubernetes.



1.19 Runbook: Access workflows (Just-in-Time and Break-glass)

1.19.1 Objectives

- Ensure you can regain access during incidents without permanently expanding IAM privileges.
- Produce SOC2-ready evidence: approvals, time bounds, and audit logs for access usage.

1.19.2 When to use/avoid

Use when

- You need elevated privileges for a bounded operational task (upgrade, rotation, restore).
- You are operating across multiple providers/regions and need consistent access semantics.

Avoid when

- You’re using break-glass as the default path (that’s a control failure and increases blast radius).

1.19.3 Architecture notes

- Prefer **workload identity** (no long-lived secrets) for services; prefer short-lived human access for operators.
- Separate access paths by **residency boundary** (EU vs US) so operators and audit logs don’t force cross-boundary dependency.
- All elevated access must be traceable to a ticket/incident and expire automatically.

1.19.4 Operational considerations

- **Least privilege:** roles scoped to specific cluster/region and bounded actions (e.g., rotate certs, cordon nodes).
- **Audit logging:** centralize within residency boundary; retain per SOC2 policy.
- **Rotation/expiry:** break-glass creds must have strict expiry and a tested rotation process; assume compromise eventually.

1.19.5 Checklist

- [ ] JIT elevation requires ticket/incident ID, approver, expiry, and scope (provider/region/cluster).
- [ ] Break-glass is sealed (hardware token/vault), time-bound, and produces immutable audit events.
- [ ] Post-incident: reconcile any manual changes back into GitOps; revoke elevated access; attach audit evidence.

1.20 Runbook: Cluster upgrade

1.20.1 Objectives

- Upgrade Kubernetes control plane and node pools safely with minimal user impact and clear rollback.
- Maintain policy and audit integrity during the change.

1.20.2 When to use/avoid

Use when

- A supported upgrade path exists and you can validate in lower environments first.

Avoid when

- Observability is degraded or backups are unverified (you’ll be flying blind).
- You cannot enforce a freeze on unrelated changes (you need a clean signal).

1.20.3 Architecture notes

- Upgrades are a **control plane + data plane** coordinated change; treat add-ons (CNI, CSI, ingress, service mesh) as first-class dependencies.
- Prefer immutable node replacement (see Node Rotation) over in-place mutation.

1.20.4 Operational considerations

- Schedule upgrades during defined windows; use risk scoring and progressive rollout.
- Ensure EU/US telemetry and artifacts remain boundary-scoped during upgrades (no “temporary” cross-region tooling).

1.20.5 Procedure (step-by-step)

1.20.5.1 Prechecks

1. Confirm approvals and change record exist (ticket linked to Git commit/PR).
2. Verify **Observability Healthy**: golden signals, SLO burn rate, and alerting functional in the target residency boundary.

3. Verify **Backup Verified** for any stateful components that affect service availability (etcd snapshots if self-managed; app data backups as applicable).
4. Validate compatibility matrix for Kubernetes version vs CNI/CSI/ingress/ service mesh.
5. Confirm you have **JIT access** and a tested break-glass path.

1.20.5.2 Execution

1. Upgrade a non-prod cluster in the same provider/region profile first (if not already done).
2. Upgrade control plane (managed control plane version bump or equivalent), one cluster at a time.
3. Perform **Node Rotation** to move workloads onto nodes with the new version.
4. Validate critical add-ons and policies (OPA/admission, CNI connectivity, DNS, ingress).
5. Progressively upgrade additional clusters (canary cluster → remainder), respecting tenant/team blast radius boundaries.

1.20.5.3 Verification

1. Confirm SLOs stable (no sustained error budget burn) and key service KPIs within expected range.
2. Confirm policy enforcement and audit logging are functioning (Kubernetes audit events present; IAM logs present).
3. Confirm artifact promotion and deployments still function end-to-end (a small canary deploy).

1.20.5.4 Rollback

1. If control plane rollback is not supported (common in managed offerings), treat rollback as “stop + mitigate”: halt rollout, scale capacity, and revert workloads to known-good node pools if possible.
2. Roll back add-on versions (CNI/ingress/service mesh) to the last known-good set via GitOps if they caused the regression.
3. If cluster is unrecoverable within RTO, follow **Control Plane Degradation** then **Restore Drill** procedures (restore to replacement cluster).

1.20.6 Checklist

- [ ] Observability healthy and SLO burn monitored throughout.
- [ ] Upgrade executed progressively (one cluster/tenant boundary at a time).
- [ ] Verification artifacts attached to the change record (dashboards, logs, timestamps).
- [ ] Rollout halted on first unresolved regression.

1.21 Runbook: Node rotation (replacement)

1.21.1 Objectives

- Replace nodes to apply OS/kernel/runtime patches, Kubernetes minor bumps, or configuration changes with controlled disruption.

1.21.2 When to use/avoid

Use when

- You need repeatable, low-risk mutation of the data plane.

Avoid when

- You cannot drain safely due to missing PDBs, non-idempotent workloads, or insufficient capacity headroom.

1.21.3 Architecture notes

- Immutable replacement reduces configuration drift and simplifies auditability.
- Requires workload readiness: PodDisruptionBudgets, health probes, and capacity planning.

1.21.4 Operational considerations

- Watch for hidden state (local disks, hostPath, node-local caches) that breaks during rotation.
- Ensure least-privilege: the actor rotating nodes should not also have broad secret/KMS privileges unless required.

1.21.5 Procedure (step-by-step)

1.21.5.1 Prechecks

1. Confirm cluster health and **Observability Healthy**.
2. Confirm workloads have PDBs and tolerate node drains; validate autoscaling headroom.
3. Confirm change record/approval and JIT access.

1.21.5.2 Execution

1. Add new node pool (or scale up replacement set) with desired version/config.
2. Cordon and drain nodes in small batches (respect PDBs and critical namespaces).
3. Monitor error rates, saturation, and queue/backlog during the drain.
4. Remove old nodes/node pool after workloads stabilize.

1.21.5.3 Verification

1. Confirm steady-state capacity, latency, and error rates.
2. Confirm no pods stuck on old nodes; confirm DaemonSets healthy.
3. Confirm audit logs show the operator actions and GitOps reconciliation is clean.

1.21.5.4 Rollback

1. Stop drains immediately if SLO burn accelerates.
2. Scale back up old node pool (if still present) and reschedule workloads.
3. Revert node pool configuration via GitOps (or IaC) to last known-good.

1.21.6 Checklist

- [ ] Batch size tuned to PDBs and error budget.
- [ ] Old nodes removed only after stability window passes.
- [ ] Evidence captured (node list before/after, drain events, SLO graphs).

1.22 Runbook: Certificate rotation

1.22.1 Objectives

- Rotate expiring certificates (ingress, mTLS, internal PKI) without breaking service-to-service and client connections.
- Maintain least-privilege and audit logging during sensitive secret operations.

1.22.2 When to use/avoid

Use when

- Certs are approaching expiry, CA rotation is required, or compromise is suspected.

Avoid when

- You can't verify trust chain propagation across clusters/regions (rotation without verification is a self-inflicted outage).

1.22.3 Architecture notes

- Separate cert domains by residency boundary if certs embed identifiers that could be considered derived data.
- Prefer automated issuance/renewal; manual rotation is break-glass only.

1.22.4 Operational considerations

- **Audit logging:** log access to certificate material and KMS decrypt events.
- **Rotation/expiry:** short-lived certs reduce compromise window but increase operational dependency on issuer availability.

1.22.5 Procedure (step-by-step)

1.22.5.1 Prechecks

1. Inventory certs: subject, SANs, issuer, expiry, and where deployed (control plane vs data plane).
2. Confirm issuer availability in-region (EU and US independently).
3. Confirm JIT access; restrict who can read secret material.

1.22.5.2 Execution

1. Issue new certs (or rotate CA) with overlap period (old and new valid concurrently).
2. Deploy to a canary ingress/gateway first; validate mTLS handshakes and client compatibility.
3. Roll out progressively across clusters and regions.
4. Revoke old certs only after confirming traffic has drained and caches updated.

1.22.5.3 Verification

1. Confirm handshake success rates and absence of trust errors in logs/ metrics.
2. Confirm no clients pinned to old chains.
3. Confirm audit events for secret access and changes are present.

1.22.5.4 Rollback

1. Re-deploy last known-good cert bundle (kept encrypted and access-controlled).
2. If CA rotation caused failure, reintroduce old CA into trust store temporarily while you correct distribution gaps.

1.22.6 Checklist

- [ ] Overlap window planned and executed.
- [ ] Least-privilege enforced for cert issuance and secret read paths.
- [ ] Rotation evidence captured (expiry before/after, rollout timeline, audit logs).

1.23 Runbook: Registry outage (image pull / promotion failure)

1.23.1 Objectives

- Keep deployments safe and predictable when your registry (or cross-region replication) is degraded.
- Avoid “panic pushes” that destroy artifact promotion traceability.

1.23.2 When to use/avoid

Use when

- Image pulls fail, promotions stall, or signatures/metadata verification breaks.

Avoid when

- Your only plan is to bypass artifact promotion controls; that’s an audit and security failure.

1.23.3 Architecture notes

- Registry availability is a supply-chain dependency; isolate by residency boundary (EU registry for EU workloads, US registry for US workloads).
- Prefer immutable digests; tags are not an operational control.

1.23.4 Operational considerations

- **Failure mode:** autoscalers and node rotations amplify registry outages (new nodes/pods pull images).
- **Mitigation:** pre-pull critical images, use node image caching where appropriate, and maintain a “break-glass allowlist” of verified digests.

1.23.5 Procedure (step-by-step)

1.23.5.1 Prechecks

1. Confirm scope: provider vs region vs global registry control plane issue.
2. Validate whether existing running pods are stable (avoid unnecessary restarts).
3. Confirm audit logging and incident ticket.

1.23.5.2 Execution

1. Freeze non-essential deployments and node rotations.
2. Switch workloads to known-cached/known-good digests if available (no retagging).
3. If multi-registry exists within the same residency boundary, fail over pulls via pre-configured mirrors.
4. Coordinate with security/policy owners before any temporary policy exceptions.

1.23.5.3 Verification

1. Confirm new pod scheduling does not trigger widespread ImagePullBackOff.
2. Confirm promotion pipeline state and that no cross-residency replication was enabled ad hoc.

1.23.5.4 Rollback

1. Remove temporary mirrors/exceptions once registry recovers.
2. Reconcile any emergency changes back into GitOps and close the loop with evidence.

1.23.6 Checklist

- [ ] Deploy freeze applied and communicated.
- [ ] No cross-residency artifact copying occurred.
- [ ] Post-incident: promotion trails intact and verified.

1.24 Runbook: Backups verification, restore drills, and evidence capture

1.24.1 Objectives

- Prove you can meet stated **RTO/RPO** and produce SOC2 evidence that restores are tested, not assumed.

1.24.2 When to use/avoid

Use when

- On a schedule (quarterly at minimum for critical tiers) and after major data plane or storage changes.

Avoid when

- You have not classified data by residency and tier; restore drills without data classification often cause residency/control violations.

1.24.3 Architecture notes

- Restore drills must be executable independently in EU and US, using residency-scoped keys, storage, and telemetry.
- Treat restore as a controlled change with explicit stop conditions.

1.24.4 Operational considerations

- **Tradeoff:** frequent drills cost time and money; skipping them costs you your RTO/RPO credibility.
- Capture evidence as artifacts: run logs, timestamps, restored object checksums, and sign-offs—stored within the same residency boundary.

1.24.5 Checklist

- [ ] Backup Verified prereq satisfied (freshness + integrity checks).
- [ ] Restore executed to isolated environment; blast radius controlled.
- [ ] Evidence retained per policy (SOC2) and residency boundaries respected.

1.25 Migration Playbooks and Adoption Roadmap

1.25.1 Objectives

- You move from single-cloud to multi-cloud Kubernetes with **measurable gates** that reduce blast radius and avoid “big bang” migrations.
- You preserve **SOC2 auditability** (change management evidence, least privilege, centralized audit logging) while enforcing **EU/US data residency** and **encryption everywhere**.
- You align platform scope with a **99.9% service availability** target by prioritizing multi-region operability before multi-cloud.

1.25.2 When to use/avoid

When to use

- You must meet **EU/US residency** boundaries for customer and derived data (logs/metrics/traces, backups, artifacts).
- Provider concentration risk is explicitly unacceptable (regulatory, contractual, or business continuity).
- You can commit to **repeatable operations**: GitOps, policy-as-code, DR drills, and on-call coverage.

When to avoid

- You can’t staff Day-2 operations (upgrades, incidents, DR tests) reliably; multi-cloud adds failure modes faster than it adds resilience.
- Your workloads are primarily **stateful with tight provider coupling** (managed DB semantics, storage replication constraints) and you don’t have a funded DR design.
- Your driver is “portability” without concrete requirements; Kubernetes reduces friction but does not eliminate IAM/LB/DNS/KMS/observability coupling.

1.25.3 Architecture notes

- Treat **multi-region (single provider)** as the default stepping stone; it reduces variance while validating your operational model.
- Treat **control plane tooling** (GitOps controllers, policy engines, CI/CD promotion, observability backends) as a **shared dependency** that can become a common point of failure; isolate by environment and residency boundary.
- Make tenant/team boundaries explicit: **provider → region → cluster → namespace + network policy**. This is your primary blast-radius and least-privilege enforcement hierarchy.

1.25.4 Operational considerations

- **Residency:** run separate pipelines, registries (or residency-scoped repos), telemetry backends, and KMS keyrings per residency boundary; enforce with policy checks and egress allowlists.
- **Auditability (SOC2):** require immutable artifact promotion (digest pinning), GitOps reconciliation logs, Kubernetes API audit logs, and IAM audit logs retained per policy and residency.
- **Encryption everywhere:** TLS for north-south traffic; mTLS for east-west when justified by threat model; at-rest encryption via KMS with documented rotation/expiry and emergency key access procedures.
- **Availability math:** 99.9% is ~43 minutes/month; multi-cloud designs that increase change failure rate can reduce real availability unless you gate changes and practice rollbacks.

  Assumptions

  - You already have (or will establish) landing zones/accounts/projects with baseline IAM, networking, logging, and KMS.

- You can run GitOps and policy-as-code with enforced code review and artifact promotion.
- You can keep EU and US data (including derived data) within their respective boundaries.

Non-goals

- Designing a universal “run anywhere” portability layer with no provider differences.
- Full application refactoring guidance (you should use ADRs to track and limit coupling instead).
- On-prem/hybrid as a primary model (operationally distinct).

Risks and mitigations

- **Operational overload:** mitigate by limiting cluster count, standardizing baselines, and completing multi-region before multi-cloud.
- **Residency violations via telemetry/backup/artifacts:** mitigate with residency-scoped tooling, egress allowlists, and CI policy checks that block cross-boundary endpoints.
- **Hidden provider coupling:** mitigate with ADR-backed dependency inventories and explicit fallbacks per provider edge (LB, DNS, KMS, IAM).
- **Controller blast radius (GitOps/policy):** mitigate with environment separation, progressive rollout, and break-glass procedures with tight audit logging.

1.25.5 Checklist

- [ ] Residency boundaries defined for data, telemetry, backups, and artifacts (EU/US) with explicit “no-cross” controls
- [ ] Service SLOs set (≥99.9%) with error budgets and on-call ownership per service/team boundary
- [ ] Artifact promotion uses immutable digests; rollbacks do not require rebuilds
- [ ] GitOps + policy-as-code enforced with audit logs retained per SOC2 and residency
- [ ] DR targets (RTO/RPO) documented per tier and validated through drills
- [ ] Cluster baseline standardized (identity, networking, ingress/egress, observability, security posture) before scaling out

---

1.25.6 Adoption Roadmap Phases (with gates)

1.25.6.1 Objectives

- You adopt capabilities in a sequence where each phase has **entry/exit criteria** and produces evidence for reliability and compliance.
- You reduce variance early (cluster baseline + CI/CD) before introducing additional regions/providers.

1.25.6.2 When to use/avoid

Use when you need a pragmatic path that balances delivery velocity with auditability and operability.

Avoid when leadership expects multi-cloud in a single quarter without investing in platform staff, training, and DR exercises; you will likely regress availability.

1.25.6.3 Architecture notes

- Each phase ends with: (1) operational runbooks, (2) a DR or failure-mode validation step, and (3) residency/compliance verification.
- Prefer “one new dimension at a time”: standardize clusters first, then scale GitOps/policy, then add regions, then add providers.

1.25.6.4 Operational considerations

- Maintain an **ADR log** per phase to document decisions, exceptions, and planned deprecations.
- Treat “done” as “observable and recoverable”: alerts, dashboards, SLOs, rollback paths, and DR test results.

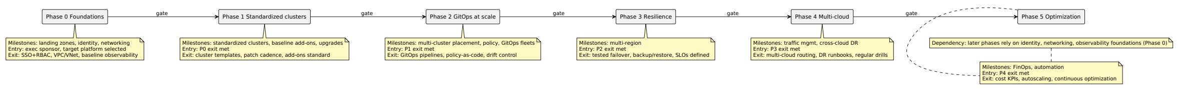

Phase 0 Foundations
gate
Phase 1 Standardized clusters
gate
Phase 2 GitOps at scale
gate
Phase 3 Resilience
gate
Phase 4 Multi-cloud
gate
Phase 5 Optimization
Milestones: landing zones, identity, networking
Entry: exec sponsor, target platform selected
Exit: SSO+RBAC, VPC/VNet, baseline observability
Milestones: standardized clusters, baseline add-ons, upgrades
Entry: P0 exit met
Exit: cluster templates, patch cadence, add-ons standard
Milestones: multi-cluster placement, policy, GitOps fleets
Entry: P1 exit met
Exit: GitOps pipelines, policy-as-code, drift control
Milestones: multi-region
Entry: P2 exit met
Exit: tested failover, backup/restore, SLOs defined
Milestones: traffic mgmt, cross-cloud DR
Entry: P3 exit met
Exit: multi-cloud routing, DR runbooks, regular drills
Dependency: later phases rely on identity, networking, observability foundations (Phase 0)
Milestones: FinOps, automation
Entry: P4 exit met
Exit: cost KPIs, autoscaling, continuous optimization

Figure (diagram_id: adoption-roadmap-phases): Phased roadmap for adopting multi-cloud container hosting with milestones and gates.

1.25.7 Readiness Assessment (what you validate before migrating)

1.25.7.1 Objectives

- You decide what to migrate first based on dependency coupling, data residency, and SLO feasibility.
- You avoid discovering provider coupling during an outage.

1.25.7.2 When to use/avoid

Use when you’re selecting candidate services and sequencing migrations.

Avoid when you don’t have an authoritative dependency view; build that first (even if imperfect) and improve iteratively.

1.25.7.3 Architecture notes

Assess each workload across:

- **Data plane coupling:** ingress/egress, DNS, LB features, storage semantics, latency sensitivity.
- **Control plane dependencies:** CI/CD, GitOps, secret management, policy, observability, identity.
- **Residency footprint:** primary data, derived data, backups, telemetry, and artifact flows.

1.25.7.4 Operational considerations

- Record results in an ADR-style inventory: “what it uses today” and “approved fallback per provider.”
- Include failure-mode review: what breaks if DNS, KMS, registry, or GitOps is unavailable?

1.25.7.4.1 Decision matrix (workload selection)

Dimension	Low risk (start here)	Medium risk	High risk (defer)
State	Stateless	Stateful with clear DR (tested)	Stateful with strong provider-native coupling

Dimension	Low risk (start here)	Medium risk	High risk (defer)
Coupling	Few external dependencies	Several managed services	Many provider-unique services and LB/DNS features
Data residency	Clearly EU-only or US-only	Mixed but separable	Cross-boundary data flows not easily separated
SLO criticality	Non-critical	Moderate	Highest-tier without proven rollback/DR
Change risk	Easy rollback	Some schema/ config coupling	Schema migrations and irreversible changes

1.25.7.5 Checklist

- [ ] Dependency inventory completed (IAM, DNS, ingress/LB, KMS, storage/ DB, observability, egress)
- [ ] Residency classification completed for data + derived data + backups + telemetry + artifacts
- [ ] SLOs and error budgets defined; alerting and on-call ownership set
- [ ] DR tiering defined with RTO/RPO and at least one successful drill for similar services

---

1.25.8 Phased Migration Playbooks (procedural)

1.25.8.1 Objectives

- You migrate services with repeatable steps: prechecks, execution, verification, and rollback.
- You minimize blast radius by migrating one boundary at a time (namespace/ cluster/region/provider).

1.25.8.2 When to use/avoid

Use when you have Phase 1–2 foundations (standard clusters + GitOps + artifact promotion).

Avoid when you still manually apply manifests or rely on long-lived credentials; you will fail SOC2 expectations and increase incident probability.

1.25.8.3 Architecture notes

- Prefer **rehosting** stateless services first: same runtime, new cluster baseline.
- Treat stateful migrations as dedicated projects with explicit RTO/RPO and DR tests.

1.25.8.4 Operational considerations

- Every playbook must produce audit evidence: Git PRs, approvals, promotion logs, and verification results.
- Enforce least privilege: per-team namespaces, workload identity, and scoped IAM roles; remove broad cluster-admin from human users.

1.25.8.5 Playbook A: Stateless service migration (single-cloud → multi-region, then multi-cloud)

1.25.8.5.1 Procedure (step-by-step)

1. **Prechecks**
   1. Confirm service SLO, dependency inventory, and residency classification are approved.
   2. Confirm container images are published as immutable digests and eligible for promotion.
   3. Confirm GitOps is healthy in source and target clusters (reconciliation not paused).
   4. Confirm observability endpoints are residency-correct (EU services to EU telemetry, US to US telemetry).
2. **Prepare**
   1. Create target namespace and baseline policies (network policy, pod security controls, resource quotas).
   2. Configure workload identity for the service and remove any static credentials from manifests.
   3. Configure ingress/egress allowlists; explicitly enumerate required egress destinations.
3. **Deploy (no traffic)**
   1. Promote the same artifact digest to the new environment via your artifact promotion process.

    2. Apply manifests via GitOps to the target cluster/region/provider with traffic weight = 0.
4. **Verification**
    1. Run conformance checks: readiness/liveness, basic synthetic transactions, dependency connectivity.
    2. Validate logs/metrics/traces arrive in the correct residency boundary and include audit context.
    3. Validate IAM: calls to cloud APIs succeed only with intended permissions (least privilege).
5. **Traffic shift (progressive delivery)**
    1. Shift 1–5% traffic; hold and observe error rates/latency against SLOs.
    2. Increase gradually to 25%, 50%, 100% if error budget impact remains acceptable.
6. **Rollback**
    1. If SLO burn rate breaches thresholds, immediately shift traffic back to the known-good endpoint.
    2. Revert GitOps change (or pin to previous digest); verify reconciliation returns to prior state.
7. **Post-migration**
    1. Remove legacy endpoints and permissions after a defined soak period.
    2. Record outcomes (success/failure modes) and update ADRs/runbooks.

1.25.8.5.2 Checklist

- [ ] Digest-pinned artifacts promoted with approvals
- [ ] Workload identity enabled; no long-lived secrets required for cloud access
- [ ] Residency validation passed for telemetry, backups (if any), and artifacts
- [ ] Progressive traffic shift and rollback tested during migration

1.25.8.6 Playbook B: “Lift and isolate” for provider-coupled dependencies

Use this when a service depends on provider-native LB/DNS/KMS/managed services that you can’t immediately replace.

1.25.8.6.1 Procedure (step-by-step)

1. **Prechecks**
    1. Document coupling in an ADR and define an acceptable fallback (even if manual).
2. **Isolate**
    1. Encapsulate provider specifics behind a stable interface (e.g., internal DNS name, gateway layer, or service abstraction).

3. **Migrate compute**
    1. Move the Kubernetes workload first while leaving the coupled dependency in place; enforce egress controls and monitor latency/cost.
4. **Verify + rollback**
    1. Verify end-to-end SLOs and audit logs; rollback by reverting traffic and GitOps state.
5. **Plan de-coupling**
    1. Schedule a follow-up milestone to reduce or remove the coupling if multi-cloud resilience is a goal.

1.25.8.6.2 Checklist

- [ ] ADR capturing coupling + fallback exists
- [ ] Egress controls and telemetry confirm cross-boundary compliance
- [ ] Latency/cost impact explicitly accepted (with a revisit date)

1.25.9 Validation and Evidence (SOC2-friendly)

1.25.9.1 Objectives

- You generate repeatable evidence that controls are operating (not just designed).
- You prove DR posture via drills, not diagrams.

1.25.9.2 When to use/avoid

Use when gating phase completion and preparing for SOC2 audits.

Avoid when evidence collection is ad-hoc; automate it to reduce toil and inconsistency.

1.25.9.3 Architecture notes

Evidence sources (per residency boundary):

- Git history and PR approvals (change management)
- Artifact promotion logs (immutability and release traceability)
- Kubernetes API audit logs + IAM audit logs (access control)
- Policy decision logs (OPA/admission decisions)
- DR drill reports (RTO/RPO outcomes)

1.25.9.4 Operational considerations

- Retention and access controls must respect residency: EU evidence stays in EU, US in US, with least-privilege access.
- Rotation/expiry: include key/cert rotation runbooks and logs showing rotations completed on schedule.

1.25.9.5 Checklist

- [ ] Automated evidence collection jobs exist per EU/US boundary
- [ ] Audit logs are immutable/append-only where feasible, access-controlled, and retained per policy
- [ ] DR drill results mapped to RTO/RPO and tracked over time
- [ ] Key/cert rotation completed and evidenced (including exceptions and break-glass use)

1.25.10 Organizational model and ownership boundaries

1.25.10.1 Objectives

- You avoid "platform as a bottleneck" while keeping guardrails strong enough for SOC2 and multi-cloud failure modes.
- You define ownership boundaries aligned to tenant/team boundaries.

1.25.10.2 When to use/avoid

Use when multiple product teams are migrating and you need consistent operations.

Avoid when responsibilities are unclear; ambiguity becomes an incident during outages and audits.

1.25.10.3 Architecture notes

- **Platform team** owns the control plane: landing zones, cluster lifecycle (CAPI or equivalent), baseline add-ons, GitOps/policy frameworks, observability platform, and DR enablement.
- **Product teams** own the data plane workloads: service configs, SLOs, runbooks, on-call, and workload-level security posture within guardrails.

1.25.10.4 Operational considerations

- Enforce least privilege: platform team does not routinely access application namespaces; product teams do not receive broad cluster-admin.
- Break-glass access exists with tight audit logging and expiry.

1.25.10.5 Checklist

- [ ] RACI defined for control plane vs data plane responsibilities
- [ ] On-call ownership mapped per service and tested via incident drills
- [ ] Break-glass procedure documented, time-bounded, audited, and periodically tested

1.26 Appendices: Checklists, Reference Configs, and Decision Records

1.26.1 Objectives

- Give you **copy/paste-ready** checklists and templates that reduce drift across cluster/region/provider.
- Provide **standard evidence artifacts** for SOC2: change history, approvals, audit logs, access reviews, and policy enforcement.
- Make decisions repeatable via lightweight **ADRs** with explicit tradeoffs and failure modes.

1.26.2 When to use/avoid

Use when

- You need consistent controls across multiple clusters and teams, and you want to **prevent “it depends” sprawl**.
- You must demonstrate **EU/US data residency**, encryption everywhere, and auditable change control.

Avoid when

- You treat these as “paper compliance.” If you don’t wire checklists into pipelines and reviews, you will drift.
- You can’t keep them current—stale checklists are worse than none because they give false confidence.

1.26.3 Architecture notes

- These appendices are **control plane adjacent**: they define the standards your control plane enforces (GitOps, policy, IAM, observability), not the workload specifics.
- Treat **EU and US as hard tenant boundaries** for customer data *and* derived data (logs/metrics/traces, backups, artifacts). Your templates must make boundary ownership explicit.

1.26.4 Operational considerations

- Run checklists as **gates**: PR templates, CI policy checks, cluster bootstrap pipelines, and quarterly access reviews.
- Store filled checklists and ADRs in a repo with immutable history (Git) and link them to incidents/changes.
- Expect failure modes: “shadow admin” access, cross-region log shipping, registry replication crossing residency boundaries, and drift from manual `kubectl`.

Assumptions

- You must meet SOC2 expectations (auditability, change control, access reviews, incident response evidence).
- You enforce **encryption in transit and at rest** with KMS-backed keys and rotation/expiry.
- You support **EU/US data residency** as a hard boundary.
- You target **99.9% availability** and use SLOs/error budgets to manage change.

Non-goals

- This is not a vendor-specific blueprint or “run anywhere with zero variance.”
- This is not a refactor guide for applications.
- This does not cover on-prem hardware lifecycle or physical networking (hybrid is mostly out of scope).

Risks and mitigations

- **Checklist theater** → Mitigate by making controls enforceable (OPA/admission, CI gates) and sampling for evidence.

- **Residency violations via telemetry/backup/artifacts** → Mitigate with residency-scoped pipelines, allowlists, and region-scoped keys and buckets. - Failure mode: “global” observability backend silently stores EU traces in US.
- **Operational overload** → Mitigate by limiting cluster count, standardizing interfaces, and starting multi-region before multi-provider.

1.27 ADR templates

1.27.1 Objectives

- Make provider-specific choices explicit and reviewable.
- Capture tradeoffs, failure modes, and rollback paths in a format you can audit.

1.27.2 When to use/avoid

Use when

- You introduce or change a shared interface: ingress, identity, KMS, DNS, GitOps tooling, registry layout, DR posture.
- You accept a known limitation (e.g., stateful DR constraints) and need a paper trail.

Avoid when

- The decision is reversible and low-impact (don’t spam ADRs); use PR discussions instead.
- You won’t maintain the ADR index (unindexed ADRs are dead weight).

1.27.3 Architecture notes

- ADRs should map cleanly to control plane/data plane boundaries and tenant/team boundaries.
- Each ADR must name the **residency scope** (EU/US/both) and the **blast radius** (single cluster, region, provider, global).

1.27.4 Operational considerations

- Require ADR IDs in PR titles for changes that affect shared controls.

- Review ADRs quarterly; mark superseded ADRs explicitly.

1.27.5 Checklist

- [ ] Stored in version control with immutable history
- [ ] Includes security, audit logging, and rotation/expiry implications
- [ ] Includes operational ownership and on-call impact
- [ ] Includes rollback and verification steps

1.27.5.1 ADR: Standard template (copy/paste)

```
# ADR-YYYYMMDD-XX: <Title>

## Status
Proposed | Accepted | Deprecated | Superseded by ADR-...

## Context
- Problem statement:
- Drivers (SLOs, SOC2 evidence needs, residency constraints, cost, opera
- In-scope clusters/regions/providers:
- Tenant/team boundaries impacted:

## Decision
- What you will do:
- Interfaces introduced/changed (APIs, CRDs, pipelines, runbooks):

## Options considered
1) Option A
2) Option B
3) Option C

## Tradeoffs
- Reliability:
- Security (least privilege, audit logging, rotation/expiry):
- Residency (EU/US derived data: logs/metrics/traces/backups/artifacts):
- Operability (day-2, upgrades, failure handling):
- Cost (steady state + failure mode cost):

## Failure modes and blast radius
- Failure mode:
- Expected blast radius:
```

```
- Detection signals:
- Manual mitigations:
- Automated mitigations:

## Rollback plan
- Preconditions:
- Steps:
- Verification:

## Consequences
- What becomes easier:
- What becomes harder:
- Follow-ups (tickets/owners/dates):

## References
- Related ADRs:
- Runbooks:
- Threat model notes:
```

1.28 Baseline cluster checklist (networking, IAM, logging, policies, upgrades)

1.28.1 Objectives

- Provide a **minimum secure and operable** baseline per cluster/region/provider.
- Reduce drift and “snowflake clusters” that break DR and audits.

1.28.2 When to use/avoid

Use when

- Bootstrapping a new cluster (any provider).
- Quarterly compliance attestation and day-2 health checks.

Avoid when

- You’re validating application-specific requirements (use workload readiness checklists in runbooks instead).

1.28.3 Architecture notes

- Treat the cluster as a **data plane** with a separate, auditable **control plane** workflow (GitOps).
- Residency controls must apply to: **logs/metrics/traces**, backups, artifacts, and KMS keys—not just primary databases.

1.28.4 Operational considerations

- Run this checklist as part of cluster lifecycle (CAPI or provider tooling) and then continuously via policy and monitoring.
- Failure mode to expect: baseline passes at creation, then drift occurs via manual changes; mitigate with GitOps + admission + periodic drift detection.

1.28.5 Checklist (baseline)

1.28.5.1 1) Identity and access (IAM) + auditability

- [ ] Human access uses SSO + MFA; no shared accounts.
- [ ] Cluster admin is **break-glass only**, time-bounded, and fully audit-logged.
- [ ] Workload identity enabled (provider equivalent of IRSA/workload identity); **no long-lived cloud keys** in Secrets.
- [ ] Least privilege roles per tenant/team boundary; separation of duties for platform vs app teams.
- [ ] Audit logs enabled for: cloud IAM, KMS, storage, network changes, and Kubernetes API server.
- [ ] Access review process defined (quarterly) with evidence retention.

1.28.5.2 2) Network baseline (ingress/egress, IPAM, segmentation)

- [ ] IPAM plan documented per region/provider; no overlapping CIDRs across connected networks.
- [ ] NetworkPolicies enforced (default deny for tenant namespaces; explicit allow).
- [ ] Egress is controlled (egress gateway or NAT + allowlists); DNS egress is logged.
- [ ] Ingress controller/gateway uses TLS 1.2+; certificates rotated automatically; HSTS where applicable.
- [ ] Private cluster endpoints where feasible; control plane endpoint access restricted (CIDR allowlist / private links).

1.28.5.3 3) Encryption everywhere (at rest + in transit)

- [ ] Node disk encryption on by default.
- [ ] etcd (or managed control plane equivalent) encrypted at rest.
- [ ] Kubernetes Secrets encryption at rest enabled (KMS-backed where supported).
- [ ] TLS in transit for: ingress, service-to-service (mTLS if required by threat model), and control plane connections.
- [ ] Key rotation/expiry documented and tested; emergency key revoke procedure exists.

1.28.5.4 4) Observability baseline (metrics/logs/traces) with residency

- [ ] Cluster emits metrics/logs/traces to a **region-scoped** backend (EU stays in EU, US stays in US).
- [ ] Derived data classification defined (PII risk in logs/traces); redaction/enrichment policies enforced.
- [ ] SLOs defined for critical services; error budget policy defined for change velocity.
- [ ] Alerting routes are tested; on-call ownership per tenant/team boundary is clear.

1.28.5.5 5) Policy and governance

- [ ] Admission control in place (OPA or equivalent) enforcing: privileged pods restrictions, image provenance requirements, allowed registries, required labels/annotations, and resource limits.
- [ ] Pod Security enforced (baseline/restricted profile as appropriate).
- [ ] Namespace and RBAC templates standardized; no cluster-wide permissions without justification.
- [ ] Supply chain policies: signed images, SBOM required, immutable digest pinning in deployment manifests.

1.28.5.6 6) Reliability and upgrades (99.9% target alignment)

- [ ] Multi-AZ/zone worker pool strategy documented; disruption budgets set for critical workloads.
- [ ] Cluster upgrade process defined (minor version cadence, max skew policy, maintenance windows).
- [ ] Backup/restore procedure exists for cluster state (and critical platform components), scoped to region/residency.

- [ ] DR posture documented (RTO/RPO per tier); game days scheduled and results retained.

1.29 Provider capability matrix (high-level)

1.29.1 Objectives

- Give you a **decision matrix** without pretending providers are interchangeable.
- Make gaps visible so you can standardize interfaces and document exceptions.

1.29.2 When to use/avoid

Use when

- Selecting baseline primitives per provider: workload identity, KMS integration, load balancing model, private cluster support, audit logging export.
- Writing ADRs that justify provider-specific variance.

Avoid when

- You need exact SKU/feature parity; validate in a lab because managed services change frequently.

1.29.3 Architecture notes

- Focus on capabilities that affect control plane/data plane separation, tenant isolation, and residency.
- Prefer **portable abstractions** (Kubernetes APIs, GitOps, OPA) and treat provider features as edges.

1.29.4 Operational considerations

- Keep this matrix versioned; tie updates to ADRs.
- Failure mode: teams assume a capability exists everywhere; mitigate via “minimum common baseline” + exception process.

1.29.5 Capability matrix (example shape; fill per your providers)

Capability area	Provider A	Provider B	Provider C	Standard interface / mitigation
Workload identity	Yes/No + notes	Yes/No + notes	Yes/No + notes	Use KSA-to-IAM mapping; forbid static keys; audit token usage
KMS integration for Secrets	Yes/No	Yes/No	Yes/No	Encrypt Secrets at rest; document rotation; residency-scoped keys
Private control plane endpoint	Yes/No	Yes/No	Yes/No	Restrict API access; log all auth; break-glass procedure
L4/L7 ingress model	Notes	Notes	Notes	Prefer Gateway API; document provider-specific controllers
NetworkPolicy enforcement	Notes	Notes	Notes	Choose CNI that enforces policies; test default-deny
Audit logging export	Notes	Notes	Notes	Centralize per residency boundary; retention per SOC2
Multi-zone nodes	Notes	Notes	Notes	Require for 99.9%; define disruption budgets
Registry + artifact replication controls	Notes	Notes	Notes	Residency-scoped registries; explicit promotion, no global replication

1.30 Example GitOps repo structure and environment overlays

1.30.1 Objectives

- Give you a repo layout that supports **immutable artifact promotion**, auditability, and EU/US separation.
- Make tenant/team boundaries explicit so least privilege is enforceable.

1.30.2 When to use/avoid

Use when

- You operate multiple clusters and need consistent promotion across dev/stage/prod.
- You need SOC2-friendly evidence for “who changed what, when, and why.”

Avoid when

- You have a single cluster and no on-call; GitOps overhead can be unjustified (but you still need auditable change control somehow).

1.30.3 Architecture notes

- Keep “platform” and “workloads” separated; platform changes have higher blast radius.
- Use overlays per **environment + residency boundary** (e.g., `prod-eu`, `prod-us`) to prevent accidental cross-boundary references.

1.30.4 Operational considerations

- Failure mode: promotion pulls from “latest” tags → mitigate by pinning **image digests** and enforcing in admission policy.
- Failure mode: a shared “global” observability endpoint breaks residency → mitigate by separate overlays, separate credentials, and policy checks.

1.30.5 Reference repo layout (example)

```
repo/
  adrs/
    ADR-20260101-01-example.md
  clusters/
```

```
    prod-eu/
      provider-a-region-1/
        kustomization.yaml
        platform/
        tenants/
    prod-us/
      provider-b-region-2/
        kustomization.yaml
        platform/
        tenants/
  platform/
    ingress/
    policy/
    observability/
    identities/
  workloads/
    team-a/
      service-1/
        base/
        overlays/
          dev/
          stage/
          prod-eu/
          prod-us/
  pipelines/
    artifact-promotion/
    policy-checks/
  docs/
    runbooks/
    checklists/
```

Minimum policy gates to enforce

- Digest-pinned images only.
- Allowed registries per residency boundary.
- Required labels: `tenant`, `team`, `data-residency`, `owner`, `risk-tier`.
- No privileged pods / host mounts without exception ADR.

1.31 Glossary and further reading

1.31.1 Objectives

- Keep terminology consistent (control plane/data plane, tenant/team boundaries, cluster/region/provider).
- Reduce ambiguity during incidents and audits.

1.31.2 When to use/avoid

Use when

- Onboarding new operators or app teams.
- Writing ADRs, runbooks, and incident timelines.

Avoid when

- You need deep conceptual training; link out to canonical docs rather than expanding this appendix endlessly.

1.31.3 Architecture notes

- This glossary must match the document's terminology to support cross-references and SOC2 evidence clarity.

1.31.4 Operational considerations

- Treat glossary updates like API changes: if you rename concepts, you risk operational confusion during incidents.

1.31.5 Checklist

- [ ] Terms used in alerts/runbooks match glossary
- [ ] Residency terminology is unambiguous (EU vs US boundaries)
- [ ] Control plane vs data plane ownership is explicit

Glossary: Use the shared glossary provided in this document (see global Glossary section).

Further reading (non-vendor-specific where possible)

- Kubernetes: API conventions, RBAC, NetworkPolicy, Gateway API.
- SLSA + OCI image specs; SBOM formats (CycloneDX/SPDX).

- OPA/Gatekeeper or equivalent admission control patterns.
- SRE: SLOs/error budgets, incident management, chaos testing.
- SOC2: change management, access control, logging/monitoring, evidence retention.